Gauche Users' Reference

version 0.9.9

Shiro Kawai (shiro@acm.org)

Table of Contents

1 Introduction **1**
- 1.1 Overview of Gauche 1
- 1.2 Notations 2
  - 1.2.1 Entry format 2
  - 1.2.2 Names and namespaces 4

2 Concepts **5**
- 2.1 Standard conformance 5
- 2.2 Multibyte strings 11
- 2.3 Multibyte scripts 11
- 2.4 Case-sensitivity 12
- 2.5 Integrated object system 12
- 2.6 Module system 14
- 2.7 Compilation 14

3 Programming in Gauche **16**
- 3.1 Invoking Gosh 16
- 3.2 Interactive development 20
  - 3.2.1 Working in REPL 21
- 3.3 Writing Scheme scripts 25
- 3.4 Debugging 27
- 3.5 Using platform-dependent features 29
- 3.6 Profiling and tuning 30
  - 3.6.1 Using profiler 30
  - 3.6.2 Performance tips 31
- 3.7 Writing Gauche modules 32
- 3.8 Using extension packages 33
- 3.9 Building standalone executables 35

4 Core syntax **38**
- 4.1 Lexical structure 38
  - 4.1.1 Sharp syntax 39
  - 4.1.2 Hash-bang token 41
- 4.2 Literals 41
- 4.3 Making procedures 42
- 4.4 Assignments 47
- 4.5 Conditionals 49
- 4.6 Binding constructs 52
- 4.7 Sequencing 55
- 4.8 Iteration 56
- 4.9 Quasiquotation 59
- 4.10 Definitions 61
  - 4.10.1 Into the Scheme-Verse 65
- 4.11 Inclusions 67
- 4.12 Feature conditional 68
- 4.13 Modules 71
  - 4.13.1 Module semantics 71

4.13.2 Modules and libraries ... 72
4.13.3 Defining and selecting modules ... 73
4.13.4 Using modules ... 73
4.13.5 Module inheritance ... 75
4.13.6 Module introspection ... 76
4.13.7 Predefined modules ... 78

5 Macros ... 80

5.1 Why hygienic? ... 80
5.2 Hygienic macros ... 83
5.2.1 Syntax-rules macro transformer ... 84
5.2.2 Explicit-renaming macro transformer ... 86
5.3 Traditional macros ... 90
5.4 Debugging macros ... 90
5.4.1 Tracing macro expansion ... 91
5.4.2 Expanding macros manually ... 93
5.5 Macro utilities ... 95

6 Core library ... 96

6.1 Types and classes ... 96
6.2 Equality and comparison ... 97
6.2.1 Equality ... 97
6.2.2 Comparison ... 99
6.2.3 Hashing ... 100
6.2.4 Basic comparators ... 103
6.2.4.1 Comparator class and constructors ... 103
6.2.4.2 Comparator predicates and accessors ... 104
6.2.4.3 Predefined comparators ... 106
6.2.4.4 Combining comparators ... 108
6.3 Numbers ... 108
6.3.1 Number classes ... 109
6.3.2 Numerical predicates ... 109
6.3.3 Numerical comparison ... 111
6.3.4 Arithmetics ... 112
6.3.5 Numerical conversions ... 118
6.3.6 Basic bitwise operations ... 121
6.3.7 Endianness ... 123
6.4 Booleans ... 124
6.5 Undefined values ... 124
6.6 Pairs and lists ... 125
6.6.1 Pair and null class ... 125
6.6.2 List predicates ... 125
6.6.3 List constructors ... 126
6.6.4 List accessors and modifiers ... 127
6.6.5 Walking over lists ... 131
6.6.6 Other list procedures ... 134
6.6.7 Association lists ... 135
6.7 Symbols ... 137
6.8 Keywords ... 139
6.8.1 Keyword and symbol integration ... 141
6.9 Identifiers ... 142
6.10 Characters ... 143
6.11 Character Set ... 147

6.12 Strings . . . 153
6.12.1 String syntax . . . 153
6.12.2 String Predicates . . . 154
6.12.3 String Constructors . . . 155
6.12.4 String interpolation . . . 155
6.12.5 String Accessors & Modifiers . . . 157
6.12.6 String Comparison . . . 158
6.12.7 String utilities . . . 158
6.12.8 Incomplete strings . . . 161
6.13 Regular expressions . . . 162
6.13.1 Regular expression syntax . . . 162
6.13.2 Using regular expressions . . . 165
6.13.3 Inspecting and assembling regular expressions . . . 172
6.14 Vectors . . . 174
6.15 Hashtables . . . 177
6.16 Treemaps . . . 182
6.17 Weak pointers . . . 186
6.18 Procedures and continuations . . . 187
6.18.1 Procedure class and applicability . . . 188
6.18.2 Universal accessor . . . 189
6.18.3 Combinators . . . 190
6.18.4 Optional argument parsing . . . 192
6.18.5 Procedure arity . . . 194
6.18.6 Applicable objects . . . 195
6.18.7 Continuations . . . 195
6.18.8 Multiple values . . . 197
6.18.9 Folding generated values . . . 197
6.19 Lazy evaluation . . . 199
6.19.1 Delay, force and lazy . . . 199
6.19.2 Lazy sequences . . . 200
6.20 Exceptions . . . 204
6.20.1 Exception handling overview . . . 205
6.20.2 Signaling exceptions . . . 207
6.20.3 Handling exceptions . . . 208
6.20.4 Conditions . . . 212
6.21 Eval and repl . . . 216
6.22 Input and Output . . . 217
6.22.1 Ports . . . 217
6.22.2 Port and threads . . . 218
6.22.3 Common port operations . . . 218
6.22.4 File ports . . . 221
6.22.5 String ports . . . 224
6.22.6 Coding-aware ports . . . 226
6.22.7 Input . . . 227
6.22.7.1 Reading data . . . 227
6.22.7.2 Reader lexical mode . . . 228
6.22.7.3 Read-time constructor . . . 229
6.22.7.4 Input utility functions . . . 231
6.22.8 Output . . . 231
6.22.8.1 Layers of output routines . . . 231
6.22.8.2 Output controls . . . 232
6.22.8.3 Object output . . . 233
6.22.8.4 Formatting output . . . 235
6.22.8.5 Low-level output . . . 239

6.23 Loading Programs 239
6.23.1 Loading Scheme file 239
6.23.2 Load dynamic library 241
6.23.3 Require and provide 241
6.23.4 Autoload 242
6.23.5 Operations on libraries 243
6.24 Sorting and merging 244
6.25 System interface 246
6.25.1 Program termination 246
6.25.2 Command-line arguments 248
6.25.3 Environment inquiry 248
6.25.4 Filesystems 250
6.25.4.1 Directories 250
6.25.4.2 Directory manipulation 252
6.25.4.3 Pathnames 253
6.25.4.4 File stats 255
6.25.4.5 Other file operations 257
6.25.5 Unix groups and users 257
6.25.6 Locale 259
6.25.7 Signal 260
6.25.7.1 Signals and signal sets 260
6.25.7.2 Sending signals 262
6.25.7.3 Handling signals 262
6.25.7.4 Masking and waiting signals 264
6.25.7.5 Signals and threads 265
6.25.8 System inquiry 266
6.25.9 Time 269
6.25.10 Process management 271
6.25.11 I/O multiplexing 274
6.25.12 Garbage collection 275
6.25.13 Miscellaneous system calls 275
6.26 Development helper API 277
6.26.1 Debugging aid 277
6.26.2 Profiler API 278

7 Object system 279

7.1 Introduction to the object system 279
7.2 Class 286
7.2.1 Defining class 286
7.2.2 Inheritance 288
7.2.3 Class object 290
7.2.4 Slot definition object 291
7.2.5 Class redefinition 292
7.2.6 Class definition examples 293
7.3 Instance 295
7.3.1 Creating instance 295
7.3.2 Accessing instance 296
7.3.3 Changing classes 297
7.4 Generic function and method 299
7.5 Metaobject protocol 301
7.5.1 Class instantiation 301
7.5.2 Customizing slot access 303
7.5.3 Method instantiation 307
7.5.4 Customizing method application 307

8 Library modules - Overview ... 308
8.1 Finding libraries you need ... 308
8.1.1 Library directory - data containers ... 309
8.1.2 Library directory - string and character ... 310
8.1.3 Library directory - data exchange ... 310
8.1.4 Library directory - files ... 311
8.1.5 Library directory - processes and threads ... 311
8.1.6 Library directory - networking ... 311
8.1.7 Library directory - input and output ... 311
8.1.8 Library directory - time ... 312
8.1.9 Library directory - bits and bytes ... 312
8.2 Naming convention of libraries ... 312
8.3 Obsolete and superseded modules ... 313

9 Library modules - Gauche extensions ... 315
9.1 `gauche.array` - Arrays ... 315
9.2 `gauche.base` - Importing gauche built-ins ... 322
9.3 `gauche.cgen` - Generating C code ... 323
9.3.1 Generating C source files ... 323
9.3.2 Generating Scheme literals ... 327
9.3.3 Conversions between Scheme and C ... 328
9.3.4 CiSE - C in S expression ... 331
9.3.4.1 CiSE overview ... 331
9.3.4.2 CiSE syntax ... 332
9.3.4.3 CiSE procedures ... 335
9.3.5 Stub generation ... 336
9.4 `gauche.charconv` - Character Code Conversion ... 339
9.4.1 Supported character encoding schemes ... 340
9.4.2 Autodetecting the encoding scheme ... 341
9.4.3 Conversion ports ... 342
9.5 `gauche.collection` - Collection framework ... 344
9.5.1 Mapping over collection ... 345
9.5.2 Selection and searching in collection ... 347
9.5.3 Miscellaneous operations on collection ... 349
9.5.4 Fundamental iterator creators ... 349
9.5.5 Implementing collections ... 351
9.6 `gauche.config` - Configuration parameters ... 352
9.7 `gauche.configure` - Generating build files ... 352
9.7.1 Structure of configure script and build files ... 353
9.7.2 Configure API ... 354
9.8 `gauche.connection` - Connection framework ... 365
9.9 `gauche.dictionary` - Dictionary framework ... 366
9.9.1 Generic functions for dictionaries ... 367
9.9.2 Generic dictionaries ... 369
9.10 `gauche.fcntl` - Low-level file operations ... 370
9.11 `gauche.generator` - Generators ... 372
9.11.1 Generator constructors ... 372
9.11.2 Generator operations ... 376
9.11.3 Generator consumers ... 381
9.12 `gauche.hook` - Hooks ... 383
9.13 `gauche.interactive` - Utilities for interactive session ... 384
9.14 `gauche.lazy` - Lazy sequence utilities ... 386
9.15 `gauche.listener` - Listener ... 389

9.16 gauche.logger - User-level logging 393
9.17 gauche.mop.propagate - Propagating slot access 395
9.18 gauche.mop.singleton - Singleton 396
9.19 gauche.mop.validator - Slot with validator 397
9.20 gauche.net - Networking 398
9.20.1 Socket address 398
9.20.2 High-level network functions 400
9.20.3 Low-level socket interface 403
9.20.4 Netdb interface 408
9.21 gauche.package - Package metainformation 411
9.22 gauche.parameter - Parameters 411
9.23 gauche.parseopt - Parsing command-line options 413
9.24 gauche.partcont - Partial continuations 417
9.25 gauche.process - High-level process interface 421
9.25.1 Running subprocess 421
9.25.2 Running process pipeline 426
9.25.3 Process object 427
9.25.4 Process ports 429
9.25.5 Process connection 433
9.26 gauche.record - Record types 433
9.26.1 Introduction 433
9.26.2 Syntactic Layer 434
9.26.3 Inspection layer 436
9.26.4 Procedural layer 437
9.26.5 Pseudo record types 438
9.27 gauche.reload - Reloading modules 439
9.28 gauche.selector - Simple dispatcher 440
9.29 gauche.sequence - Sequence framework 441
9.29.1 Fundamental sequence accessors 441
9.29.2 Slicing sequence 442
9.29.3 Mapping over sequences 442
9.29.4 Other operations over sequences 443
9.29.5 Implementing sequence 448
9.30 gauche.syslog - Syslog 448
9.31 gauche.termios - Terminal control 449
9.31.1 Posix termios interface 449
9.31.2 Common high-level terminal control 452
9.32 gauche.test - Unit Testing 452
9.33 gauche.threads - Threads 457
9.33.1 Thread programming tips 458
9.33.2 Thread procedures 459
9.33.3 Synchronization primitives 462
9.33.4 Thread exceptions 467
9.34 gauche.time - Measure timings 468
9.35 gauche.unicode - Unicode utilities 471
9.35.1 Unicode transfer encodings 472
9.35.2 Unicode text segmentation 474
9.35.3 Full string case conversion 475
9.35.4 East asian width property 476
9.36 gauche.uvector - Uniform vectors 476
9.36.1 Uvector basic operations 477
9.36.2 Uvector conversion operations 484
9.36.3 Uvector numeric operations 486
9.36.4 Uvector block I/O 488

9.36.5 Bytevector compatibility 490
9.37 gauche.version - Comparing version numbers 491
9.38 gauche.vport - Virtual ports 493

10 Library modules - R7RS standard libraries 499

10.1 R7RS integration 499
10.1.1 Traveling between two worlds back and forth 499
10.1.2 Three import forms 501
10.2 R7RS small language 503
10.2.1 R7RS library form 503
10.2.2 scheme.base - R7RS base library 504
10.2.3 scheme.case-lambda - R7RS case-lambda 507
10.2.4 scheme.char - R7RS char library 507
10.2.5 scheme.complex - R7RS complex numbers 508
10.2.6 scheme.cxr - R7RS cxr accessors 508
10.2.7 scheme.eval - R7RS eval 508
10.2.8 scheme.file - R7RS file library 509
10.2.9 scheme.inexact - R7RS inexact numbers 509
10.2.10 scheme.lazy - R7RS lazy evaluation 509
10.2.11 scheme.load - R7RS load 510
10.2.12 scheme.process-context - R7RS process context 510
10.2.13 scheme.read - R7RS read 510
10.2.14 scheme.repl - R7RS repl 510
10.2.15 scheme.time - R7RS time 511
10.2.16 scheme.write - R7RS write 511
10.2.17 scheme.r5rs - R5RS compatibility 512
10.3 R7RS large 512
10.3.1 scheme.list - R7RS lists 512
10.3.2 scheme.vector - R7RS vectors 517
10.3.3 scheme.vector.@ - R7RS uniform vectors 522
10.3.4 scheme.sort - R7RS sort 522
10.3.5 scheme.set - R7RS sets 525
10.3.6 scheme.charset - R7RS character sets 533
10.3.6.1 Character-set constructors 533
10.3.6.2 Character-set comparison 534
10.3.6.3 Character-set iteration 534
10.3.6.4 Character-set query 535
10.3.6.5 Character-set algebra 535
10.3.7 scheme.hash-table - R7RS hash tables 536
10.3.8 scheme.ideque - R7RS immutable deques 540
10.3.9 scheme.generator - R7RS generators 542
10.3.10 scheme.lseq - R7RS lazy sequences 544
10.3.11 scheme.stream - R7RS stream 546
10.3.12 scheme.box - R7RS boxes 547
10.3.13 scheme.list-queue - R7RS list queues 547
10.3.14 scheme.ephemeron - R7RS ephemeron 550
10.3.15 scheme.comparator - R7RS comparators 551
10.3.16 scheme.regex - R7RS regular expressions 552
10.3.17 scheme.mapping - R7RS mappings 563
10.3.17.1 Mappings 563
10.3.17.2 Hashmaps 570
10.3.18 scheme.division - R7RS integer division 574
10.3.19 scheme.bitwise - R7RS bitwise operations 575
10.3.20 scheme.fixnum - R7RS fixnums 579

10.3.21 `scheme.flonum` - R7RS flonum 581

11 Library modules - SRFIs 590

11.1 `srfi-1` - List library 590
11.2 `srfi-4` - Homogeneous vectors 590
11.3 `srfi-5` - A compatible let form with signatures and rest arguments 590
11.4 `srfi-7` - Feature-based program configuration language 590
11.5 `srfi-13` - String library 591
11.5.1 General conventions 591
11.5.2 String predicates 592
11.5.3 String Constructors 592
11.5.4 String selection 593
11.5.5 String comparison 594
11.5.6 String Prefixes & Suffixes 595
11.5.7 String searching 596
11.5.8 String case mapping 596
11.5.9 String reverse & append 597
11.5.10 String mapping 597
11.5.11 String rotation 598
11.5.12 Other string operations 599
11.5.13 String filtering 599
11.5.14 Low-level string procedures 600
11.6 `srfi-14` - Character-set library 600
11.7 `srfi-19` - Time data types and procedures 600
11.7.1 Time types 600
11.7.2 Time queries 601
11.7.3 Time procedures 601
11.7.4 Date 602
11.7.5 Date reader and writer 604
11.8 `srfi-27` - Sources of Random Bits 605
11.9 `srfi-29` - Localization 606
11.10 `srfi-37` - args-fold: a program argument processor 607
11.11 `srfi-41` - Streams 609
11.12 `srfi-42` - Eager comprehensions 610
11.13 `srfi-43` - Vector library (legacy) 615
11.14 `srfi-55` - Requiring extensions 616
11.15 `srfi-60` - Integers as bits 616
11.16 `srfi-66` - Octet vectors 618
11.17 `srfi-69` - Basic hash tables 618
11.18 `srfi-74` - Octet-addressed binary blocks 620
11.19 `srfi-98` - Accessing environment variables 622
11.20 `srfi-106` - Basic socket interface 623
11.21 `srfi-111` - Boxes 625
11.22 `srfi-112` - Environment inquiry 625
11.23 `srfi-113` - Sets and bags 626
11.24 `srfi-114` - Comparators 626
11.25 `srfi-115` - Sets and bags 631
11.26 `srfi-117` - Queues based on lists 631
11.27 `srfi-118` - Simple adjustable-size strings 631
11.28 `srfi-127` - Lazy sequence (srfi) 632
11.29 `srfi-132` - Sort library 632
11.30 `srfi-133` - Vector library 632
11.31 `srfi-141` - Integer division 632

11.32 srfi-143 - Fixnums ... 632
11.33 srfi-146 - Mappings and hashmaps ... 632
11.34 srfi-151 - Bitwise operations ... 633
11.35 srfi-152 - String library (reduced) ... 633
11.36 srfi-154 - First-class dynamic extents ... 635
11.37 srfi-158 - Generators and accumulators ... 636
11.38 srfi-160 - Homogeneous numeric vector libraries ... 636
11.39 srfi-162 - Comparator sublibrary ... 636
11.40 srfi-173 - Hooks (srfi) ... 637

12 Library modules - Utilities ... 638

12.1 binary.io - Binary I/O ... 638
12.2 binary.pack - Packing binary data ... 641
12.3 compat.chibi-test - Running Chibi-scheme test suite ... 644
12.4 compat.norational - Rational-less arithmetic ... 644
12.5 control.job - A common job descriptor for control modules ... 645
12.6 control.thread-pool - Thread pools ... 646
12.7 crypt.bcrypt - Password hashing ... 648
12.8 data.cache - Cache ... 649
12.9 data.heap - Heap ... 652
12.10 data.ideque - Immutable deques ... 654
12.11 data.imap - Immutable map ... 654
12.12 data.queue - Queue ... 656
12.13 data.random - Random data generators ... 660
12.14 data.ring-buffer - Ring buffer ... 665
12.15 data.sparse - Sparse data containers ... 666
12.15.1 Sparse vectors ... 667
12.15.2 Sparse matrixes ... 669
12.15.3 Sparse tables ... 671
12.16 data.trie - Trie ... 672
12.17 dbi - Database independent access layer ... 676
12.17.1 DBI user API ... 677
12.17.2 Writing drivers for DBI ... 680
12.18 dbm - Generic DBM interface ... 682
12.18.1 Opening and closing a dbm database ... 683
12.18.2 Accessing a dbm database ... 685
12.18.3 Iterating on a dbm database ... 685
12.18.4 Managing dbm database instance ... 686
12.18.5 Dumping and restoring dbm database ... 686
12.18.6 Writing a dbm implementation ... 687
12.19 dbm.fsdbm - File-system dbm ... 687
12.20 dbm.gdbm - GDBM interface ... 688
12.21 dbm.ndbm - NDBM interface ... 690
12.22 dbm.odbm - Original DBM interface ... 690
12.23 file.filter - Filtering file content ... 691
12.24 file.util - Filesystem utilities ... 692
12.24.1 Directory utilities ... 693
12.24.2 Pathname utilities ... 696
12.24.3 File attribute utilities ... 698
12.24.4 File operations ... 699
12.24.5 Temporary files and directories ... 701
12.24.6 Lock files ... 702
12.25 math.const - Mathematic constants ... 704

12.26 `math.mt-random` - Mersenne Twister Random number generator ... 704
12.27 `math.prime` - Prime numbers ... 705
12.28 `os.windows` - Windows support ... 708
12.28.1 Windows dialogs ... 708
12.28.2 Windows console API ... 709
12.29 `rfc.822` - RFC822 message parsing ... 712
12.30 `rfc.base64` - Base64 encoding/decoding ... 716
12.31 `rfc.cookie` - HTTP cookie handling ... 717
12.32 `rfc.ftp` - FTP client ... 718
12.33 `rfc.hmac` - HMAC keyed-hashing ... 721
12.34 `rfc.http` - HTTP ... 721
12.35 `rfc.icmp` - ICMP packets ... 727
12.36 `rfc.ip` - IP packets ... 727
12.37 `rfc.json` - JSON parsing and construction ... 728
12.38 `rfc.md5` - MD5 message digest ... 730
12.39 `rfc.mime` - MIME message handling ... 730
12.40 `rfc.quoted-printable` - Quoted-printable encoding/decoding ... 736
12.41 `rfc.sha` - SHA message digest ... 736
12.42 `rfc.tls` - Transport layer security ... 737
12.43 `rfc.uri` - URI parsing and construction ... 738
12.44 `rfc.zlib` - zlib compression library ... 743
12.45 `slib` - SLIB interface ... 746
12.46 `sxml.ssax` - Functional XML parser ... 747
12.46.1 SSAX data types ... 747
12.46.2 SSAX low-level parsing code ... 750
12.46.3 SSAX higher-level parsers and scanners ... 753
12.46.4 SSAX Highest-level parsers - XML to SXML ... 755
12.47 `sxml.sxpath` - SXML query language ... 757
12.47.1 SXPath basic converters and applicators ... 758
12.47.2 SXPath query language ... 761
12.47.3 SXPath extension ... 764
12.48 `sxml.tools` - Manipulating SXML structure ... 766
12.48.1 SXML predicates ... 766
12.48.2 SXML accessors ... 766
12.48.3 SXML modifiers ... 768
12.48.4 SXPath auxiliary utilities ... 770
12.48.5 SXML to markup conversion ... 770
12.49 `sxml.serializer` - Serializing XML and HTML from SXML ... 771
12.49.1 Simple SXML serializing ... 771
12.49.2 Custom SXML serializing ... 771
12.50 `text.console` - Text terminal control ... 773
12.51 `text.csv` - CSV tables ... 776
12.52 `text.diff` - Calculate difference of text streams ... 779
12.53 `text.edn` - EDN parsing and construction ... 780
12.54 `text.gettext` - Localized messages ... 783
12.55 `text.html-lite` - Simple HTML document construction ... 784
12.56 `text.parse` - Parsing input stream ... 786
12.57 `text.progress` - Showing progress on text terminals ... 788
12.58 `text.sql` - SQL parsing and construction ... 790
12.59 `text.template` - Simple template expander ... 790
12.60 `text.tr` - Transliterate characters ... 791
12.61 `text.tree` - Lazy text construction ... 793
12.62 `util.combinations` - Combination library ... 794
12.63 `util.digest` - Message digester framework ... 795

12.64 util.dominator - Calculate dominator tree ... 796
12.65 util.isomorph - Determine isomorphism ... 797
12.66 util.lcs - The longest common subsequence ... 798
12.67 util.levenshtein - Levenshtein edit distance ... 799
12.68 util.match - Pattern matching ... 801
12.69 util.record - SLIB-compatible record type ... 806
12.70 util.relation - Relation framework ... 807
12.71 util.stream - Stream library ... 809
12.71.1 Stream primitives ... 809
12.71.2 Stream constructors ... 810
12.71.3 Stream binding ... 813
12.71.4 Stream consumers ... 814
12.71.5 Stream operations ... 814
12.72 util.toposort - Topological sort ... 818
12.73 util.unification - Unification ... 818
12.74 www.cgi - CGI utility ... 819
12.75 www.cgi.test - CGI testing ... 824
12.76 www.css - CSS parsing and construction ... 825

Appendix A References ... 829

Appendix B C to Scheme mapping ... 832

Appendix C Function and Syntax Index ... 843

Appendix D Module Index ... 882

Appendix E Lexical syntax index ... 885

Appendix F Class Index ... 886

Appendix G Variable Index ... 889

1 Introduction

This is a users' guide and reference manual of the Gauche Scheme. Here I tried to describe the implementation precisely, sometimes referring to background design choices.

The target readers are those who already know Scheme and want to write useful programs in Gauche. For those who are new to Scheme, it'll be easier to start from some kind of tutorial. I'm planning to write one.

This manual only deals with Scheme side of things. Gauche has another face, a C interface. Details of it will be discussed in a separate document to be written. Those who want to use Gauche as an embedded language, or want to write an extension, need that volume.

For the Scheme side, I tried to make this manual self-contained for the reader's convenience, i.e. as far as you want to look up Gauche's features you don't need to refer to other documents. For example, description of functions defined in the standard documents are included in this manual, instead of saying "see the standard document". However, this document is not a verbatim copy of the standard documents; sometimes I omit detailed discussions for brevity. I put pointers to the original documents, so please consult them if you need to refer to the standards.

If you're reading this document off-line, you may find the most recent version on the web:

`https://practical-scheme.net/gauche/`.

1.1 Overview of Gauche

Gauche is a Scheme script engine; it reads Scheme programs, compiles it on-the-fly and executes it on a virtual machine. Gauche conforms the language standard "Revised^7 Report on the Algorithmic Language Scheme" ([R7RS], page 829), and supports various common libraries defined in SRFIs (`https://srfi.schemers.org`).

The goal of Gauche is to provide a handy tool for programmers and system administrators to handle daily works conveniently and efficiently in the production environment.

There are lots of Scheme implementations available, and each of them has its design emphasis and weaknesses. Gauche is designed with emphasis on the following criteria.

Quick startup
: One of the situation Gauche is aiming at is in the production environment, where you write ten-lines throw-away script that may invoked very frequently. This includes CGI scripts as well. Gauche provides frequently used common features as a part of rich built-in functions or precompiled Scheme libraries that can be loaded very quickly.

Fully utilizing multi-core
: Gauche supports native threads on most platforms. The internals are fully aware of preemptive/concurrent threads (that is, no "giant global lock"), so that you can utilize multiple cores on your machine.

Multibyte strings
: We can no longer live happily in ASCII-only or 1-byte-per-character world. The practical language implementations are required to handle multibyte (wide) characters. Gauche supports multibyte strings natively, providing robust and consistent support than *ad hoc* library-level implementation. See Section 2.2 [Multibyte strings], page 11, for details.

Integrated object system
: A powerful CLOS-like object system with MetaObject protocol (mostly compatible with STklos and Guile) is provided.

System interface

Although Scheme abstracts lots of details of the machine, sometimes you have to bypass these high-level layers and go down to the basement to make things work. Gauche has built-in support of most of POSIX.1 system calls. Other modules, such as networking module, usually provide both high-level abstract interface and low-level interface close to system calls.

Enhanced I/O

No real application can be written without dealing with I/O. Scheme neatly abstracts I/O as a port, but defines least operations on it. Gauche uses a port object as a unified abstraction, providing utility functions to operate on the underlying I/O system. See Section 6.22 [Input and output], page 217, for the basic I/O support.

Extended language

Gauche is not just an implementation of Scheme; it has some language-level enhancements. For example, *lazy sequences* allows you to have lazy data structures that behaves as if they're ordinary lists (except that they're realized lazily). It is different from library-level lazy structure implementation such as streams (srfi-41), in a sense that you can use any list-processing procedures on lazy sequences. It enables programs to use lazy algorithms more liberally.

1.2 Notations

1.2.1 Entry format

In this manual, each entry is represented like this:

`foo` *arg1 arg2* [Category]

[spec]{module} Description of foo . . .

Category denotes the category of the entry **foo**. The following categories will appear in this manual:

Function	A Scheme function.
Special Form	A special form (in the R7RS term, "syntax").
Macro	A macro.
Module	A module
Class	A class.
Generic Function	A generic function
Method	A method
Reader Syntax	A lexical syntax that is interpreted by the reader.
Parameter	A parameter, which is a procedure that follows a certain protocol and used to manipulate the dynamic environment. See Section 9.22 [Parameters], page 411, for the details.
Generic application	In Gauche, you can "apply" a non-procedure object to arguments as if it is a procedure (see Section 6.18.6 [Applicable objects], page 195, for the details). This entry explains the behavior of an object applied to arguments.
Subprocess argument	This appears in `do-process` and `run-process` to explain their keyword argument (see Section 9.25.1 [Running subprocess], page 421)
EC Qualifier	This is for SRFI-42 Eager Comprehension qualifiers. (see Section 11.12 [Eager comprehensions], page 610).

For functions, special forms and macros, the entry may be followed by one or more arguments. In the argument list, the following notations may appear:

`arg ...` Indicates zero or more arguments.

`:optional` *x y z*
`:optional (`*x x-default*`) (`*y y-default*`)` *z*
: Indicates it may take up to three optional arguments. The second form specifies default values to *x* and *y*. This is Gauche's enhancement to Scheme; see Section 4.3 [Making procedures], page 42, for the definition of complete argument list syntax.

`:key` *x y z*
`:key (`*x x-default*`) (`*y y-default*`)` *z*
: Indicates it may take keyword arguments *x*, *y* and *z*. The second form shows the default values for *x* and *y*. This is also Gauche's enhancement to Scheme; see Section 4.3 [Making procedures], page 42, for the definition of complete argument list syntax.

`:rest` *args*
: Indicates it may take rest arguments. This is also Gauche's enhancement to Scheme; see Section 4.3 [Making procedures], page 42, for the definition of complete argument list syntax.

Following the entry line, we may indicate the specification the entry comes from, and/or the module the entry is provided when it's not built-in.

The specification is shown in brackets. You'll see the following variations.

[R7RS], [R7RS *library*]
: It is defined in R7RS. If the entry is about a procedure, a syntax or a macro, *library* is also shown to indicate the name is exported from the `scheme.`*library* module (or the `(scheme` *library*`)` library, in R7RS terms).

[R7RS+], [R7RS+ *library*]
: It is defined in R7RS, but extended by Gauche, e.g. accepting more optional arguments or different type of arguments. The description contains how it is extended from R7RS. When you're writing a portable program, you need to be careful not to use Gauche-specific features.

[R6RS], [R6RS+], [R5RS], [R5RS+]
: It is defined in R6RS or R5RS. The plus sign means it has extended by Gauche. Since R7RS is mostly upward-compatible to R5RS, and has a lot in common with R6RS, we mark an entry as R5RS or R6RS only if it is not a part of R7RS.

[SRFI-*n*], [SRFI-*n*+]
: The entry works as specified in SRFI-*n*. If it is marked as "[SRFI-*n*+]", the entry has additional functionality.

[POSIX] The API of the entry reflects the API specified in POSIX.

The module is shown in curly-braces. If the module isn't shown, it is built-in for Gauche. (Note: When you're writing R7RS code, Gauche built-ins are available through `(gauche base)` module, see Section 9.2 [Importing gauche built-ins], page 322).

Some entries may be available from more than one modules through re-exporting or module inheritance. We only list the primary module in that case.

Here's an actual entry for an example:

```
-- Function: utf8->string u8vector :optional start end
    [R7RS base] {gauche.unicode} Converts a sequence of utf8 octets in
```

```
U8VECTOR to a string.  Optional START and/or END argument(s) will
limit the range of the input.
```

This shows the function `utf8->string` is specified by R7RS, in `(scheme base)` library. Gauche originally provides it from `gauche.unicode` module. You can import the function from either one, but in general, it's good to use `(import (scheme base))` when writing R7RS code, and `(use gauche.unicode)` when writing Gauche code. See Section 10.1 [R7RS integration], page 499, for the details of differences in writing in R7RS and Gauche.

1.2.2 Names and namespaces

Since R6RS, you can split toplevel definitions of Scheme programs into multiple namespaces. In the standards such namespaces are called *libraries*. Gauche predates R6RS and has been calling them *modules*, and we use the latter throughout this manual.

(Note: RnRS libraries are more abstract concept than Gauche's modules; RnRS defines libraries in a way that they can be implemented in various ways, and it just happens that Gauche realises the library semantics using modules. When you write a portable R7RS library, be aware not to rely on Gauche-specific module semantics. Especially, RnRS libraries are more static than Gauche modules; you cannot add definitions to exiting libraries within RnRS, for example.)

Sometimes the same name is used for multiple definitions in different modules. If we need to distinguish those names, we prefix the name with the module name and a hash sign. For example, `gauche#lambda` means `lambda` defined in `gauche` module. This does not mean you can write `gauche#lambda` in the source code, though: This notation is just for explanation.

2 Concepts

In this chapter I describe a few Gauche's design concepts that help you to understand how Gauche works.

2.1 Standard conformance

Gauche conforms “Revised^7 Report of Algorithmic Language Scheme,” (R7RS) including optional syntax and procedures. We cover R7RS small language (see Section 10.2 [R7RS small language], page 503), as well as part of R7RS large libraries (see Section 10.3 [R7RS large], page 512).

- Gauche has a special kind of symbols, called keywords. They’re symbols with its name beginning with a colon (e.g. `:key`), but behaves as if it is automatically bound to itself. See Section 6.8 [Keywords], page 139, for the details. Keywords are used extensively when passing so-called keyword arguments (see Section 4.3 [Making procedures], page 42).
- Continuations created in a certain situation (specifically, inside a Scheme code that is called from external C routine) have limited extent (See Section 6.18.7 [Continuations], page 195, for details).
- Full numeric tower (integer, rational, real and complex numbers) are supported, but rationals are only exact, and complex numbers are always inexact.

Note that, since Gauche predates R7RS, most existing Gauche source code doesn’t follow the R7RS program/library structure. Gauche can read both traditional Gauche modules/scripts and R7RS programs/libraries seamlessly. See Chapter 10 [Library modules - R7RS standard libraries], page 499, for the details of how R7RS is integrated into Gauche.

Gauche also supports the following SRFIs (Scheme Request for Implementation).

SRFI-0, Feature-based conditional expansion construct.
: This has become a part of R7RS small. Gauche supports this as Built-in. See Section 4.12 [Feature conditional], page 68.

SRFI-1, List library (R7RS lists)
: This has become a part of R7RS large. See Section 10.3.1 [R7RS lists], page 512. (Some of SRFI-1 procedures are built-in).

SRFI-2, AND-LET*: an AND with local bindings, a guarded LET* special form.
: Supported natively. See Section 4.6 [Binding constructs], page 52.

SRFI-4, Homogeneous numeric vector datatypes.
: The module `gauche.uvector` provides a superset of `srfi-4` procedures, including arithmetic operations and generic interface on the SRFI-4 vectors. See Section 9.36 [Uniform vectors], page 476.

SRFI-5, A compatible let form with signatures and rest arguments
: Supported by the module `srfi-5`. See Section 11.3 [A compatible let form with signatures and rest arguments], page 590.

SRFI-6, Basic String Ports.
: This has become a part of R7RS small. Gauche supports this as built-in. See Section 6.22.5 [String ports], page 224.

SRFI-7, Feature-based program configuration language
: Supported as an autoloaded macro. See Section 11.4 [Feature-based program configuration language], page 590.

SRFI-8, receive: Binding to multiple values.
: Syntax `receive` is built-in. See Section 4.6 [Binding constructs], page 52.

SRFI-9, Defining record types.
: Supported by the module `gauche.record`. See Section 9.26 [Record types], page 433.

SRFI-10, Sharp-comma external form.
: Built-in. See Section 6.22.7.3 [Read-time constructor], page 229.

SRFI-11, Syntax for receiving multiple values.
: This has become a part of R7RS small. Gauche supports it as built-in. See Section 4.6 [Binding constructs], page 52.

SRFI-13, String library
: Supported by the module `srfi-13`. See Section 11.5 [String library], page 591. (Some of SRFI-13 procedures are built-in).

SRFI-14, Character-set library
: This has become a part of R7RS large. Character-set object and a few procedures are built-in. See Section 6.11 [Character set], page 147. Complete set of SRFI-14 is supported by the module `scheme.charset`. See Section 10.3.6 [R7RS character sets], page 533.

SRFI-16, Syntax for procedures of variable arity (case-lambda)
: This has become a part of R7RS small. Built-in. See Section 4.3 [Making procedures], page 42.

SRFI-17, Generalized set!
: Built-in. See Section 4.4 [Assignments], page 47.

SRFI-18, Multithreading support
: Some SRFI-18 features are built-in, and the rest is in `gauche.threads` module. See Section 9.33 [Threads], page 457.

SRFI-19, Time Data Types and Procedures.
: Time data type is Gauche built-in (see Section 6.25.9 [Time], page 269). Complete set of SRFI-19 is supported by the module `srfi-19`. See Section 11.7 [Time data types and procedures], page 600.

SRFI-22, Running Scheme scripts on Unix
: Supported. See Section 3.3 [Writing Scheme scripts], page 25.

SRFI-23, Error reporting mechanism.
: This has become a part of R7RS small. Built-in. See Section 6.20.2 [Signaling exceptions], page 207.

SRFI-25, Multi-dimensional array primitives.
: Supported by the module `gauche.array`, which defines superset of SRFI-25. See Section 9.1 [Arrays], page 315.

SRFI-26, Notation for specializing parameters without currying.
: Built-in. See Section 4.3 [Making procedures], page 42.

SRFI-27, Sources of Random Bits.
: Supported by the module `srfi-27`. See Section 11.8 [Sources of random bits], page 605.

SRFI-28, Basic format strings.
: Gauche's built-in `format` procedure is a superset of SRFI-28 `format`. See Section 6.22.8 [Output], page 231.

SRFI-29, Localization
: Supported by the module `srfi-29`. See Section 11.9 [Localization], page 606.

SRFI-30, Nested multi-line comments.
: This has become a part of R7RS small. Supported by the native reader. See Section 4.1 [Lexical structure], page 38.

SRFI-31, A special form rec for recursive evaluation. Built-in.
: See Section 4.6 [Binding constructs], page 52.

SRFI-34, Exception Handling for Programs
: This has become a part of R7RS small. Built-in. See Section 6.20 [Exceptions], page 204. (However, Gauche implements srfi-18's semantics of `raise` literally, which differs slightly from srfi-34's. This may be changed in future.)

SRFI-35, Conditions
: Built-in. See Section 6.20.4 [Conditions], page 212.

SRFI-36, I/O Conditions
: Partly supported. See Section 6.20.4 [Conditions], page 212.

SRFI-37, args-fold: a program argument processor
: Supported by the module `srfi-37`. See Section 11.10 [A program argument processor], page 607.

SRFI-38, External Representation for Data With Shared Structure
: Built-in. See Section 6.22.7.1 [Reading data], page 227, and Section 6.22.8 [Output], page 231.

SRFI-39, Parameter objects
: This has become a part of R7RS small. Supported by the module `gauche.parameter`. See Section 9.22 [Parameters], page 411.

SRFI-40, A Library of Streams
: Supported by the module `util.stream`. See Section 12.71 [Stream library], page 809.

SRFI-41, Streams
: This has become a part of R7RS large. See Section 10.3.11 [R7RS stream], page 546. Most of stream procedures are also in `util.stream` (see Section 12.71 [Stream library], page 809).

SRFI-42, Eager comprehensions
: Supported by the module `srfi-42`. See Section 11.12 [Eager comprehensions], page 610.

SRFI-43, Vector library
: Supported by the module `srfi-43`. See Section 11.13 [Vector library (Legacy)], page 615. Note that this srfi is superseded by R7RS `scheme.vector` library (formerly known as `srfi-133`). See Section 10.3.2 [R7RS vectors], page 517.

SRFI-45, Primitives for Expressing Iterative Lazy Algorithms
: Built-in. See Section 6.19 [Lazy evaluation], page 199.

SRFI-46, Basic Syntax-rules Extensions
: This has become a part of R7RS small. Built-in. See Section 5.2 [Hygienic macros], page 83.

SRFI-55, require-extension
: Supported as an autoloaded macro. See Section 11.14 [Requiring extensions], page 616.

SRFI-60, Integers as bits
: Most procedures are built-in: See Section 11.34 [Bitwise operations], page 633. The complete support is in `srfi-60` module: See Section 11.15 [Integers as bits], page 616.

SRFI-61, A more general `cond` clause
: Supported natively. See Section 4.5 [Conditionals], page 49.

SRFI-62, S-expression comments
: This has become a part of R7RS small. Supported by the native reader. See Section 4.1 [Lexical structure], page 38.

SRFI-64, A Scheme API for test suites
: Supported by the module `srfi-64`.

SRFI-66, Octet vectors
: Supported by the module `srfi-66` (see Section 11.16 [Octet vectors], page 618). This is mostly a subset of `gauche.uvector`, but has one slight difference.

SRFI-69, Basic hash tables
: Supported by the module `srfi-69` (see Section 11.17 [Basic hash tables], page 618). Note that this srfi is superseded by R7RS `scheme.hash-table` library (formerly known as `srfi-125`). See Section 10.3.7 [R7RS hash tables], page 536.

SRFI-74, Octet-addressed binary blocks
: Supported by the module `srfi-74` (see Section 11.18 [Octet-addressed binary blocks], page 620).

SRFI-78, Lightweight testing
: Supported by the module `srfi-78`. Since Gauche already has its own test framework (see Section 9.32 [Unit testing], page 452), this is mainly for third-party modules that adopt srfi-78 for testing.

SRFI-87, `=>` in case clauses
: This has become a part of R7RS small. Supported natively. See Section 4.5 [Conditionals], page 49.

SRFI-95, Sorting and merging
: Supported natively. See Section 6.24 [Sorting and merging], page 244.

SRFI-96, SLIB Prerequisites
: This srfi is not exactly a library, but rather a description of features the platform should provide to support SLIB. In order to load this module, SLIB must be already installed. See Section 12.45 [SLIB], page 746, for the details.

SRFI-98, An interface to access environment variables
: Supported by the module `srfi-98`. See Section 11.19 [Accessing environment variables], page 622.

SRFI-99, ERR5RS Records
: Supported by the module `gauche.record`. See Section 9.26 [Record types], page 433.

SRFI-106, Basic socket interface
: Supported by the module `srfi-106`. See Section 11.20 [Basic socket interface], page 623.

SRFI-111, Boxes
: This has become a part of R7RS large. Supported by the module `scheme.box`. See Section 10.3.12 [R7RS boxes], page 547.

SRFI-112, Environment inquiry
: Supported by the module `srfi-112`. See Section 11.22 [Portable runtime environment inquiry], page 625.

SRFI-113, Sets and Bags
: This has become a part of R7RS large. Supported by the module `scheme.set`. See Section 10.3.5 [R7RS sets], page 525.

SRFI-114, Comparators
: Some of the features are built-in (see Section 6.2.4 [Basic comparators], page 103). Full srfi spec is supported by the module `srfi-114` (see Section 11.24 [Comparators], page 626).

SRFI-115, Scheme Regular Expressions
: This has become a part of R7RS large. Supported by the module `scheme.regex`. See Section 10.3.16 [R7RS regular expressions], page 552.

SRFI-117, Queues based on lists.
: This has become a part of R7RS large. Supported by the module `scheme.list-queue`, which is implemented on top of `data.queue`. (see Section 10.3.13 [R7RS list queues], page 547)

SRFI-118, Simple adjustable-size strings
: Supported by the module `srfi-118`. (see Section 11.27 [Simple adjustable-size strings], page 631)

SRFI-121, Generators
: This has become a part of R7RS large. Gauche's `gauche.generator` is superset of srfi-121 (see Section 9.11 [Generators], page 372).

SRFI-124, Ephemerons
: This has become a part of R7RS large. Supported by `scheme.ephemeron`. Note: Current Gauche's implementation isn't optimal. See Section 10.3.14 [R7RS ephemerons], page 550.

SRFI-125, Intermediate hash tables
: This has become a part of R7RS large. Supported by `scheme.hash-table` (see Section 10.3.7 [R7RS hash tables], page 536). Note that Gauche's native interface provides the same functionalities, but under slightly different names for the backward compatibility. See Section 6.15 [Hashtables], page 177.

SRFI-127, Lazy sequences
: This has become a part of R7RS large. Supported by `scheme.lseq` (see Section 10.3.10 [R7RS lazy sequences], page 544).

SRFI-128, Comparators (reduced)
: This has become a part of R7RS large. Built-in. See Section 6.2.4 [Basic comparators], page 103, for the details.

SRFI-129, Titlecase procedures
: The procedures `char-title-case?` and `char-titlecase` are built-in, and `string-titlecase` is in `gauche.unicode`. For the compatibility, you can `(use srfi-129)` or `(import (srfi 129))` to get these three procedures.

SRFI-131, ERR5RS Record Syntax (reduced)
: This srfi is a pure subset of srfi-99, and `gauche.record`'s `define-record-type` covers it. See Section 9.26 [Record types], page 433.

SRFI-132, Sort libraries
: This has become a part of R7RS large. Supported by the module `scheme.sort`. See Section 10.3.4 [R7RS sort], page 522.

SRFI-133, Vector library (R7RS-compatible)
: This has become a part of R7RS large. Supported by the module `scheme.vector`. See Section 10.3.2 [R7RS vectors], page 517.

SRFI-134, Immutable Deques
: This has become a part of R7RS large. The module `data.ideque` is compatible to srfi-134. See Section 12.10 [Immutable deques], page 654.

SRFI-141, Integer division
: This has become a part of R7RS large. Supported by the module `scheme.division`. See Section 10.3.18 [R7RS integer division], page 574.

SRFI-143, Finxums
: This has become a part of R7RS large. Supported by the module `scheme.fixnum`. See Section 10.3.20 [R7RS fixnum], page 579.

SRFI-144, Flonums
: This has become a part of R7RS large. Supported by the module `scheme.flonum`. See Section 10.3.21 [R7RS flonum], page 581.

SRFI-145, Assumptions
: Built-in. See Section 4.5 [Conditionals], page 49.

SRFI-146, Mappings
: This has become a part of R7RS large. Supported by the module `scheme.mapping`. See Section 10.3.17 [R7RS mappings], page 563.

SRFI-149, Basic syntax-rules template extensions
: The built-in `syntax-rules` support `srfi-149`.

SRFI-151, Bitwise operations
: Supported by the module `srfi-151` (see Section 11.34 [Bitwise operations], page 633). Note that many equivalent procedures are provided built-in (see Section 6.3.6 [Basic bitwise operations], page 121).

SRFI-152, String library (reduced)
: Supported by the module `srfi-152` (see Section 11.35 [String library (reduced)], page 633).

SRFI-154, First-class dynamic extents
: Suppored by the module `srfi-154`. (see Section 11.36 [First-class dynamic extents], page 635).

SRFI-158, Generators and accumulators
: This has become a part of R7RS large. Supported by the module `scheme.generator` (see Section 10.3.9 [R7RS generators], page 542). Note that most of generator procedures are supported by `gauche.generator` (see Section 9.11 [Generators], page 372).

SRFI-160, Homogeneous numeric vector libraries
: This has become a part of R7RS large, supported by the module `scheme.vector.@` where `@` is one of `base`, `u8`, `s8`, `u16`, `s16`, `u32`, `s32`, `u64`, `s64`, `f32`, `f64`, `c64`, or `c128` (see Section 10.3.3 [R7RS uniform vectors], page 522).

SRFI-162, Comparators sublibrary
: Supported by the module `srfi-162`. See Section 11.39 [Comparator sublibrary], page 636.

SRFI-173, Hooks
: Supported by the module `srfi-173` (see Section 11.40 [Hooks (srfi)], page 637), which is a thin layer on top of `gauche.hook` (see Section 9.12 [Hooks], page 383)

2.2 Multibyte strings

Traditionally, a string is considered as a simple array of bytes. Programmers tend to imagine a string as a simple array of characters (though a character may occupy more than one byte). It's not the case in Gauche.

Gauche supports *multibyte string* natively, which means characters are represented by variable number of bytes in a string. Gauche retains semantic compatibility of Scheme string, so such details can be hidden, but it'll be helpful if you know a few points.

A string object keeps a type tag and a pointer to the storage of the string body. The storage of the body is managed in a sort of "copy-on-write" way—if you take substring, e.g. using directly by `substring` or using regular expression matcher, or even if you copy a string by `copy-string`, the underlying storage is shared (the "anchor" of the string is different, so the copied string is not `eq?` to the original string). The actual string is copied only if you destructively modify it.

Consequently the algorithm like pre-allocating a string by `make-string` and filling it with `string-set!` becomes *extremely* inefficient in Gauche. Don't do it. (It doesn't work with mulitbyte strings anyway). Sequential access of string is much more efficient using *string ports* (see Section 6.22.5 [String ports], page 224).

String search primitives such as `string-scan` (see Section 6.12.7 [String utilities], page 158) and regular expression matcher (see Section 6.13 [Regular expressions], page 162) can return a matched string directly, without using index access at all.

You can choose *internal* encoding scheme at the time of compiling Gauche. At runtime, a procedure `gauche-character-encoding` can be used to query the internal encoding. At compile time, you can use a feature identifier to check the internal encoding. (see Section 3.5 [Platform-dependent features], page 29.) Currently, the following internal encodings are supported.

`utf-8`
: UTF-8 encoding of Unicode. This is the default. The feature identifier `gauche.ces.utf8` indicates Gauche is compiled with this internal encoding.

`euc-jp`
: EUC-JP encoding of ASCII, JIS X 0201 kana, JIS X 0212 and JIS X 0213:2000 Japanese character set. The feature identifier `gauche.ces.eucjp` indicates Gauche is compiled with this internal encoding.

`sjis`
: Shift-JIS encoding of JIS X 0201 kana and JIS X 0213:2000 Japanese character set. For source-code compatibility, the character code between 0 and 0x7f is mapped to ASCII. The feature identifier `gauche.ces.sjis` indicates Gauche is compiled with this internal encoding.

`none`
: 8-bit fixed-length character encoding, with the code between 0 and 0x7f matches ASCII. It's up to the application to interpret the string with certain character encodings. The feature identifier `gauche.ces.none` indicates Gauche is compiled with this internal encoding.

Conversions from other encoding scheme is provided as a special port. See Section 9.4 [Character code conversion], page 339, for details.

The way to specify the encoding of source programs will be explained in the next section.

2.3 Multibyte scripts

You can use characters other than `us-ascii` not only in literal strings and characters, but in comments, symbol names, literal regular expressions, and so on.

By default, Gauche assumes a Scheme program is written in its internal character encoding. It is fine as far as you're writing scripts to use your own environment, but it becomes a problem if somebody else tries to use your script and finds out you're using different character encoding than his/hers.

So, if Gauche finds a comment something like the following within the first two lines of the program source, it assumes the rest of the source code is written in <encoding-name>, and does the appropriate character encoding conversion to read the source code:

```
;; coding: <encoding-name>
```

More precisely, a comment in either first or second line that matches a regular expression `#/coding[:=]\s*([\w.-]+)/` is recognized, and the first submatch is taken as an encoding name. If there are multiple matches, only the first one is effective. The first two lines must not contain characters other than us-ascii in order for this mechanism to work.

The following example tells Gauche that the script is written in EUC-JP encoding. Note that the string `"-*-"` around the coding would be recognized by Emacs to select the buffer's encoding appropriately.

```
#!/usr/bin/gosh
;; -*- coding: euc-jp -*-

... script written in euc-jp ...
```

Internally, the handling of this *magic comment* is done by a special type of port. See Section 6.22.6 [Coding-aware ports], page 226, for the details. See also Section 6.23.1 [Loading Scheme file], page 239, for how to disable this feature.

2.4 Case-sensitivity

Historically, most Lisp-family languages are case-insensitive for symbols. Scheme departed from this tradition since R6RS, and the symbols are read in case-sensitive way. (Note that symbols have been case-sensitive internally even in R5RS Scheme; case-insensitivity is about readers.)

Gauche reads and writes symbols in case-sensitive manner by default, too. However, to support legacy code, you can set the reader to case-insensitive mode, in the following ways:

Use `#!fold-case` reader directive

> When Gauche sees a token `#!fold-case` during reading a program, the reader switches to case-insensitive mode. A token `#!no-fold-case` has an opposite effect—to make the reader case-sensitive. These tokens affect the port from which they are read, and are in effect until EOF or another instance of these tokens are read. See Section 4.1 [Lexical structure], page 38, for more details on `#!` syntax. This is the way defined in R6RS and R7RS.

Use `-fcase-fold` command-line argument

> Alternatively, you can give a command-line argument `-fcase-fold` to the `gosh` command (see Section 3.1 [Invoking Gosh], page 16). In this mode, the reader folds uppercase characters in symbols to lowercase ones. If a symbol name contains uppercase characters, it is written out using `|`-escape (see Section 6.7 [Symbols], page 137).

2.5 Integrated object system

Gauche has a STklos-style object system, similar to CLOS. If you have used some kind of object oriented (OO) languages, you'll find it easy to understand the basic usage:

```
;; Defines a class point, that has x and y coordinate
(define-class point ()
```

```
    ((x :init-value 0)
     (y :init-value 0))
    )

  (define-method move ((p point) dx dy)
    (inc! (slot-ref p 'x) dx)
    (inc! (slot-ref p 'y) dy))

  (define-method write-object ((p point) port)
    (format port "[point ~a ~a]"
            (slot-ref p 'x)
            (slot-ref p 'y)))
```

However, if you are familiar with mainstream OO languages but new to CLOS-style object system, Gauche's object system may look strange when you look deeper into it. Here I describe several characteristics of Gauche object system quickly. See Chapter 7 [Object system], page 279, for details.

Everything is an object (if you care)

You have seen this tagline for the other languages. And yes, in Gauche, everything is an object in the sense that you can query its class, and get various meta information of the object at run time. You can also define a new method on any class, including built-in ones.

Note that, however, in CLOS-like paradigm it doesn't really matter whether everything is an object or not, because of the following characteristics:

Method is dispatched by all of its arguments.

Unlike other object-oriented languages such as C++, Objective-C, Python, Ruby, etc., in which a method always belong to a single class, a Gauche method doesn't belong to a specific class.

For example, suppose you define a numeric vector class `<num-vector>` and a numeric matrix class `<num-matrix>`. You can define a method `product` with all possible combinations of those type of arguments:

```
(product <num-vector> <num-matrix>)
(product <num-matrix> <num-vector>)
(product <num-vector> <num-vector>)
(product <num-matrix> <num-matrix>)
(product <number>     <num-vector>)
(product <number>     <num-matrix>)
(product <number>     <number>)
```

Each method belongs to neither `<num-vector>` class nor `<num-matrix>` class.

Since a method is not owned by a class, you can always define your own method on the existing class (except a few cases that the system prohibits altering pre-defined methods). The above example already shows it; you can make `product` method work on the built-in class `<number>`. That is why I said it doesn't make much sense to discuss whether everything is object or not in CLOS-style object system.

To step into the details a bit, the methods are belong to a *generic function*, which is responsible for dispatching appropriate methods.

Class is also an instance.

By default, a class is also an instance of class `<class>`, and a generic function is an instance of class `<generic>`. You can subclass `<class>` to customize how a class is initialized or how its slots are accessed. You can subclass `<generic>` to customize how the applicable methods are selected, which order those methods are called, etc. The mechanism is called *metaobject protocol*. Metaobject protocol allows you to extend the language by the language itself.

To find examples, see the files `lib/gauche/singleton.scm` and `lib/gauche/mop/validator.scm` included in the distribution. You can also read `lib/gauche/mop/object.scm`, which actually defines how a class is defined in Gauche. For more details about metaobject protocol, see [MOP], page 829.

Class doesn't create namespace
In the mainstream OO language, a class often creates its own namespace. This isn't the case in CLOS-style object system. In Gauche, a namespace is managed by the module system which is orthogonal to the object system.

2.6 Module system

Gauche has a simple module system that allows modularized development of large software.

A higher level interface is simple enough from the user's point of view. It works like this: When you want to use the features provided by module `foo`, you just need to say `(use foo)` in your code. This form is a macro and interpreted at compile time. Usually it loads the files that defines `foo`'s features, and imports the external APIs into the calling module.

The `use` mechanism is built on top of two independent lower mechanisms, namespace separation and file loading mechanism. Those two lower mechanisms can be used separately, although it is much more convenient when used together.

The `use` mechanism is not transitive; that is, if a module B uses a module A, and a module C uses the module B, C doesn't see the bindings in A. It is because B and A is not in the `is-a` relationship. Suppose the module A implements a low-level functionality and the module B implements a high-level abstraction; if C is using B, what C wants to see is just a high-level abstraction, and doesn't concern how B implements such functionality. If C wants to access low-level stuff, C has to `use` A explicitly.

There is another type of relationship, though. You might want to take an exiting module A, and add some interface to it and provide the resulting module B as an extension of A. In such a case, B is-a A, and it'd be natural that the module that uses B can also see A's bindings. In Gauche, it is called *module inheritance* and realized by `extend` form.

The following sections in this manual describes modules in details.

- Section 3.7 [Writing Gauche modules], page 32, explains the convention of writing modules.
- Section 4.13 [Modules], page 71, describes special forms and macros to define and to use modules, along the built-in functions to introspect module internals.

2.7 Compilation

By default, Gauche reads toplevel Scheme forms one at a time, compile it immediately to intermediate form and execute it on the VM. As long as you use Gauche interactively, it looks like an interpreter. (There's an experimental ahead-of-time compiler as well. See `HOWTO-precompile.txt` if you want to give a try.)

The fact that we have separate compilation/execution phase, even interleaved, may lead a subtle surprise if you think Gauche as an interpreter. Here's a few points to keep in mind:

load is done at run time.
`load` is a procedure in Gauche, therefore evaluated at run time. If the loaded program defines a macro, which is available for the compiler after the toplevel form

containing `load` is evaluated. So, suppose `foo.scm` defines a macro `foo`, and you use the macro like this:

```
;; in "foo.scm"
(define-syntax foo
  (syntax-rules () ((_ arg) (quote arg))))

;; in your program
(begin (load "foo") (foo (1 2 3)))
  ⇒ error, bad procedure: '1'

(load "foo")
(foo (1 2 3)) ⇒ '(1 2 3)
```

The `(begin (load ...))` form fails, because the compiler doesn't know `foo` is a special form at the compilation time and compiles `(1 2 3)` as if it is a normal procedure call. The latter example works, however, since the execution of the toplevel form `(load "foo")` is done before `(foo (1 2 3))` is compiled.

To avoid this kind of subtleties, use `require` or `use` to load a program fragments. Those are recognized by the compiler.

require is done at compile time

On the other hand, since `require` and `use` is recognized by the compiler, the specified file is loaded even if the form is in the conditional expression. If you really need to load a file on certain condition, use `load` or do dispatch in macro (e.g. `cond-expand` form (see Section 4.12 [Feature conditional], page 68).)

3 Programming in Gauche

3.1 Invoking Gosh

Gauche can be used either as an independent Scheme scripting engine or as an embedded Scheme library. An interactive interpreter which comes with Gauche distribution is a program named `gosh`.

`gosh` [*options*] [*scheme-file arg* . . .] [Program]

Gauche's interpreter. Without *scheme-file*, `gosh` works interactively, i.e. it reads a Scheme expression from the standard input, evaluates it, and prints the result, and repeat that until it reads EOF or is terminated.

If `gosh` is invoked without *scheme-file*, but the input is not a terminal, it enters read-eval-print loop but not writes out a prompt while waiting input form. This is useful when you pipe Scheme program into `gosh`. You can force this behavior or suppress this behavior by `-b` and `-i` options.

If *scheme-file* is specified, `gosh` runs it as a Scheme program and exit. See Section 3.3 [Writing Scheme scripts], page 25, for details.

Command-line options

The following command line options are recognized by `gosh`. The first command line argument which doesn't begin with '−' is recognized as the script file. If you want to specify a file that begins with a minus sign, use a dummy option '`--`'.

`-I` *path* [Command Option]

Prepends *path* to the load path list. You can specify this option more than once to add multiple paths.

`-A` *path* [Command Option]

Appends *path* to the tail of the load path list. You can specify this option more than once to add multiple paths.

`-q` [Command Option]

Makes `gosh` not to load the default initialization file.

`-V` [Command Option]

Prints the `gosh` version and exits.

`-v` *version* [Command Option]

If *version* is not the running `gosh`'s version, execute the specified version of `gosh` instead if it is installed. This is useful when you want to invoke specific version of Gauche. Note that *version* must be 0.9.6 or later.

`-u` *module* [Command Option]

Use *module*. Before starting execution of *scheme-file* or entering the read-eval-print loop, the specified module is *use*d, i.e. it is loaded and imported (See Section 4.13.3 [Defining and selecting modules], page 73, for details of `use`). You can specify this option more than once to use multiple modules.

`-l` *file* [Command Option]

Load *file* before starting execution of *scheme-file* or entering the read-eval-print loop. The file is loaded in the same way as `load` (see Section 6.23.1 [Loading Scheme file], page 239). You can specify this option more than once to load multiple files.

`-L` *file* [Command Option]

Load *file* like `-l`, but if *file* does not exist, this silently ignores it instead of reporting an error. This option can also be specified multiple times.

`-e` *scheme-expression* [Command Option]

Evaluate *scheme-expression* before starting execution of *scheme-file* or entering the read-eval-print loop. Evaluation is done in the *interaction-environment* (see Section 6.21 [Eval and repl], page 216). You can specify this option more than once to evaluate multiple expressions.

`-E` *scheme-expression* [Command Option]

Same as -e, except the *scheme-expression* is read as if it is surrounded by parenthesis. For example:

```
% gosh -umath.const -E"print (sin (* pi/180 15))" -Eexit
0.25881904510252074
```

`-b` [Command Option]

Batch. Does not print prompts even if the input is a terminal.

`-i` [Command Option]

Interactive. Print prompts even if the input is not a terminal.

`-m` *module* [Command Option]

When a script file is given, this option makes the module named *module* in which the `main` procedure is looked for, instead of the `user` module. See Section 3.3 [Writing Scheme scripts], page 25, for the details of executing scripts.

If the named module doesn't exist after loading the script, an error is signaled.

This is useful to write a Scheme module that can also be executed as a script.

`-f` *compiler-option* [Command Option]

This option controls compiler and runtime behavior. For now we have following options available:

no-inline
: Prohibits the compiler from inlining procedures and constants. Equivalent to no-inline-globals, no-inline-locals, no-inline-constants and no-inline-setters combined.

no-inline-globals
: Prohibits the compiler from inlining global procedures.

no-inline-locals
: Prohibits the compiler from inlining local procedures.

no-inline-constants
: Prohibits the compiler from inlining constants.

no-inline-setters
: Prohibits the compiler from inlining setters.

no-post-inline-pass
: Prohibits the compiler from running post-inline optimization pass.

no-lambda-lifting-pass
: Prohibits the compiler from running lambda-lifting pass.

load-verbose
: Reports whenever a file is loaded. Useful to check precisely which files are loaded in what order.

include-verbose
: Reports whenever a file is included. Useful to check precisely which files are included in what order.

warn-legacy-syntax
: Warns if the reader sees legacy hex-escape syntax in string literals. See Section 6.22.7.2 [Reader lexical mode], page 228.

no-source-info
: Don't keep source information for debugging. Consumes less memory.

case-fold
: Ignore case for symbols. See Section 2.4 [Case-sensitivity], page 12.

test
: Adds `"../src"` and `"../lib"` to the load path before loading initialization file. This is useful when you want to test the compiled `gosh` interpreter inside source tree, without installing it.

`-p` *profiler-option* [Command Option]

Turn on the profiler. The following *profiler-option* is recognized:

`time`
: Records and reports time spent on function calls and number of times each function is called.

`load`
: Records and reports time spent on loading each modules. Useful to tune start-up time of the scripts. (Results are in elapsed time).

See Section 3.6.1 [Using profiler], page 30, for the details of the profiler.

`-r` *standard-revision* [Command Option]

Start `gosh` with an environment of the specified revision of Scheme standard. Currently only 7 is supported as *standar-revision*.

By default, `gosh` starts with `user` module, which inherits `gauche` module. That means you can use whole Gauche core procedures by default without explicitly declaring it.

Proper R7RS code always begins with either `define-library` or R7RS-style `import` form, and Gauche recognizes it and automatically switch to R7RS environments so that R7RS scripts and libraries can be executed by Gauche without special options. However, users who are learning R7RS Scheme may be confused when the initial environment doesn't look like R7RS.

By giving `-r7` option, `gosh` starts with `r7rs.user` module that extends the `r7rs` module, which defines two R7RS forms, `import` and `define-library`.

If you invoke `gosh` into an interactive REPL mode with `-r7` option, all standard R7RS-small libraries (except `(scheme r5rs)` are already imported for your convenience.

See Chapter 10 [Library modules - R7RS standard libraries], page 499, for the details on how Gauche supports R7RS.

(Note: The `-r7` option doesn't change reader lexical mode (see Section 6.22.7.2 [Reader lexical mode], page 228) to `strict-r7`. That's because using `strict-r7` mode by default prevents many Gauche code from being loaded.)

`--` [Command Option]

When `gosh` sees this option, it stops processing the options and takes next command line argument as a script file. It is useful in case if you have a script file that begins with a minus sign, although it is not generally recommended.

The options -I, -A, -l, -u, -e and -E are processes in the order of appearance. For example, adding a load path by -I affects the -l and -u option after it but not before it.

Environment variables

The following environment variables are recognized:

GAUCHE_AVAILABLE_PROCESSORS [Environment variable]

You can get the number of system's processors by sys-available-processors (see Section 6.25.3 [Environment inquiry], page 248); libraries/programs may use this info to optimize number of parallel threads. But you might change that, for testing and benchmarking—e.g. a program automatically uses 8 threads if there are 8 cores, but you might want to run it with 1, 2, 4 threads as well to see the effect of parallelization. This environment variable overrides the return value of sys-available-processors.

GAUCHE_DYNLOAD_PATH [Environment variable]

You can specify additional load paths for dynamically loaded objects by this environment variable, delimiting the paths by ':'. The paths are appended before the system default load paths.

See Section 6.23.2 [Loading dynamic library], page 241, for the details of how Gauche finds dynamically loadable objects.

GAUCHE_EDITOR [Environment variable]
EDITOR [Environment variable]

This is used by ed procedure in gauche.interactive module. See Section 9.13 [Interactive session], page 384, for the details.

GAUCHE_KEYWORD_DISJOINT [Environment variable]
GAUCHE_KEYWORD_IS_SYMBOL [Environment variable]

These two environment variables affect whether keywords are treated as symbols or not. See Section 6.8 [Keywords], page 139, for the details.

GAUCHE_LOAD_PATH [Environment variable]

You can specify additional load paths by this environment variable, delimiting the paths by ':'. The paths are appended before the system default load paths.

See Section 6.23.1 [Loading Scheme file], page 239, for the details of how Gauche finds files to load.

GAUCHE_QUASIRENAME_MODE [Environment variable]

This affects quasirename behavior, to keep the backward compatibility with 0.9.7 and before. See Section 5.2.2 [Explicit-renaming macro transformer], page 86, for the details.

GAUCHE_READ_EDIT [Environment variable]

This is used by gauche.interactive module to enable line-editor on REPL prompt. See Section 3.2 [Interactive development], page 20, for the details.

GAUCHE_REPL_NO_PPRINT [Environment variable]

This is used by gauche.interactive module to suppress pretty-printing on REPL prompt. See Section 3.2 [Interactive development], page 20, for the details.

GAUCHE_SUPPRESS_WARNING [Environment variable]

Suppress system warnings (WARNING: ...). Not generally recommended; use only if you absolutely need to.

GAUCHE_TEST_RECORD_FILE [Environment variable]

This is used by gauche.test module (see Section 9.32 [Unit testing], page 452). If defined, names a file the test processes keep the total statistics.

GAUCHE_TEST_REPORT_ERROR [Environment variable]

This is used by gauche.test module (see Section 9.32 [Unit testing], page 452). If defined, reports stack trace to stderr when the test thunk raises an error (even when it is expected). Useful for diagnosis of unexpected errors.

TMP [Environment variable]
TMPDIR [Environment variable]
TEMP [Environment variable]
USERPROFILE [Environment variable]

These may affect the return value of sys-tmpdir. Different environment variables may be used on different platforms. See Section 6.25.4.3 [Pathnames], page 253, for the details.

Windows-specific executable

On Windows-native platforms (mingw), two interpreter executables are installed. gosh.exe is compiled as a Windows console application and works just like ordinary gosh; that is, it primarily uses standard i/o for communication. Another executable, gosh-noconsole.exe, is compiled as a Windows no-console (GUI) application. It is not attached to a console when it is started. Its standard input is connected to the NUL device. Its standard output and standard error output are special ports which open a new console when something is written to them for the first time. (NB: This magic only works for output via Scheme ports; direct output from low-level C libraries will be discarded.)

The main purpose of gosh-noconsole.exe is for Windows scripting. If a Scheme script were associated to gosh.exe and invoked from Explorer, it would always open a new console window, which is extremely annoying. If you associate Scheme scripts to gosh-noconsole.exe instead, you can avoid console from popping up.

If you're using the official Windows installer, Scheme scripts (*.scm) have already associated to gosh-noconsole.exe and you can invoke them by double-clicking on Explorer. Check out some examples under C:\Program Files\Gauche\examples.

3.2 Interactive development

When gosh is invoked without any script files, it goes into interactive read-eval-print loop (REPL).

To exit the interpreter, type EOF (usually Control-D in Unix terminals) or evaluate (exit).

In the interactive session, gosh loads and imports gauche.interactive module (see Section 9.13 [Interactive session], page 384) into user module, for the convenience. Also, if there's a file .gaucherc under the user's home directory. You may put settings there that would help interactive debugging. (As of Gauche release 0.7.3, .gaucherc is no longer loaded when gosh is run in script mode.)

Note that .gaucherc is always loaded in the user module, even if gosh is invoked with -r7 option. The file itself is a Gauche-specific feature, so you don't need to consider portability in it.

I recommend you to run gosh inside Emacs, for it has rich features useful to interact with internal Scheme process. Put the following line to your .emacs file:

```
(setq scheme-program-name "gosh -i")
```

And you can run gosh by M-x run-scheme.

If you set an environment variable GAUCHE_READ_EDIT and the terminal is capable of cursor control, gosh starts with an experimental line-editor. After typing input, you can move back and change text, or invoke older input to edit, with Emacs-like key bindings. This feature is experimental—only very rudimental editor operations are supported, and you may experience weird behaviors. Let us know if you encounter buggy behaviors.

When `gosh` is invoked with read-edit mode, you see the prompt `gosh$`, instead of the default `gosh>`.

If you want to use multibyte characters in the interaction, make sure your terminal's settings is in sync with `gosh`'s internal character encodings.

3.2.1 Working in REPL

When you enter REPL, Gauche prompts you to enter a Scheme expression:

```
gosh>
```

After you complete a Scheme expression and type ENTER, the result of evaluation is printed.

```
gosh> (+ 1 2)
3
gosh>
```

The REPL session binds the last three results of evaluation in the global variables `*1`, `*2` and `*3`. You can use the previous results via those history variables in subsequent expressions.

```
gosh> *1
3
gosh> (+ *2 3)
6
```

If the Scheme expression yields multiple values (see Section 6.18.8 [Multiple values], page 197), they are printed one by one.

```
gosh> (min&max 1 -1 8 3)
-1
8
gosh>
```

The history variable `*1`, `*2` and `*3` only binds the first value. A list of all values are bound to `*1+`, `*2+` and `*3+`.

```
gosh> *1
-1
gosh> *2+
(-1 8)
```

(Note that, when you evaluate `*1` in the above example, the history is shifted—so you need to use `*2+` to refer to the result of `(min&max 1 -1 8 3)`.)

The `*history` procedure shows the value of history variables:

```
gosh> (*history)
*1: (-1 8)
*2: -1
*3: -1
gosh>
```

As a special case, if an evaluation yields zero values, history isn't updated. The `*history` procedure returns no values, so merely looking at the history won't change the history itself.

```
gosh> (*history)
*1: (-1 8)
*2: -1
*3: -1
gosh> (values)
gosh> (*history)
*1: (-1 8)
*2: -1
```

```
*3: -1
```

Finally, a global variable `*e` is bound to the last uncaught error condition object.

```
gosh> (filter odd? '(1 2 x 4 5))
*** ERROR: integer required, but got x
Stack Trace:
_______________________________________
  0  (eval expr env)
        At line 173 of "/usr/share/gauche-0.9/0.9.3.3/lib/gauche/interactive.scm"
gosh> *e
#<error "integer required, but got x">
```

(The error stack trace may differ depending on your installation.)

In REPL prompt, you can also enter special *top-level commands* for common tasks. Top-level commands are not Scheme expressions, not even S-expressions. They work like traditional line-oriented shell commands instead.

Top-level commands are prefixed by comma to be distinguished from ordinary Scheme expressions. To see what commands are available, just type `,help` and return.

```
gosh> ,help
You're in REPL (read-eval-print-loop) of Gauche shell.
Type a Scheme expression to evaluate.
A word preceded with comma has special meaning.  Type ,help <cmd>
to see the detailed help for <cmd>.
Commands can be abbreviated as far as it is not ambiguous.

 ,apropos|a  Show the names of global bindings that match the regexp.
 ,cd         Change the current directory.
 ,describe|d Describe the object.
 ,help|h     Show the help message of the command.
 ,history    Show REPL history.
 ,info|doc   Show info document for an entry of NAME, or search entries by REGEXP.
 ,load|l     Load the specified file.
 ,print-all|pa
             Print previous result (*1) without abbreviation.
 ,print-mode|pm
             View/set print-mode of REPL.
 ,pwd        Print working directory.
 ,reload|r   Reload the specified module, using gauche.reload.
 ,sh         Run command via shell.
 ,source     Show source code of the procedure if it's available.
 ,use|u      Use the specified module.  Same as (use module option ...).
```

To see the help of each individual commands, give the command name (without comma) to the `help` command:

```
gosh> ,help d
Usage: d|describe [object]
Describe the object.
Without arguments, describe the last REPL result.
```

The `,d` (or `,describe`) top-level command describes the given Scheme object or the last result if no object is given. Let's try some:

```
gosh> (sys-stat "/home")
#<<sys-stat> 0x2d6adc0>
gosh> ,d
```

```
#<<sys-stat> 0x2d6adc0> is an instance of class <sys-stat>
slots:
  type      : directory
  perm      : 493
  mode      : 16877
  ino       : 2
  dev       : 2081
  rdev      : 0
  nlink     : 9
  uid       : 0
  gid       : 0
  size      : 208
  atime     : 1459468837
  mtime     : 1401239524
  ctime     : 1401239524
```

In the above example, first we evaluated `(sys-stat "/home")`, which returns `<sys-stat>` object. The subsequent `,d` top-level command describes the returned `<sys-stat>` object.

The description depends on the type of objects. Some types of objects shows extra information. If you describe an exact integer, it shows alternative interpretations of the number:

```
gosh> ,d 1401239524
1401239524 is an instance of class <integer>
  (#x538537e4, ~ 1.3Gi, 2014-05-28T01:12:04Z as unix-time)
gosh> ,d 48
48 is an instance of class <integer>
  (#x30, #\0 as char, 1970-01-01T00:00:48Z as unix-time)
```

If you describe a symbol, its known bindings is shown.

```
gosh> ,d 'filter
filter is an instance of class <symbol>
Known bindings for variable filter:
  In module `gauche':
    #<closure (filter pred lis)>
  In module `gauche.collection':
    #<generic filter (2)>
```

If you describe a procedure, and its source code location is known, that is also shown (see the `Defined at...` line):

```
gosh> ,d string-interpolate
#<closure (string-interpolate str :optional (legacy? #f))> is an
instance of class <procedure>
Defined at "../lib/gauche/interpolate.scm":64
slots:
  required  : 1
  optional  : #t
  optcount  : 1
  locked    : #f
  currying  : #f
  constant  : #f
  info      : (string-interpolate str :optional (legacy? #f))
  setter    : #f
```

Let's see a couple of other top-level commands. The `,info` command shows the manual entry of the given procedure, variable, syntax, module or a class. (The text is searched from the installed info document of Gauche. If you get an error, check if the info document is properly installed.)

```
gosh> ,info append
 -- Function: append list ...
     [R7RS] Returns a list consisting of the elements of the first LIST
     followed by the elements of the other lists.  The resulting list is
     always newly allocated, except that it shares structure with the
     last list argument.  The last argument may actually be any object;
     an improper list results if the last argument is not a proper list.

gosh> ,info srfi-19
 -- Module: srfi-19
     This SRFI defines various representations of time and date, and
     conversion methods among them.

     On Gauche, time object is supported natively by '<time>' class
     (*note Time::).  Date object is supported by '<date>' class
     described below.

gosh> ,info <list>
 -- Builtin Class: <list>
     An abstract class represents lists.  A parent class of '<null>' and
     '<pair>'.  Inherits '<sequence>'.

     Note that a circular list is also an instance of the '<list>'
     class, while 'list?' returns false on the circular lists and dotted
     lists.
          (use srfi-1)
          (list? (circular-list 1 2)) => #f
          (is-a? (circular-list 1 2) <list>) => #t
```

You can also give a regexp pattern to `,info` command (see Section 6.13 [Regular expressions], page 162). It shows the entries in the document that match the pattern.

```
gosh> ,info #/^string-.*\?/
string-ci<=?              Full string case conversion:44
                          String Comparison:19
string-ci<?               Full string case conversion:43
                          String Comparison:18
string-ci=?               Full string case conversion:42
                          String Comparison:17
string-ci>=?              Full string case conversion:46
                          String Comparison:21
string-ci>?               Full string case conversion:45
                          String Comparison:20
string-immutable?         String Predicates:9
string-incomplete?        String Predicates:12
string-null?              SRFI-13 String predicates:6
string-prefix-ci?         SRFI-13 String Prefixes & Suffixes:28
string-prefix?            SRFI-13 String Prefixes & Suffixes:26
string-suffix-ci?         SRFI-13 String Prefixes & Suffixes:29
string-suffix?            SRFI-13 String Prefixes & Suffixes:27
```

The ,a command (or ,apropos) shows the global identifiers matches the given name or regexp:

```
gosh> ,a filter
filter                          (gauche)
filter!                         (gauche)
filter$                         (gauche)
filter-map                      (gauche)
```

Note: The apropos command looks for symbols from the current process—that is, it only shows names that have been loaded and imported. But it also mean it can show any name as far as it exists in the current process, regardless of whether it's a documented API or an internal entry.

On the other hand, the info command searches info document, regardless of the named entity has loaded into the current process or not. It doesn't show undocumented APIs.

You can think that apropos is an introspection tool, while info is a document browsing tool.

When the result of evaluation is a huge nested structure, it may take too long to display the result. Gauche actually set a limit of length and depth in displaying structures, so you might occasionally see the very long or deep list is trucated, with ... to show there are more items. (Try evaluating (make-list 100) on REPL).

You can type ,pa (or ,print-all) toplevel REPL command to fully redisplay the previous result without omission.

By default, REPL prints out the result using *pretty print*:

```
gosh> ,u sxml.ssax
gosh> (call-with-input-file "src/Info.plist" (cut ssax:xml->sxml <> '()))
(*TOP*
 (*PI* xml "version=\"1.0\" encoding=\"UTF-8\"")
 (plist
  (|@| (version "1.0"))
  (dict (key "CFBundleDevelopmentRegion") (string "English")
   (key "CFBundleExecutable") (string "Gauche") (key "CFBundleIconFile")
   (string) (key "CFBundleIdentifier") (string "com.schemearts.gauche")
   (key "CFBundleInfoDictionaryVersion") (string "6.0")
   (key "CFBundlePackageType") (string "FMWK") (key "CFBundleSignature")
   (string "????") (key "CFBundleVersion") (string "1.0")
   (key "NSPrincipalClass") (string))))
```

If you want to turn off pretty printing for some reason, type ,pm pretty #f (or ,print-mode pretty #f) on the toplevel prompt, or start gosh with the environment variable GAUCHE_REPL_NO_PPRINT set.

Type ,pm default to make print mode back to default. For more details, type ,help pm.

Note: If you invoke gosh with -q option, which tells gosh not to load the initialization files, you still get a REPL prompt but no fancy features such as history variables are available. Those convenience features are implemented in gauche.interactive module, which isn't loaded with -q option.

3.3 Writing Scheme scripts

When a Scheme program file is given to gosh, it makes the user module as the current module, binds a global variable *argv* to the list of the remaining command-line arguments, and then loads the Scheme program. If the first line of *scheme-file* begins with two character sequence "#!", the entire line is ignored by gosh. This is useful to write a Scheme program that works as an executable script in unix-like systems.

Typical Gauche script has the first line like these

```
#!/usr/local/bin/gosh
```
  or,
```
#!/usr/bin/env gosh
```
  or,
```
#!/bin/sh
:; exec gosh -- $0 "$@"
```

The second and third form uses a "shell trampoline" technique so that the script works as far as `gosh` is in the PATH. The third form is useful when you want to pass extra arguments to `gosh`, for typically `#!`-magic of executable scripts has limitations for the number of arguments to pass the interpreter.

After the file is successfully loaded, `gosh` calls a procedure named '`main`' if it is defined in the user module. `Main` receives a single argument, a list of command line arguments. Its first element is the script name itself.

When `main` returns, and its value is an integer, `gosh` uses it for exit code of the program. Otherwise, `gosh` exits with exit code 70 (`EX_SOFTWARE`). This behavior is compatible with the SRFI-22.

If the `main` procedure is not defined, `gosh` exits after loading the script file.

Although you can still write the program main body as toplevel expressions, like shell scripts or Perl scripts, it is much convenient to use this '`main`' convention, for you can load the script file interactively to debug.

Using `-m` command-line option, you can make `gosh` call `main` procedure defined in a module other than the `user` module. It is sometimes handy to write a Scheme module that can also be executed as a script.

For example, you write a Scheme module `foo` and *within it*, you define the `main` procedure. You don't need to export it. If the file is loaded as a module, the `main` procedure doesn't do anything. But if you specify `-m foo` option and give the file as a Scheme script to `gosh`, then the `main` procedure is invoked after loading the script. You can code tests or small example application in such an alternate main procedure.

Note on R7RS Scripts: If the script is written in R7RS Scheme (which can be distinguished by the first `import` declaration, see Section 10.1.2 [Three forms of import], page 501), it is read into `r7rs.user` module and its `main` isn't called. You can give `-mr7rs.main` command-line argument to call the `main` function in R7RS script. Alternatively, as specified in SRFI-22, if the script interpreter's basename is `scheme-r7rs`, we assume the script is R7RS SRFI-22 script and calls `main` in `r7rs.user` module rather than `user` module. We don't install such an alias, but you can manually make symbolic link or just copy `gosh` binary as `scheme-r7rs`.

Although the argument of the `main` procedure is the standard way to receive the command-line arguments, there are a couple of other ways to access to the info. See Section 6.25.2 [Command-line arguments], page 248, for the details.

Now I show several simple examples below. First, this script works like `cat(1)`, without any command-line option processing and error handling.

```
#!/usr/bin/env gosh

(define (main args)   ;entry point
  (if (null? (cdr args))
      (copy-port (current-input-port) (current-output-port))
      (for-each (lambda (file)
                  (call-with-input-file file
                    (lambda (in)
```

```
                        (copy-port in (current-output-port)))))
                  (cdr args)))
  0)
```

The following script is a simple grep command.

```
#!/usr/bin/env gosh

(define (usage program-name)
  (format (current-error-port)
          "Usage: ~a regexp file ...\n" program-name)
  (exit 2))

(define (grep rx port)
  (with-input-from-port port
    (lambda ()
      (port-for-each
       (lambda (line)
         (when (rxmatch rx line)
           (format #t "~a:~a: ~a\n"
                   (port-name port)
                   (- (port-current-line port) 1)
                   line)))
       read-line))))

(define (main args)
  (if (null? (cdr args))
      (usage (car args))
      (let ((rx (string->regexp (cadr args))))
        (if (null? (cddr args))
            (grep rx (current-input-port))
            (for-each (lambda (f)
                        (call-with-input-file f
                          (lambda (p) (grep rx p))))
                      (cddr args)))))
  0)
```

See also Section 9.23 [Parsing command-line options], page 413, for a convenient way to parse command-line options.

3.4 Debugging

Gauche doesn't have much support for debugging yet. The idea of good debugging interfaces are welcome.

For now, the author uses the classic 'debug print stub' technique when necessary. Gauche's reader supports special syntaxes beginning with `#?`, to print the intermediate value.

The syntax `#?=`*expr* shows *expr* itself before evaluating it, and prints its result(s) after evaluation.

```
gosh> #?=(+ 2 3)
#?="(stdin)":1:(+ 2 3)
#?-    5
5
gosh> #?=(begin (print "foo") (values 'a 'b 'c))
#?="(stdin)":2:(begin (print "foo") (values 'a 'b 'c))
```

```
foo
#?-    a
#?+    b
#?+    c
a
b
c
```

Note: If the debug stub is evaluated in a thread other than the primordial thread (see Section 9.33 [Threads], page 457), the output includes a number to distinguish which thread it is generated. In the following example, `#<thread ...>` and the prompt is the output of REPL in the primordial thread, but following `#?=[1]...` and `#?-[1]...` are the debug output from the thread created by `make-thread`. The number is for debugging only— they differ for each thread, but other than that there's no meaning.

```
gosh> (use gauche.threads)
gosh> (thread-start! (make-thread (^[] #?=(+ 2 3))))
#<thread #f (1) runnable 0xf51400>
gosh> #?=[1]"(standard input)":1:(+ 2 3)
#?-[1]    5
```

The syntax `#?,(`*proc arg* `...)` is specifically for procedure call; it prints the value of arguments right before calling *proc*, and prints the result(s) of call afterwards.

```
gosh> (define (fact n)
        (if (zero? n)
            1
            (* n #?,(fact (- n 1)))))
fact
#?,"(standard input)":4:calling `fact' with args:
#?,> 4
#?,"(standard input)":4:calling `fact' with args:
#?,> 3
#?,"(standard input)":4:calling `fact' with args:
#?,> 2
#?,"(standard input)":4:calling `fact' with args:
#?,> 1
#?,"(standard input)":4:calling `fact' with args:
#?,> 0
#?-    1
#?-    1
#?-    2
#?-    6
#?-    24
120
```

Internally, the syntax `#?=`*x* and `#?,`*x* are read as `(debug-print` *x*`)` and `(debuf-funcall` *x*`)`, respectively, and the macros `debug-print` and `debug-funcall` handles the actual printing. See Section 6.26.1 [Debugging aid], page 277, for more details.

The reasons of special syntax are: (1) It's easy to insert the debug stub, for you don't need to enclose the target expression by extra parenthesis, and (2) It's easy to find and remove those stabs within the editor.

3.5 Using platform-dependent features

Gauche tries to provide low-level APIs close to what the underlying system provides, but sometimes they vary among systems. For example, POSIX does not require `symlink`, so some systems may lack `sys-symlink` (see Section 6.25.4.2 [Directory manipulation], page 252). Quite a few unix-specific system functions are not available on Windows platform. To allow writing a portable program across those platforms, Gauche uses `cond-expand` (see Section 4.12 [Feature conditional], page 68) extensively. A set of extended *feature-identifier*s is provided to check availability of specific features. For example, on systems that has `symlink`, a feature identifier `gauche.sys.symlink` is defined. So you can write a code that can switch based on the availability of `sys-symlink` as follows:

```
(cond-expand
 (gauche.sys.symlink
   ... code that uses sys-symlink ...)
 (else
   ... alternative code ...)
 )
```

If you're familiar with system programming in C, you can think it equivalent to the following C idiom:

```
#if defined(HAVE_SYMLINK)
... code that uses symlink ...
#else
... alternative code ...
#endif
```

There are quite a few such feature identifiers; each identifier is explained in the manual entry of the procedures that depend on the feature. Here we list a few important ones:

`gauche`
: This feature identifier is always defined. It is useful when you write Scheme code portable across multiple implementations.

`gauche.os.windows`
: Defined on Windows native platform. Note that cygwin does not define this feature identifier (but see below).

`gauche.os.cygwin`
: Defined on Cygwin.

`gauche.sys.threads`
: Defined if Gauche is compiled with thread support. See Section 9.33 [Threads], page 457.

`gauche.sys.pthreads`
`gauche.sys.wthreads`
: Defined to indicate the underlying thread implementation when Gauche has thread support. See Section 9.33 [Threads], page 457.

`gauche.net.ipv6`
: Defined if Gauche is compiled with IPv6 support.

`gauche.ces.utf8`
`gauche.ces.eucjp`
`gauche.ces.sjis`
`gauche.ces.none`
: Either one of these feature identifiers is defined, according to the compile-time option of Gauche's internal character encoding. See Section 2.2 [Multibyte strings], page 11, for the details of the internal character encoding.

Because `cond-expand` is a macro, the body of clauses are expanded into toplevel if `cond-expand` itself is in toplevel. That means you can switch toplevel definitions:

```
(cond-expand
 (gauche.os.windows
  (define (get-current-user)
    ... get current username ...))
 (else
  (define (get-current-user)
    (sys-uid->user-name (sys-getuid)))))
```

Or even conditionally "use" the modules:

```
(cond-expand
 (gauche.os.windows
   (use "my-windows-compatibility-module"))
 (else))
```

The traditional technique of testing a toplevel binding (using `global-variable-bound?`, see Section 4.13.6 [Module introspection], page 76) doesn't work well in this case, since the `use` form takes effect at compile time. It is strongly recommended to use `cond-expand` whenever possible.

Currently the set of feature identifiers are fixed at the build time of Gauche, so it's less flexible than C preprocessor conditionals. We have a plan to extend this feature to enable adding new feature identifiers; but such feature can complicate semantics when compilation and execution is interleaved, so we're carefully assessing the effects now.

A couple of notes:

Feature identifiers are not variables. They can only be used within the *feature-requirement* part of `cond-expand` (see Section 4.12 [Feature conditional], page 68, for the complete definition of feature requirements).

By the definition of `srfi-0`, `cond-expand` raises an error if no feature requirements are satisfied and there's no `else` clause. A rule of thumb is to provide `else` clause always, even it does nothing (like the above example that has empty `else` clause).

3.6 Profiling and tuning

If you find your script isn't running fast enough, there are several possibilities to improve its performance.

It is always a good idea to begin with finding which part of the code is consuming the execution time. Gauche has a couple of basic tools to help it. A built-in sampling profiler, explained in the following subsection, can show how much time is spent in each procedure, and how many times it is called. The `gauche.time` module (Section 9.34 [Measure timings], page 468) provides APIs to measure execution time of specific parts of the code.

Optimization is specialization—you look for the most common patterns of execution, and put a special path to support those patterns efficiently. Gauche itself is no exception, so there are some patterns Gauche can handle very efficiently, while some patterns it cannot. The next subsection, Section 3.6.2 [Performance tips], page 31, will give you some tips of how to adapt your code to fit the patterns Gauche can execute well.

3.6.1 Using profiler

As of 0.8.4, Gauche has a built-in profiler. It is still experimental quality and only be tested on Linux. It isn't available for all platforms. It works only in single-thread applications for now.

To use the profiler non-interactively, give `-ptime` command-line option to gosh.

```
% gosh -ptime your-script.scm
```

After the execution of `your-script.scm` is completed, Gauche prints out the table of functions with its call count and its consumed time, sorted by the total consumed time.

```
Profiler statistics (total 1457 samples, 14.57 seconds)
                                                    num    time/    total
Name                                                calls  call(ms) samples
---------------------------------------------------+------+-------+-----------
combinations*                                       237351  0.0142   337( 23%)
(lset-difference #f)                               1281837  0.0020   256( 17%)
(make-anchor make-anchor)                          3950793  0.0005   198( 13%)
member                                             4627246  0.0004   190( 13%)
filter                                              273238  0.0030    81(  5%)
every                                              1315131  0.0004    59(  4%)
(lset-difference #f #f)                            1281837  0.0004    54(  3%)
(make-entry make-entry)                             730916  0.0005    40(  2%)
(clear? #f)                                         730884  0.0005    33(  2%)
(initialize #f)                                     599292  0.0005    32(  2%)
fold                                                237307  0.0013    30(  2%)
acons                                               806406  0.0004    29(  1%)
clear?                                               33294  0.0084    28(  1%)
(combinations* #f)                                  805504  0.0002    15(  1%)
(make-exit make-exit)                               730884  0.0002    15(  1%)
lset-difference                                     237318  0.0006    15(  1%)
reverse!                                            475900  0.0001     6(  0%)
(fold <top> <top> <list>)                           237323  0.0003     6(  0%)
procedure?                                          238723  0.0002     4(  0%)
pair?                                               237307  0.0001     3(  0%)
 :
 :
```

Note that the time profiler uses statistic sampling. Every 10ms the profiler interrupts the process and records the function that is executed then. Compared to the individual execution time per function call, which is the order of nanoseconds, this sampling rate is very sparse. However, if we run the program long enough, we can expect the distribution of samples per each function approximately reflects the distribution of time spent in each function.

Keep in mind that the number is only approximation; the number of sample counts for a function may easily vary if the program deals with different data sets. It should also be noted that, for now, GC time is included in the function in which GC is triggered. This sometimes causes a less important function to "float up" to near-top of the list. To know the general pattern, it is a good custom to run the program with several different data sets.

On the other hand, the call count is accurate since Gauche actually counts each call.

Because all functions are basically anonymous in Scheme, the 'name' field of the profiler result is only a hint. The functions bound at toplevel is generally printed with the global variable name it is bound at the first time. Internal functions are printed as a list of names, reflecting the nesting of functions. Methods are also printed as a list of the name and specializers.

The profiler has its own overhead; generally the total process time will increase 20-30%. If you want to turn on the profiler selectively, or you're running a non-stop server program and want to obtain the statistics without exiting the server, you can call the profiler API from your program; see Section 6.26.2 [Profiler API], page 278, for the details.

3.6.2 Performance tips

Don't guess, just benchmark. It is the first rule of performance tuning. Especially for the higher-level languages like Scheme, what impacts on performance greatly depends on the implementation. Certain operations that are very cheap on an implementation may be costly on others. Gauche has such implementation-specific characteristics, and to know some of them would help to see what to look out in the benchmark results.

"80% of execution time is spent in 20% of the code" is another old saying. Don't obscure your code by "potential" optimization that has little impact on the actual execution. We describe some tips below, but it doesn't mean you need to watch them all the time. It is better to keep most of the code clean and easy to understand, and only do tricks on the innermost loop.

Ports: To satisfy the specification of SRFI-18 (Threading), every call to I/O primitives of Gauche locks the port. This overhead may be visible if the application does a lot of I/O with smaller units (e.g. every bytes). The primitives that deals with larger unit, such as `read` and `read-uvector`, are less problematic, since usually they just lock the port once per call and do all the low-level I/O without the lock overhead. (Note: this doesn't mean those primitives *guarantee* to lock the port throughout the execution of the function; furthermore, the port locking feature is optimized for the case that port accesses rarely collide. If you know it is possible that more than one threads read from or write to the same port, it is your responsibility to use mutex explicitly to avoid the collision.)

If you find out the locking is indeed a bottleneck, there are couple of things you can consider: (1) Try using the larger-unit primitives, instead of calling the smaller-unit ones. (2) Use `with-port-locking` (see Section 6.22.2 [Port and threads], page 218) to lock the port in larger context.

Strings: Because of the multibyte strings, two operations are particularly heavy in Gauche: string mutation and indexed string access. It is a design choice; we encourage the programming style that avoids those operations. When you sequentially access the string, string ports (see Section 6.22.5 [String ports], page 224) provide a cleaner and more efficient way. When you search and retrieve a substring, there are various higher-level primitives are provided (see Section 6.12.7 [String utilities], page 158, Section 6.13 [Regular expressions], page 162, and Section 11.5 [String library], page 591, for example). If you're using strings to represent an octet sequence, use uniform vectors (see Section 9.36 [Uniform vectors], page 476) instead.

Deep recursion: Gauche's VM uses a stack for efficient local frame allocation. If recursion goes very deep (depending on the code, but usually several hundreds to a thousand), the stack overflows and Gauche moves the content of the stack into the heap. This incurs some overhead. If you observe a performance degradation beyond a certain amount of data, check out this possibility.

Generic functions: Because of its dynamic nature, generic function calls are slower than procedure calls. Not only because of the runtime dispatch overhead, but also because Gauche's compile-time optimizer can't do much optimization for generic function calls. You don't need to avoid generic functions because of performance reasons in general, but if you do find single function call consuming a large part of execution time and it is calling a generic function in its inner loop—then it may be worth to modify it.

Redefining builtin functions: Gauche inlines some builtin functions if they are not redefined. Although sometimes it is useful to redefine basic functions, you may want to limit the effect. For example, put redefined functions in a separate module and use the module in the code that absolutely needs those functions replaced.

Closure creation: When you create a closure, its closing environment is copied to the heap. This overhead is small, but it still may be visible when a closure is created within an innermost loop that is called millions of times. If you suspect this is a problem, try disassemble the function. Gauche's compiler uses some basic techniques of closure analysis to avoid creating closures for typical cases, in which case you see the local function's bodies are inlined. If you see a `CLOSURE` instruction, though, it means a closure is created.

This list isn't complete, and may change when Gauche's implementation is improved, so don't take this as fixed features. We'll adjust it occasionally.

3.7 Writing Gauche modules

Gauche's libraries are organized by modules. Although Gauche can load any valid Scheme programs, there is a convention that Gauche's libraries follow. When you write a chunk of

Scheme code for Gauche, it is convenient to make it a module, so that it can be shared and/or reused.

Usually a module is contained in a file, but you can make a multi-file module. First I explain the structure of a single-file module. The following template is the convention used in Gauche's libraries.

```
;; Define the module interface
(define-module foo
  (use xxx)
  (use yyy)
  (export foo1 foo2 foo3)
  )
;; Enter the module
(select-module foo)

... module body ...
```

This file must be saved as "foo.scm" in some directory in the *load-path*.

The define-module form creates a module foo. It also loads and imports some other modules by 'use' macros, and declares which symbols the foo module exports, by 'export' syntax. (See section Section 4.13.3 [Defining and selecting modules], page 73, for detailed specification of those syntaxes).

Those use forms or export forms are not required to appear in the define-module form, but it is a good convention to keep them in there at the head of the file so that it is visually recognizable which modules foo depends and which symbols it exports.

The second form, 'select-module', specifies the rest of the file is evaluated in the module foo you just defined. Again, this is just a convention; you can write entire module body inside define-module. However, I think it is error-prone, for the closing parenthesis can be easily forgotten or the automatic indentation mechanism of editor will be confused.

After select-module you can write whatever Scheme expression. It is evaluated in the selected module, foo. Only the bindings of the exported symbols will be directly accessible from outside.

So, that's it. Other programs can use your module by just saying '(use foo)'. If you want to make your module available on your site, you can put it to the site library location, which can be obtained by

```
(gauche-site-library-directory)
```

in gosh, or

```
gauche-config --sitelibdir
```

from shell.

If you feel like to conserve global module name space, you can organize modules hierarchically. Some Gauche libraries already does so. See Chapter 8 [Library modules - Overview], page 308, for examples. For example, text.tr module is implemented in "text/tr.scm" file. Note that the pathname separator '/' in the file becomes a period in the module name.

3.8 Using extension packages

Building and installing packages

Gauche comes with some amount of libraries, but they aren't enough at all to use Gauche in the production environment. There are number of additional libraries available. We call them *extension packages*, or simply packages. Each package usually provides one or more modules that adds extra functionality. Most of the packages provide binding to other C libraries, such as graphics libraries or database clients. If the package has some C code, it is likely that you need to compile it on your machine with the installed Gauche system.

Usually a package is in the form of compressed tarball, and the standard "ungzip + untar + configure + make + make install" sequence does the job. Read the package's document, for you may be able to tailor the library for your own needs by giving command-line options to the `configure` script.

From Gauche 0.8, an utility script called `gauche-package` is installed for the convenience. It automates the build and install process of packages.

Suppose you have downloaded a package `Package-1.0.tar.gz`. If the package follows the convention, all you have to do is to type this:

```
$ gauche-package install Package-1.0.tar.gz
```

It ungzips and untars the package, `cd` into the `Package-1.0` subdirectory, run configure, make, and make install. By default, `gauche-package` untars the tarball in the current working directory. You can change it by a customization file; see below.

If you need a special privilege to install the files, you can use `--install-as` option which runs `make install` part via the `sudo` program.

```
$ gauche-package install --install-as=root Package-1.0.tar.gz
```

If it doesn't work for you, you can just build the package by `gauche-package build Package-1.0.tar.gz`, then manually cd to the `Package-1.0` directory and run `make install`.

You can give configuration options via `-C` or `--configure-options` command-line argument, like this:

```
$ gauche-package install -C "--prefix=/usr/local" Package-1.0.tar.gz
```

If the package has adopted the new package description file, it can remember the configuration options you have specified, and it will automatically reuse them when you install the package again. (If you're a package developer, check out `examples/spigot/README` file in the Gauche source tree to see how to cooperate with Gauche's package management system.)

If you don't have a tarball in your local directory, but you know the URL where you can download it, you can directly give the URL to `gauche-package`. It understands `http` and `ftp`, and uses either `wget` or `ncftpget` to download the tarball, then runs configure and make.

```
$ gauche-package install http://www.example.com/Package-1.0.tar.gz
```

Customizing `gauche-package`

The `gauche-package` program reads `~/.gauche-package` if it exists. It must contain an associative list of parameters. It may look like this:

```
(
 (build-dir . "/home/shiro/tmp")
 (gzip      . "/usr/local/bin/gzip")
 (bzip2     . "/usr/local/bin/bzip2")
 (tar       . "/usr/local/bin/gtar")
)
```

The following is a list of recognized parameters. If the program isn't given in the configuration file, `gauche-package` searches `PATH` to find one.

`build-dir`
: A directory where the tarball is extracted. If URL is given, the downloaded file is also placed in this directory.

`bzip2`
: Path to the program `bzip2`.

`cat` Path to the program `cat`.

`make` Path to the program `make`.

`ncftpget` Path to the program `ncftpget`.

`rm` Path to the program `rm`.

`sudo` Path to the program `sudo`.

`tar` Path to the program `tar`.

`wget` Path to the program `wget`.

3.9 Building standalone executables

When you want to distribute your Gauche scripts or applications, the users need to install Gauche runtime on their machine. Although it is always the case for any language implementations—you need Java runtime to run Java applications, or C runtime to run C applications—it may be an extra effort to ask users to install not-so-standard language runtimes.

To ease distribution of Gauche applications, you can create a stand-alone executable. It statically links entire Gauche system so that it runs by just copying the executable file.

Quick recipe

To generate a standalone executable, just give your script file to the `build-standalone` script, which is installed as a part of Gauche.

```
gosh build-standalone yourscript.scm
```

It will create an executable file `yourscript` (or `yourscript.exe` on Windows) in the current directory.

To specify the output name different from the script name, give `-o` option:

```
gosh build-standalone -o yourcommand yourscript.scm
```

When your script needs supporting library files, you should list those files as well:

```
gosh build-standalone yourscript.scm lib/library1.scm lib/library2.scm
```

The library file paths need to be relative to the respective load path. See the explanation of `-I` option below.

Catches

There are a few things you should be aware of.

- The size of the binary tend to be large, since it contains the entire Gauche system regardless of whether your application use it or not. You can strip down the size if you need to, but you need to rebuild Gauche library to do so. See `doc/HOWTO-standalone.txt` in the source tree for the details.
- The generated binary still depends on external dynamically linked libraries, such as libpthread. The exact dependency may differ how Gauche is configured, and can be checked by running system-provided tools, such as `ldd` on most Unix systems and MinGW or `otool -L` on OSX, on the generated standalone binaries. You may want to ensure the users have required libraries.
- Currently we don't yet have a convenient way to statically link extension libraries. We're working on it.
- If Gauche is configured to use gdbm, it is linked to the standalone binary by default, hence the binary itself is covered by GPL. In case if you need to distribute binaries under BSD license, you need to give `-D GAUCHE_STATIC_EXCLUDE_GDBM` flag to `build-standalone`. It makes `build-standalone` not to link gdbm (and your script won't be able to use it).

- If you build Gauche with mbedTLS support (if you have libmbedtls on your machine, Gauche include its support by default), the resulting standalone binary also depends on libmbedtls DSO files. If you're not sure mbedTLS DSO files are available on target machines, you can exclude `rfc.tls.mbed` module by giving `-D GAUCHE_STATIC_EXCLUDE_MBED` flag to `build-standalone`.

Using build-standalone

`gosh build-standalone` [*options*] *script-file* [*library-file* . . .] [Program]

Create a stand-alone binary to execute a Gauche program in *script-file*. It is executed as if it is run as `gosh script-file`, with a few differences.

The main thing is that since *script-file* is no longer loaded from file, code that references paths relative to *script-file* won't work. One example is `(add-load-path dir :relative)` (see Section 6.23.1 [Loading Scheme file], page 239). Auxiliary library files required by *script-file* must be explicitly listed as *library-file* . . . , so that they are bundled together into the executable.

The following command-line options are recognized.

`-o` *outfile* [Command Option]

Specifies output executable filename. When omitted, the basename of *script-file* without extension is used. (Or, on Windows, swapping extension with `.exe`).

`-D` *var*[=*val*] [Command Option]

Add C preprocessor definitions while compiling the generated C code. An important use case of this option is to exclude gdbm dependency from the generated binaries, by specifying `-D GAUCHE_STATIC_EXCLUDE_GDBM`. Note that you need a whitespace between `-D` and *var*.

This option can be specified multiple times.

`-I` *load-path* [Command Option]

Specifies the load path where *library-file* . . . are searched. The names given to *library-file* must match how they are `load`ed or `use`d. If such paths are not relative to the directory you run `build-standalone`, you have to tell where to find those libraries with this option.

For example, suppose you have this structure:

```
project/src/
    +----- main.scm
    |          (use myscript.util)
    +----- myscript/util.scm
               (define-module myscript.util ...)
```

If you run `build-standalone` in the directory as `src`, you can just say this:

```
gosh build-standalone main.scm myscript/util.scm
```

But if you run it under `project`, you need to say this:

```
gosh build-standalone -I src src/main.scm myscript/util.scm
```

Another example; you have a separate library directory:

```
project/
    +----- src/main.scm
    |          (use myscript.util)
    +----- lib/myscript/util.scm
               (define-module myscript.util ...)
```

If you run `build-standalone` in `src`, you say this:

```
gosh build-standalone -I ../lib main.scm myscript/util.scm
```

Or, if you run it in `project`, you say this:

```
gosh build-standalone -I lib src/main.scm myscript/util.scm
```

This option can be specified multiple times. Note that a whitespace is required between `-I` and *load-path*.

`--header-dir` *dir* [Command Option]
`--library-dir` *dir* [Command Option]

These tells `build-standalone` where to find Gauche C headers and static libraries.

If you've installed Gauche on your system, `build-standalone` automatically finds these from the installed directory and you don't need to worry about them. Use these option only when you need to use Gauche runtime that's not installed.

4 Core syntax

4.1 Lexical structure

Gauche extends R7RS Scheme lexical parser in some ways. Besides, because of historical reasons, a few of the default lexical syntax may conflict R7RS specification. You can set a reader mode to make it R7RS compliant.

Hash-bang directives

Tokens beginning with `#!` may have special meanings to the reader. R7RS defines two of such directives—`#!fold-case` and `#!no-fold-case`, which switches whether symbols are read in case-folding or non-case-folding mode, respectively.

see Section 4.1.2 [Hash-bang token], page 41, below, for all the directives Gauche has.

Square brackets

Gauche adopts the R6RS syntax that regards `[]` the same as `()`. Both kind of parentheses are equivalent, but the kind of corresponding open and close parentheses must match. Some seasoned Lisper may frown on them, but it helps visually distinguish different roles of parentheses.

A general convention is to use `[]` for groupings other than function and macro application. If such grouping nests, however, use `()` for outer groupings. Examples:

```
(cond [(test1 x) (y z)]
      [(test2 x) (s t)]
      [else (u v)])

(let ([x (foo a b)]
      [y (bar c d)])
  (baz x y))
```

It is purely optional, so you don't need to use them if you don't like them. R7RS doesn't adopt this syntax and leaves `[]` for extensions, so it is safe to stick to `()` in portable R7RS programs. (If the reader is in `strict-r7` mode, an error is signalled when `[]` is used. See Section 6.22.7.2 [Reader lexical mode], page 228, for the details.)

Scheme-specific modes of some editors (e.g. Quack on Emacs) allows you to type just `)` and inserts either `]` or `)` depending on which kind parenthesis it is closing. We recommend using such modes if you use this convention.

Symbol names

Symbol names are case sensitive by default (see Section 2.4 [Case-sensitivity], page 12). Symbol name can begin with digits, '`+`' or '`-`', as long as the entire token doesn't consist valid number syntax. Other weird characters can be included in a symbol name by surrounding it with '`|`', e.g. '`|this is a symbol|`'. See Section 6.7 [Symbols], page 137, for details.

Numeric literals

Either integral part or fraction part of inexact real numbers can be omitted if it is zero, i.e. `30.`, `.25`, `-.4` are read as real numbers. The number reader recognizes '`#`' as insignificant digits. Complex numbers can be written both in the rectangular format (e.g. `1+0.3i`) and in the polar format (e.g. `3.0@1.57`). Inexact real numbers include the positive infinity, the negative infinity, and NaN, which are represented as `+inf.0`, `-inf.0` and `+nan.0`, respectively. (`-nan.0` is also read as NaN.)

As an extension of Gauche, a character _ can be inserted in or around a sequence of digits in number literals, as far as the literal is explicitly prefixed (#e, #x, etc). Those _'s are just ignored. It is to improve readability, e.g. #b1100_1010_1111_1110.

Gauche also adopts Common-Lisp style radix prefixed numeric literals, e.g. #3r120 (120 in base-3, 15 in decimal). Radix between 2 and 36 are recognized; alphabetic letters a-zA-Z are used beyond decimal.

For the polar notation of complex numbers, Gauche allows the suffix pi to denote the phase by multiples of pi. The Scheme syntax use radians for the phase, but you can only approximate pi with the floating point numbers, so it can't represent round numbers except zero angle.

```
gosh> 2@3.141592653589793
-2.0+2.4492935982947064e-16i
```

With the pi suffix, you can get a round numbers.

```
gosh> 2@1pi
-2.0
gosh> 2@0.5pi
0.0+2.0i
gosh> 2@-0.5pi
0.0-2.0i
```

Hex character escapes

You can denote a character using hexadecimal notation of the character code in some literals; specifically, character literals, charcter set literals, string literals, symbols, regular expression literals.

R7RS adopted a hex escape notation \xNNNN; for strings and symbols surrounded by vertical bars, and #\xNNNN for characters. The number of digits is variable, and the character code is Unicode codepoint.

Gauche had been using two types of escapes; \u and \x. In general, u is for Unicode codepoint, while x is for the character code in the internal encoding. Besides, except character literals, we used fixed number of digits, instead of using the terminator ; as in R7RS.

Since 0.9.4, we interpret \x-escape as R7RS whenever if it consists a valid R7RS hex-escape, and if not, try to interpret it as legacy Gauche hex-escape.

Although rarely, there are cases that can interpreted both in R7RS syntax and legacy Gauche syntax, but yielding different characters. Reading legacy files with such literals in the current Gauche may cause unexpected behavior. You can switch the reader mode so that it becomes backward-compatible. See Section 6.22.7.2 [Reader lexical mode], page 228, for the details.

Extended sharp syntax

Many more special tokens begins with '#' are defined. See the table below.

4.1.1 Sharp syntax

The table below lists sharp-syntaxes.

`#!` [R6RS][R7RS][SRFI-22] It is either a beginning of an interpreter line (shebang) of a script, or a special token that affects the mode of the reader. See 'hash-bang token' section below.

`#"` Introduces an interpolated string. See Section 6.12.4 [String interpolation], page 155.

`##`, `#$`, `#%`, `#&`, `#'` Unused.

#(	[R7RS] Introduces a vector.
#)	Unused.
#*	If followed by a double quote, denotes an incomplete string. See Section 6.12 [Strings], page 153.
#+	Unused.
#,	[SRFI-10] Introduces reader constructor syntax.
#-, #.	Unused.
#/	Introduces a literal regular expression. See Section 6.13 [Regular expressions], page 162.
#0 ... #9	#n#, #n=: [SRFI-38] Shared substructure definition and reference. #nR, #nr: Radix prefixed numeric literals.
#:	Uninterned symbol. See Section 6.7 [Symbols], page 137.
#;	[SRFI-62] S-expression comment. Reads next one S-expression and discard it.
#<	Introduces an unreadable object.
#=, #>	Unused.
#?	Introduces debug macros. See Section 3.4 [Debugging], page 27.
#@	Unused.
#a	Unused.
#b	[R7RS] Binary number prefix.
#c	Unused.
#d	[R7RS] Decimal number prefix.
#e	[R7RS] Exact number prefix.
#f	[R7RS] Boolean false, or introducing R7RS uniform vector. See Section 9.36 [Uniform vectors], page 476. R7RS defines both #f and #false as a boolean false value.
#g, #h	Unused.
#i	[R7RS] Inexact number prefix.
#j, #k, #l, #m, #n	Unused.
#o	[R7RS] Octal number prefix.
#p, #q, #r	Unused.
#s	[R7RS vector.@] introducing R7RS uniform vector. See Section 9.36 [Uniform vectors], page 476.
#t	[R7RS] Boolean true. R7RS defines #t and #true as a boolean true value.
#u	[R7RS vector.@] introducing R7RS uniform vector. See Section 9.36 [Uniform vectors], page 476. R7RS uses #u8 prefix for bytevectors, which is compatible to u8 uniform vectors.
#v, #w	Unused.
#x	[R7RS] Hexadecimal number prefix.
#y, #z	Unused.
#[	Introduces a literal character set. See Section 6.11 [Character set], page 147.
#\	[R7RS] Introduces a literal character. See Section 6.10 [Characters], page 143.
#], #^, #_	Unused.

`` #` ``	Legacy syntax for string interpolation, superseded by `#"`.
`#{`	Unused.
`#|`	[SRFI-30] Introduces a block comment. Comment ends by matching '`|#`'.
`#}`, `#~`	Unused.

4.1.2 Hash-bang token

A character sequence `#!` has two completely different semantics, depending on how and where it occurs.

If a file begins with `#!/` or `#! ` (hash, bang, and a space), then the reader assumes it is an interpreter line (shebang) of a script and ignores the rest of characters until the end of line. (Actually the source doesn't need to be a file. The reader checks whether it is the beginning of a port.)

Other than the above case, `#!`*`identifier`* is read as a token with special meanings. This kind token can be a special directive for the reader, instead of read as a datum.

By default, the following tokens are recognized.

`#!fold-case`
`#!no-fold-case`
: Switches the reader's case sensitivity; `#!fold-case` makes the reader case insensitive, and `#!no-fold-case` makes it case sensitive. (Also see Section 2.4 [Case-sensitivity], page 12).

`#!r6rs`
: This token is introduced in R6RS and used to indicate the program strictly conforms R6RS. Gauche doesn't conform R6RS, but currently it just issues warning when it sees `#!r6rs` token, and it keeps reading on.

`#!r7rs`
: Make the reader `strict-r7` mode, that complies R7RS. See Section 6.22.7.2 [Reader lexical mode], page 228, for the details.

`#!gauche-legacy`
: Make the reader `legacy` mode, that is compatible to Gauche 0.9.3 and before. See Section 6.22.7.2 [Reader lexical mode], page 228, for the details.

4.2 Literals

`quote` *datum* [Special Form]

[R7RS base] Evaluates to *datum*.

```
(quote x) ⇒ x
(quote (1 2 3)) ⇒ (1 2 3)
```

`'`*`datum`* [Reader Syntax]

[R7RS] Equivalent to `(quote datum)`.

```
'x ⇒ x
'(1 2 3) ⇒ (1 2 3)
```

Note: RnRS says it is an error to alter the value of a literal expression. Gauche doesn't check constant-ness of pairs and vectors, and does not signal an error if you modify them using the destructive procedures such as `set-car!` and `vector-set!`. Doing so will cause unexpected results. Gauche does check constant-ness of strings, and signals an error if you try to alter a constant string.

4.3 Making procedures

`lambda` *formals body* ... [Special Form]
`^` *formals body* ... [Special Form]

[R7RS+] Evaluates to a procedure. The environment in effect when this expression is evaluated is stored in the procedure. When the procedure is called, *body* is evaluated sequentially in the stored environment extended by the bindings of the formal arguments, and returns the value(s) of the last expression in the body.

`^` is a concise alias of `lambda`. It is Gauche's extension.

```
(lambda (a b) (+ a b))
  ⇒ procedure that adds two arguments

((lambda (a b) (+ a b)) 1 2) ⇒ 3

((^(a b) (+ a b)) 1 2)       ⇒ 3
```

Gauche also extends R7RS `lambda` to take extended syntax in *formals* to specify optional and keyword arguments easily. The same functionality can be written in pure R7RS, with parsing variable-length arguments explicitly, but the code tends to be longer and verbose. It is recommended to use extended syntax unless you're writing portable code.

Formals should have one of the following forms:

- `(variable ...)` : The procedure takes a fixed number of arguments. The actual arguments are bound to the corresponding variables.

  ```
  ((lambda (a) a) 1)    ⇒ 1

  ((lambda (a) a) 1 2)  ⇒ error - wrong number of arguments
  ```

- `variable` : The procedure takes any number of arguments. The actual arguments are collected to form a new list and bound to the variable.

  ```
  ((lambda a a) 1 2 3)  ⇒ (1 2 3)
  ```

- `(variable_0 ... variable_N-1 . variable_N)` : The procedure takes at least N arguments. The actual arguments up to N is bound to the corresponding variables. If more than N arguments are given, the rest arguments are collected to form a new list and bound to *variable_N*.

  ```
  ((lambda (a b . c) (print "a=" a " b=" b " c=" c)) 1 2 3 4 5)
   ⇒ prints a=1 b=2 c=(3 4 5)
  ```

- `(variable ... extended-spec ...)` : Extended argument specification. Zero or more variables that specifies required formal arguments, followed by an *extended spec*, a list beginning with a keyword `:optional`, `:key` or `:rest`.

  The *extended-spec* part consists of the optional argument spec, the keyword argument spec and the rest argument spec. They can appear in any combinations.

  `:optional` *optspec* ...

  Specifies optional arguments. Each *optspec* can be either one of the following forms:

  ```
  variable
  (variable init-expr)
  ```

  The *variable* names the formal argument, which is bound to the value of the actual argument if given, or the value of the expression *init-expr* otherwise. If *optspec* is just a variable, and the actual argument is not given to it, then *variable* will be bound to `#<undef>` (see Section 6.5 [Undefined values], page 124).

The expression *init-expr* is only evaluated if the actual argument for *variable* is not given. The scope in which *init-expr* is evaluated includes the preceding formal arguments.

```
((lambda (a b :optional (c (+ a b))) (list a b c))
 1 2)    ⇒ (1 2 3)

((lambda (a b :optional (c (+ a b))) (list a b c))
 1 2 -1) ⇒ (1 2 -1)

((lambda (a b :optional c) (list a b c))
 1 2)    ⇒ (1 2 #<undef>)

((lambda (:optional (a 0) (b (+ a 1))) (list a b))
 )       ⇒ (0 1)
```

The procedure signals an error if more actual arguments than the number of required and optional arguments are given, unless it also has `:key` or `:rest` argument spec.

```
((lambda (:optional a b) (list a b)) 1 2 3)
 ⇒ error - too many arguments

((lambda (:optional a b :rest r) (list a b r)) 1 2 3)
 ⇒ (1 2 (3))
```

`:key` *keyspec* `...` `[:allow-other-keys [`*variable*`]]`

Specifies keyword arguments. Each *keyspec* can be either one of the following forms.

```
variable
(variable init-expr)
((keyword variable) init-expr)
```

The *variable* names the formal argument, which is bound to the actual argument given with the keyword of the same name as *variable*. When the actual argument is not given, *init-expr* is evaluated and the result is bound to *variable* in the second and third form, or `#<undef>` is bound in the first form.

```
(define f (lambda (a :key (b (+ a 1)) (c (+ b 1)))
            (list a b c)))

(f 10)            ⇒ (10 11 12)
(f 10 :b 4)       ⇒ (10 4 5)
(f 10 :c 8)       ⇒ (10 11 8)
(f 10 :c 1 :b 3)  ⇒ (10 3 1)
```

With the third form you can name the formal argument differently from the keyword to specify the argument.

```
((lambda (:key ((:aa a) -1)) a) :aa 2)
  ⇒ 2
```

By default, the procedure with keyword argument spec raises an error if a keyword argument with an unrecognized keyword is given. Giving `:allow-other-keys` in the formals suppresses this behavior. If you give *variable* after `:allow-other-keys`, the list of unrecognized keywords and their arguments are bound to it. Again, see the example below will help to understand the behavior.

```
((lambda (:key a) a)
 :a 1 :b 2)  ⇒ error - unknown keyword :b

((lambda (:key a :allow-other-keys) a)
 :a 1 :b 2)  ⇒ 1

((lambda (:key a :allow-other-keys z) (list a z))
 :a 1 :b 2)  ⇒ (1 (:b 2))
```

When used with `:optional` argument spec, the keyword arguments are searched after all the optional arguments are bound.

```
((lambda (:optional a b :key c) (list a b c))
 1 2 :c 3)  ⇒ (1 2 3)

((lambda (:optional a b :key c) (list a b c))
 :c 3)      ⇒ (:c 3 #<undef>)

((lambda (:optional a b :key c) (list a b c))
 1 :c 3)     ⇒ error - keyword list not even
```

`:rest` *variable*

Specifies the rest argument. If specified without `:optional` argument spec, a list of remaining arguments after required arguments are taken is bound to *variable*. If specified with `:optional` argument spec, the actual arguments are first bound to required and all optional arguments, and the remaining arguments are bound to *variable*.

```
((lambda (a b :rest z) (list a b z))
 1 2 3 4 5)  ⇒ (1 2 (3 4 5))

((lambda (a b :optional c d :rest z) (list a b c d z))
 1 2 3 4 5)  ⇒ (1 2 3 4 (5))

((lambda (a b :optional c d :rest z) (list a b c d z))
 1 2 3)      ⇒ (1 2 3 #<undef> ())
```

When the rest argument spec is used with the keyword argument spec, both accesses the same list of actual argument—the remaining arguments after required and optional arguments are taken.

```
((lambda (:optional a :rest r :key k) (list a r k))
 1 :k 3)  ⇒ (1 (:k 3) 3)
```

See also `let-optionals*`, `let-keywords` and `let-keywords*` macros in Section 6.18.4 [Optional argument parsing], page 192, for an alternative way to receive optional/keyword arguments within the spec of R7RS.

`^c` *body* ... [Macro]

A shorthand notation of `(lambda (`*c*`)` *body* `...)`. where *c* can be any character in `#[_a-z]`.

```
(map (^x (* x x)) '(1 2 3 4 5)) ⇒ (1 4 9 16 25)
```

`cut` *expr-or-slot expr-or-slot2* ... [Macro]
`cute` *expr-or-slot expr-or-slot2* ... [Macro]

[SRFI-26] Convenience macros to notate a procedure compactly. This form can be used to realize partial application, a.k.a sectioning or projection.

Each *expr-or-slot* must be either an expression or a symbol `<>`, indicating a 'slot'. The last *expr-or-slot* can be a symbol `<...>`, indicating a 'rest-slot'. `Cut` expands into a `lambda` form that takes as many arguments as the number of slots in the given form, and whose body is an expression

```
(expr-or-slot expr-or-slot2 ...)
```

where each occurrence of `<>` is replaced to the corresponding argument. In case there is a rest-slot symbol, the resulting procedure is also of variable arity, and all the extra arguments are passed to the call of *expr-or-slot*. See the fourth example below.

```
(cut cons (+ a 1) <>)  ≡ (lambda (x2) (cons (+ a 1) x2))
(cut list 1 <> 3 <> 5) ≡ (lambda (x2 x4) (list 1 x2 3 x4 5))
(cut list)             ≡ (lambda () (list))
(cut list 1 <> 3 <...>)
   ≡ (lambda (x2 . xs) (apply list 1 x2 3 xs))
(cut <> a b)           ≡ (lambda (f) (f a b))

;; Usage
(map (cut * 2 <>) '(1 2 3 4))
(for-each (cut write <> port) exprs)
```

`Cute` is a variation of `cut` that evaluates *expr-or-slot*s before creating the procedure.

```
(cute cons (+ a 1) <>)
   ≡ (let ((xa (+ a 1))) (lambda (x2) (cons xa x2)))
```

Gauche provides a couple of different ways to write partial applications concisely; see the `$` macro below, and also the `pa$` procedure (see Section 6.18.3 [Combinators], page 190).

`$` *arg* . . . [Macro]

A macro to chain applications, hinted from Haskell's `$` operator (although the meaning is different). Within the macro arguments *arg* . . ., `$` delimits the last argument. For example, the following code makes the last argument for the procedure `f` to be `(g c d ...)`

```
($ f a b $ g c d ...)
≡ (f a b (g c d ...))
```

The `$` notation can be chained.

```
($ f a b $ g c d $ h e f ...)
≡ (f a b (g c d (h e f ...)))
```

If `$*` appears in the argument list instead of `$`, it fills the rest of the arguments, instead of just the last argument.

```
($ f a b $* g c d ...)
≡ (apply f a b (g c d ...))

($ f a b $* g $ h $* hh ...)
≡ (apply f a b (g (apply h (hh ...))))
```

Furthermore, if the argument list ends with `$` or `$*`, the whole expression becomes a procedure expecting the last argument(s).

```
($ f a b $ g c d $ h e f $)
≡ (lambda (arg) (f a b (g c d (h e f arg))))
≡ (.$ (cut f a b <>) (cut g c d <>) (cut h e f <>))

($ f a b $ g c d $ h e f $*)
≡ (lambda args (f a b (g c d (apply h e f args))))
≡ (.$ (cut f a b <>) (cut g c d <>) (cut h e f <...>))
```

The more functional the code becomes, the more you tempted to write it as a chain of nested function calls. Scheme's syntax can get awkward in such code. Close parentheses tend to clutter at the end of expressions. Inner applications tends to pushed toward right columns with the standard indentation rules. Compare the following two code functionally equivalent to each other:

```
(intersperse ":"
             (map transform-it
                  (delete-duplicates (map cdr
                                          (group-sequence input)))))


($ intersperse ":"
   $ map transform-it
   $ delete-duplicates
   $ map cdr $ group-sequence input)
```

It is purely a matter of taste, and also this kind of syntax sugars can be easily abused. Use with care, but it may work well if used sparingly, like spices.

As a corner case, if neither `$` nor `$*` appear in the argument list, it just calls the function.

```
($ f a b c) ≡ (f a b c)
```

`case-lambda` *clause* . . . [Macro]

[R7RS case-lambda] Each *clause* should have the form (*formals expr* . . .), where *formals* is a formal arguments list as for `lambda`.

This expression evaluates to a procedure that accepts a variable number of arguments and is lexically scoped in the same manner as procedures resulting from `lambda` expressions. When the procedure is called with some arguments, then the first *clause* for which the arguments agree with *formals* is selected, where agreement is specified as for the *formals* of a `lambda` expression. The variables of `formals` are bound to the given arguments, and the *expr* . . . are evaluated within the environment.

It is an error for the arguments not to agree with the *formals* of any *clause*.

```
(define f
  (case-lambda
    [() 'zero]
    [(a) `(one ,a)]
    [(a b) `(two ,a ,b)]))

(f)       ⇒ zero
(f 1)     ⇒ (one 1)
(f 1 2)   ⇒ (two 1 2)
(f 1 2 3) ⇒ Error: wrong number of arguments to case lambda

(define g
  (case-lambda
    [() 'zero]
    [(a) `(one ,a)]
    [(a . b) `(more ,a ,@b)]))

(g)       ⇒ zero
(g 1)     ⇒ (one 1)
(g 1 2 3) ⇒ (more 1 2 3)
```

Note that the clauses are examined sequentially to match the number of arguments, so in the following example `g2` never returns `(one ...)`.

```
(define g2
```

```
  (case-lambda
    [() 'zero]
    [(a . b) `(more ,a ,@b)]
    [(a) `(one ,a)]))

(g2 1)    ⇒ (more 1)
```

4.4 Assignments

`set!` *symbol expression* [Special Form]
`set!` (*proc arg . . .*) *expression* [Special Form]

[R7RS+ base][SRFI-17] First, *expression* is evaluated. In the first form, the binding of *symbol* is modified so that next reference of *symbol* will return the result of *expression*. If *symbol* is not locally bound, the global variable named *symbol* must already exist, or an error is signaled.

The second form is a "generalized set!" specified in SRFI-17. It is a syntactic sugar of the following form.

```
((setter proc) arg ... expression)
```

Note the order of the arguments of the setter method differs from CommonLisp's `setf`.

Some examples:

```
(define x 3)
(set! x (list 1 2))
x                    ⇒ (1 2)

(set! (car x) 5)
x                    ⇒ (5 2)
```

`set!-values` (*var . . .*) *expr* [Macro]

Sets values of multiple variables at once. *Expr* must yield as many values as *var* . . . . Each value is set to the corresponding *var*.

```
(define a 0)
(define b 1)
(set!-values (a b) (values 3 4))
a ⇒ 3
b ⇒ 4
(set!-values (a b) (values b a))
a ⇒ 4
b ⇒ 3
```

`setter` *proc* [Function]

[SRFI-17] Returns a setter procedure associated to the procedure *proc*. If no setter is associated to *proc*, its behavior is undefined.

A setter procedure *g* of a procedure *f* is such that when used as `(g a b ... v)`, the next evaluation of `(f a b ...)` returns *v*.

To associate a setter procedure to another procedure, you can use the setter of `setter`, like this:

```
(set! (setter f) g)
```

A procedure's setter can be "locked" to it. System default setters, like `set-car!` for `car`, is locked and can't be set by the above way. In order to lock a setter to a user defined procedure, use `getter-with-setter` below.

If *proc* is not a procedure, a setter generic function of `object-apply` is returned; it allows the applicable object extension to work seamlessly with the generalized `set!`. See Section 6.18.6 [Applicable objects], page 195, for the details.

`has-setter?` *proc* [Function]

Returns `#t` if a setter is associated to *proc*.

`getter-with-setter` *get set* [Function]

[SRFI-17] Takes two procedure *get* and *set*. Returns a new procedure which does the same thing as *get*, and its setter is locked to *set*.

The intention of this procedure is, according to the SRFI-17 document, to allow implementations to inline setters efficiently. Gauche hasn't implement such optimization yet.

A few macros that adopts the same semantics of generalized `set!` are also provided. They are built on top of `set!`.

`push!` *place item* [Macro]

Conses *item* and the value of *place*, then sets the result to *place*. *place* is either a variable or a form (*proc arg* ...), as the second argument of `set!`. The result of this form is undefined.

```
(define x (list 2))
(push! x 3)
x ⇒ (3 2)

(push! (cdr x) 4)
x ⇒ (3 4 2)
```

When *place* is a list, it roughly expands like the following.

```
(push! (foo x y) item)
 ≡
 (let ((tfoo foo)
       (tx x)
       (ty y))
   ((setter tfoo) tx ty (cons item (tfoo tx ty))))
```

Note: Common Lisp's `push` macro takes its argument reverse order. I adopted this order since it is consistent with other destructive operations. Perl's `push` function takes the same argument order, but it appends *item* at the end of the array (Perl's `unshift` is closer to `push!`). You can use a queue (see Section 12.12 [Queue], page 656) if you need a behavior of Perl's `push`.

`pop!` *place* [Macro]

Retrieves the value of *place*, sets its `cdr` back to *place* and returns its `car`.

```
(define x (list 1 2 3))
(pop! x) ⇒ 1
x ⇒ (2 3)

(define x (vector (list 1 2 3)))
x ⇒ #((1 2 3))
(pop! (vector-ref x 0)) ⇒ 1
x ⇒ #((2 3))
```

Note: This works the same as Common Lisp's `pop`. Perl's `pop` pops value from the end of the sequence; its `shift` does the same thing as `pop!`.

`inc!` *place :optional delta* [Macro]
`dec!` *place :optional delta* [Macro]

Evaluates the value of *place*. It should be a number. Adds (`inc!`) or subtracts (`dec!`) *delta* to/from it, and then stores the result to *place*. The default value of *delta* is 1.

This is like Common Lisp's `incf` and `decf`, except that you can't use the result of `inc!` and `dec!`.

`update!` *place proc* [Macro]

Generalized form of `push!` etc. *Proc* must be a procedure which takes one argument and returns one value. The original value of *place* is passed to the *proc*, then its result is set to *place*.

```
(define a (cons 2 3))
(update! (car a) (lambda (v) (* v 3)))
a ⇒ (6 . 3)

(update! (cdr a) (cut - <> 3))
a ⇒ (6 . 0)
```

4.5 Conditionals

`if` *test consequent alternative* [Special Form]
`if` *test consequent* [Special Form]

[R7RS base] *Test* is evaluated. If it yields a true value, *consequent* is evaluated. Otherwise, *alternative* is evaluated. If *alternative* is not provided, it results undefined value.

```
(if (number? 3) 'yes 'no) ⇒ yes
(if (number? #f) 'yes 'no) ⇒ no

(let ((x '(1 . 2)))
  (if (pair? x)
      (values (car x) (cdr x))
      (values #f #f)))
  ⇒ 1 and 2
```

`cond` *clause1 clause2 . . .* [Special Form]

[R7RS+ base][SRFI-61] Each *clause* must be the form

```
(test expr ...)
(test => expr)
(test guard => expr)
(else expr expr2 ...)
```

The last form can appear only as the last clause.

`cond` evaluates *test* of each clauses in order, until it yields a true value. Once it yields true, if the clause is the first form, the corresponding *expr*s are evaluated and the result(s) of last *expr* is(are) returned; if the clause is the second form, the *expr* is evaluated and it must yield a procedure that takes one argument. Then the result of *test* is passed to it, and the result(s) it returns will be returned.

The third form is specified in SRFI-61. In this form, *test* can yield arbitrary number of values. The result(s) of *test* is(are) passed to *guard*; if it returns a true value, *expr* is applied with an equivalent argument list, and its result(s) is(are) returned. If *guard* returns `#f`, the evaluation proceeds to the next clause.

If no test yields true, and the last clause is not the fourth form (else clause), an undefined value is returned.

If the last clause is else clause and all tests are failed, *expr*s in the else clause are evaluated, and its last *expr*'s result(s) is(are) returned.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less)) ⇒ greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal)) ⇒ equal
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f)) ⇒ 2
```

`case` *key clause1 clause2 . . .* [Special Form]

[R7RS+ base][SRFI-87] *Key* may be any expression. Each *clause* should have the form

```
((datum ...) expr expr2 ...)
((datum ...) => proc)
```

where each *datum* is an external representation of some object. All the *datum*s must be distinct. The last *clause* may be an "else clause," which has the form

```
(else expr expr2 ...)
(else => proc)
```

First, *key* is evaluated and its result is compared against each *datum*. If the result of evaluating *key* is equivalent (using `eqv?`, see Section 6.2.1 [Equality], page 97), to a *datum*, then the expressions in the corresponding clause are evaluated sequentially, and the result(s) of the last expression in the *clause* is(are) returned from the case expression. The forms containing `=>` are specified in SRFI-87. In these forms, the result of *key* is passed to *proc*, and its result(s) is(are) returned from the case expression.

If the result of evaluating *key* is different from every *datum*, then if there is an else clause its expressions are evaluated and the result(s) of the last is(are) the result(s) of the case expression; otherwise the result of the case expression is undefined.

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ⇒ composite

(case (car '(c d))
  ((a) 'a)
  ((b) 'b)) ⇒ undefined

(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else 'consonant)) ⇒ consonant

(case 6
  ((2 4 6 8) => (cut + <> 1))
  (else => (cut - <> 1))) ⇒ 7

(case 5
  ((2 4 6 8) => (cut + <> 1))
  (else => (cut - <> 1))) ⇒ 4
```

`ecase` *key clause1 clause2 . . .* [Macro]

This works exactly like `case`, except when there's no `else` clause and the value of *key* expression doesn't match any of *datum*s provided in *clause*s. While `case` form returns undefined value for such case, *ecase* raises an error.

It is taken from Common Lisp. It's a convenient form when you want to detect when unexpected value is passed just in case.

```
(ecase 5 ((1) 'a) ((2 3) 'b) ((4) 'c))
 ⇒ ERROR: ecase test fell through: got 5, expecting one of (1 2 3 4)
```

`and` *test . . .* [Special Form]

[R7RS base] The *test* expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then `#t` is returned.

```
(and (= 2 2) (> 2 1)) ⇒ #t
(and (= 2 2) (< 2 1)) ⇒ #f
(and 1 2 'c '(f g))   ⇒ (f g)
(and)                 ⇒ #t
```

`or` *test . . .* [Special Form]

[R7RS base] The *test* expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then `#f` is returned.

```
(or (= 2 2) (> 2 1)) ⇒ #t
(or (= 2 2) (< 2 1)) ⇒ #t
(or #f #f #f)        ⇒ #f
(or (memq 'b '(a b c))
    (/ 3 0)) ⇒ (b c)
```

`when` *test expr1 expr2 . . .* [Special Form]

`unless` *test expr1 expr2 . . .* [Special Form]

[R7RS base] Evaluates *test*. If it yields true value (or false in case of `unless`), *expr1* and *expr2 . . .* are evaluated sequentially, and the result(s) of the last evaluation is(are) returned. Otherwise, undefined value is returned.

`assume` *test-expr message . . .* [Macro]

[SRFI-145] This declares the programmer's intent that the code following this path always satisfy *test-expr*.

Currently, Gauche evaluates *test-expr*, and if it evaluates to false, an error is signalled.

```
(define (real-sqrt x)
  (assume (and (real? x) (>= x 0)))
  (sqrt x))

gosh> (real-sqrt -1)
*** ERROR: Invalid assumption: (and (real? x) (>= x 0))
```

Note: This form is advisory—*test-expr* isn't guaranteed to be evaluated, nor it isn't guaranteed for an error is signaled when *test-expr* fails. For example, we may add an optimization option that omits testing in speed-optimized code in future. We may also enhance the compiler to generate better code using the given information—for example, in the above `real-sqrt` code, the compiler could theoretically deduce that `(sqrt x)` only needs to work as real functions, so it would be able to generate specialized code. Use this form to inform the compiler and the reader your intention.

`assume-type` *expr type* [Macro]

Check if the value of expression *expr* has type *type*. If not, raises an error.

```
(assume-type expr type) ≡ (assume (is-a? expr type))
```

Note: Like `assume`, this form is advisory; it is not guaranteed that the check is performed, nor *expr* is evaluated.

On the other hand, the type assumption may be used by the compiler future compilers for optimizations.

4.6 Binding constructs

`let` ((*var expr*) . . .) *body* . . . [Special Form]
`let*` ((*var expr*) . . .) *body* . . . [Special Form]
`letrec` ((*var expr*) . . .) *body* . . . [Special Form]
`letrec*` ((*var expr*) . . .) *body* . . . [Special Form]

[R7RS base] Creates a local scope where *var* . . . are bound to the value of *expr* . . ., then evaluates *body* . . . . *var*s must be symbols, and there shouldn't be a duplication. The value(s) of the last expression of *body* . . . becomes the value(s) of this form.

The four forms differ in terms of the scope and the order *expr*s are evaluated. `Let` evaluates *expr*s before (outside of) `let` form. The order of evaluation of *expr*s is undefined, and the compiler may reorder those *expr*s freely for optimization. `Let*` evaluates *expr*s, in the order they appears, and each *expr* is evaluated in the scope where *var*s before it are bound.

`Letrec` evaluates *expr*s, in an undefined order, in the environment where *var*s are already bound (to an undefined value, initially). `letrec` is necessary to define mutually recursive local procedures. Finally, `letrec*` uses the same scope rule as `letrec`, and it evaluates *expr* in the order of appearance.

```
(define x 'top-x)

(let  ((x 3) (y x)) (cons x y)) ⇒ (3 . top-x)
(let* ((x 3) (y x)) (cons x y)) ⇒ (3 . 3)

(let ((cons (lambda (a b) (+ a b)))
      (list (lambda (a b) (cons a (cons b 0)))))
  (list 1 2))  ⇒ (1 2 . 0)

(letrec ((cons (lambda (a b) (+ a b)))
         (list (lambda (a b) (cons a (cons b 0)))))
  (list 1 2))  ⇒ 3
```

You need to use `letrec*` if evaluation of one *expr* requires the value of *var* that appears before the *expr*. In the following example, calculating the value of *a* and *b* requires the value of *cube*, so you need `letrec*`. (Note the difference from the above example, where *calculating* the value of *list* doesn't need to take the value of *cons* bound in the same `letrec`. The value of *cons* isn't required until *list* is actually applied.)

```
(letrec* ((cube (lambda (x) (* x x x)))
          (a (+ (cube 1) (cube 12)))
          (b (+ (cube 9) (cube 10))))
  (= a b)) ⇒ #t
```

This example happens to work with `letrec` in the current Gauche, but it is not guaranteed to keep working in future. You just should not rely on evaluation order when you use `letrec`. In retrospect, it would be a lot simpler if we only have `letrec*`. Unfortunately `letrec` preceded for long time in Scheme history and it's hard to remove that. Besides, `letrec` does have more opportunities to optimize than `letrec*`.

`let1` *var expr body* . . . [Macro]

A convenient macro when you have only one variable. Expanded as follows.

```
(let ((var expr)) body ...)
```

`if-let1` *var expr then* [Macro]
`if-let1` *var expr then else* [Macro]

This macro simplifies the following idiom:

```
(let1 var expr
  (if var then else))
```

`rlet1` *var expr body . . .* [Macro]

This macro simplifies the following idiom:

```
(let1 var expr
  body ...
  var)
```

`and-let*` *(binding . . .) body . . .* [Macro]

[SRFI-2] In short, it works like `let*`, but returns `#f` immediately whenever the expression in *binding*s evaluates to `#f`.

Each *binding* should be one of the following form:

`(variable expression)`
: The *expression* is evaluated; if it yields true value, the value is bound to *variable*, then proceed to the next binding. If no more bindings, evaluates *body* . . . . If *expression* yields `#f`, stops evaluation and returns `#f` from `and-let*`.

`(expressionx)`
: In this form, *variable* is omitted. *Expression* is evaluated and the result is used just to determine whether we continue or stop further evaluation.

`bound-variable`
: In this form, *bound-variable* should be an identifier denoting a bound variable. If its value is not `#f`, we continue the evaluation of the clauses.

Let's see some examples. The following code searches *key* from an assoc-list *alist* and returns its value if found.

```
(and-let* ((entry (assoc key alist))) (cdr entry))
```

If *arg* is a string representation of an exact integer, returns its value; otherwise, returns 0:

```
(or (and-let* ((num (string->number arg))
               ( (exact? num) )
               ( (integer? num) ))
      num)
    0)
```

The following is a hypothetical code that searches a certain server port number from a few possibilities (environment variable, configuration file, ...)

```
(or (and-let* ((val (sys-getenv "SERVER_PORT")))
      (string->number val))
    (and-let* ((portfile (expand-path "~/.server_port"))
               ( (file-exists? portfile) )
               (val (call-with-input-string portfile port->string)))
      (string->number val))
    8080) ; default
```

`and-let1` *var test exp1 exp2 . . .* [Macro]

Evaluates *test*, and if it isn't `#f`, binds *var* to it and evaluates *exp1 exp2* . . . . Returns the result(s) of the last expression. If *test* evaluates to `#f`, returns `#f`.

This can be easily written by `and-let*` or `if-let1` as follows. However, we've written this idiom so many times that it deserves another macro.

```
(and-let1 var test
  exp1
  exp2 ...)

≡

(and-let* ([var test])
  exp1
  exp2 ...)

≡

(if-let1 var test
  (begin exp1 exp2 ...)
  #f)
```

`fluid-let` ((*var val*) . . .) *body* . . . [Macro]

A macro that emulates dynamic scoped variables. *Vars* must be variables bound in the scope including `fluid-let` form. *Vals* are expressions. `Fluid-let` first evaluates *vals*, then evaluates *body* . . ., with binding *vars* to the corresponding values during the dynamic scope of *body* . . . .

Note that, in multithreaded environment, the change of the value of *vars* are visible from all the threads. This form is provided mainly for the porting convenience. Use parameter objects instead (see Section 9.22 [Parameters], page 411) for thread-local dynamic state.

```
(define x 0)

(define (print-x) (print x))

(fluid-let ((x 1))
  (print-x))  ⇒ ;; prints 1
```

`receive` *formals expression body* . . . [Special Form]

[SRFI-8] This is the way to receive multiple values. *Formals* can be a (maybe-improper) list of symbols. *Expression* is evaluated, and the returned value(s) are bound to *formals* like the binding of lambda formals, then *body* . . . are evaluated.

```
(define (divrem n m)
  (values (quotient n m) (remainder n m)))

(receive (q r) (divrem 13 4) (list q r))
  ⇒ (3 1)

(receive all (divrem 13 4) all)
  ⇒ (3 1)

(receive (q . rest) (divrem 13 4) (list q rest))
  ⇒ (3 (1))
```

See also `call-with-values` in Section 6.18.8 [Multiple values], page 197, which is the procedural equivalent of `receive`. You can use `define-values` (see Section 4.10 [Definitions], page 61) to bind multiple values to variables simultaneously. Also `let-values` and `let*-values` below provides `let`-like syntax with multiple values.

`let-values` ((*vars expr*) ...) *body* ... [Macro]

[R7RS base] *vars* are a list of variables. *expr* is evaluated, and its first return value is bound to the first variable in *vars*, its second return value to the second variable, and so on, then *body* is evaluated. The scope of *expr*s are the outside of `let-values` form, like `let`.

```
(let-values (((a b) (values 1 2))
             ((c d) (values 3 4)))
  (list a b c d)) ⇒ (1 2 3 4)

(let ((a 1) (b 2) (c 3) (d 4))
  (let-values (((a b) (values c d))
               ((c d) (values a b)))
    (list a b c d))) ⇒ (3 4 1 2)
```

vars can be a dotted list or a single symbol, like the lambda parameters.

```
(let-values (((x . y) (values 1 2 3 4)))
  y) ⇒ (2 3 4)

(let-values ((x (values 1 2 3 4)))
  x) ⇒ (1 2 3 4)
```

If the number of values returned by *expr* doesn't match what *vars* expects, an error is signaled.

`let*-values` ((*vars expr*) ...) *body* ... [Macro]

[R7RS base] Same as `let-values`, but each *expr*'s scope includes the preceding *vars*.

```
(let ((a 1) (b 2) (c 3) (d 4))
  (let*-values (((a b) (values c d))
                ((c d) (values a b)))
    (list a b c d))) ⇒ (3 4 3 4)
```

`rec` *var expr* [Macro]
`rec` (*name . vars*) *expr* ... [Macro]

[SRFI-31] A macro to evaluate an expression with recursive reference.

In the first form, evaluates expr while *var* in *expr* is bound to the result of *expr*. The second form is equivalent to the followings.

```
(rec name (lambda vars expr ...))
```

Some examples:

```
;; constant infinite stream
(rec s (cons 1 (delay s)))

;; factorial function
(rec (f n)
  (if (zero? n)
      1
      (* n (f (- n 1)))))
```

4.7 Sequencing

`begin` *form* ... [Special Form]

[R7RS base] Evaluates *form*s sequentially, and returns the last result(s).

`Begin` doesn't introduce new scope like `let`, that is, you can't place "internal define" at the beginning of *form*s generally. Semantically `begin` behaves as if *form*s are spliced into the surrounding context. For example, toplevel expression like the following is the same as two toplevel definitions:

```
(begin (define x 1) (define y 2))
```

Here's a trickier example:

```
(let ()
  (begin
    (define x 2)
    (begin
      (define y 3)
    ))
  (+ x y))

  ≡

(let ()
  (define x 2)
  (define y 3)
  (+ x y))
```

`begin0` *exp0 exp1* ... [Macro]

Evaluates *exp0*, *exp1*, ..., then returns the result(s) of *exp0*. The name is taken from MzScheme. This is called `prog1` in CommonLisp.

Unlike `begin`, this *does* creates a new scope, for the `begin0` form is expanded as follows.

```
(receive tmp exp0
  exp1 ...
  (apply values tmp))
```

4.8 Iteration

`do` ((*variable init* [*step*]) ...) (*test expr* ...) *body* ... [Special Form]

[R7RS base]

1. Evaluates *init* ... and binds *variable* ... to each result. The following steps are evaluated under the environment where *variables* are bound.
2. Evaluate *test*. If it yields true, evaluates *expr* ... and returns the result(s) of last *expr*.
3. Otherwise, evaluates *body* ... for side effects.
4. Then evaluates *step* ... and binds each result to a fresh *variable* ..., and repeat from the step 2.

The following example loops 10 times while accumulating each value of *i* to *j* and returns it.

```
(do ((i 0 (+ i 1))
     (j 0 (+ i j)))
    ((= i 10) j)
  (print j))
 ⇒ 45 ; also prints intermediate values of j
```

If *step* is omitted, the previous value of *variable* is carried over. When there's no *expr*, the non-false value returned by *test* becomes the value of the `do` expression.

Since `do` syntax uses many parentheses, some prefer using square brackets as well as parentheses to visually distinguish the groupings. A common way is to group each variable binding, and the test clause, by square brackets.

```
(do ([i 0 (+ i 1)]
```

```
     [j 0 (+ i j)])
    [(= i 10) j]
  (print j))
```

Note: Unlike Common Lisp (and "for loops" in many languages), *variable* is freshly bound for each iteration. The following example loops 5 times and creates a list of closures, each of which closes the variable *i*. When you call each closures, you can see that each of them closes different *i* at the time of the iteration they were created.

```
(define closures
  (do ([i 0 (+ i 1)]
       [c '() (cons (^[] i) c)])
      [(= i 5) (reverse c)]
    ))

((car closures))  ⇒ 0
((cadr closures)) ⇒ 1
```

`let` *name* ((*var init*) ...) *body* ... [Special Form]

[R7RS base] This variation of `let` is called "named let". It creates the following procedure and binds it to *name*, then calls it with *init* ....

```
(lambda (var ...) body ...)
```

This syntax itself isn't necessarily related to iteration. However, the whole point of named let is that the above lambda expression is within the scope of *name*—that is, you can call *name* recursively within *body*. Hence this is used very often to write a loop by recursion (thus, often the procedure is named *loop*, as in the following example.)

```
(let loop ([x 0] [y '()])
  (if (= x 10)
    y
    (loop (+ x 1) (cons x y))))
 ⇒ (9 8 7 6 5 4 3 2 1 0)
```

Of course you don't need to loop with a named let; you can call *name* in non-tail position, pass *name* to other higher-order procedure, etc. Named let exists since it captures a very common pattern of local recursive procedures. Some Schemers even prefer named let to `do`, for the better flexibility.

The following rewrite rule precisely explains the named let semantics. The tricky use of `letrec` in the expansion is to make *proc* visible from *body* ... but not from *init* ....

```
(let proc ((var init) ...) body ...)
 ≡
((letrec ((proc (lambda (var ...) body ...)))
   proc)
 init ...)
```

`dotimes` ([*variable*] *num-expr* [*result*]) *body* ... [Macro]

`dolist` ([*variable*] *list-expr* [*result*]) *body* ... [Macro]

Convenience loop syntaxes, imported from Common Lisp. They are not very Scheme-y, in a sense that these rely on some side-effects in *body* .... Nevertheless these capture some common pattern in day-to-day scripting.

You can use `dotimes` to repeat *body* ... for a number of times given by *num-expr*, and `dolist` to repeat *body* ... while traversing a list given by *list-expr*. While *body* ... is evaluated, *variable* is bound to the current iteration count (in `dotimes`), or the current element in the list (in `dolist`).

```
(dotimes (n 5) (write n))
```

```
 ⇒ writes "01234"

(dolist (v '(a b c d e)) (write v))
 ⇒ writes "abcde"
```

If you don't need to refer to *variable*, you can omit it. For example, the following example prints `year!` 10 times:

```
(dotimes (10) (print "yeah!"))
```

If the third element (*result*) is given in `dotimes` or `dolist`, it is evaluated after all repetition is done, and its result becomes the result of `dotimes`/`dolist`. While *result* is evaluated, *variable* is bound to the number of repetitions (in `dotimes`) or `()` (in `dolist`). It is supported because Common Lisp has it.

Note that a fresh *variable* is bound for each iteration, as opposed to Common Lisp where *variable* is mutated. So if you create a closure closing *variable*, it won't be overwritten by the subsequent iteration.

If you need more than simple iteration, you can use `do` form, named `let`, or Section 11.12 [Eager comprehensions], page 610, which provides rich way to iterate.

`while` *expr body* ... [Macro]
`while` *expr* `=>` *var body* ... [Macro]
`while` *expr guard* `=>` *var body* ... [Macro]

Var is an identifier and *guard* is a procedure that takes one argument.

In the first form, *expr* is evaluated, and if it yields a true value, *body* ... are evaluated. It is repeated while *expr* yields true value.

In the second form, *var* is bound to a result of *expr* in the scope of *body* ....

In the third form, the value *expr* yields are passed to *guard*, and the execution of *body* ... is repeated while *guard* returns a true value. *var* is bound to the result of *expr*.

The return value of `while` form itself isn't specified.

```
(let ((a '(0 1 2 3 4)))
  (while (pair? a)
    (write (pop! a)))) ⇒ prints "01234"

(let ((a '(0 1 2 3 #f 5 6)))
  (while (pop! a) integer? => var
    (write var))) ⇒ prints "0123"
```

`until` *expr body* ... [Macro]
`until` *expr guard* `=>` *var body* ... [Macro]

Like `while`, but the condition is reversed. That is, the first form repeats evaluation of *expr* and *body* ... until *expr* yields true. In the second form, the result of *expr* is passed to *guard*, and the execution is repeated until it returns true. *Var* is bound to the result of *expr*.

(The second form without *guard* isn't useful in `until`, since *var* would always be bound to `#f`).

The return value of `until` form itself isn't specified.

```
(let ((a '(0 1 2 3 4)))
  (until (null? a)
    (write (pop! a)))) ⇒ prints "01234"

(until (read-char) eof-object? => ch
  (write-char ch))
 ⇒ reads from stdin and writes char until EOF is read
```

4.9 Quasiquotation

`quasiquote` *template* [Special Form]

[R7RS base] Quasiquotation is a convenient way to build a structure that has some fixed parts and some variable parts. See the explanation below.

`‘template` [Reader Syntax]

[R7RS] The syntax `‘x` is read as `(quasiquote x)`.

`unquote` *datum ...* [Special Form]
`unquote-splicing` *datum ...* [Special Form]

[R7RS base] These syntaxes have meaning only when they appear in the *template* of quasiquoted form. R5RS says nothing about these syntaxes appear outside of quasiquote. Gauche signals an error in such case, for it usually indicates you forget quasiquote somewhere.

R5RS only allows `unquote` and `unquote-splicing` to take a single argument; it is undefined if you have `(unquote)` or `(unquote x y)` inside quasiquoted form. R6RS allows zero or multi-arguments, and Gauche follows that.

`,datum` [Reader Syntax]
`,@datum` [Reader Syntax]

[R7RS] The syntaxes `,x` and `,@x` are read as `(unquote x)` and `(unquote-splicing x)`, respectively.

Quasiquote basics

Suppose you want to create a list `(foo bar x y)`, where `foo` and `bar` are symbols, and *x* and *y* are the value determined at runtime. (For the sake of explanation, let’s assume we have variables *x* and *y* that provides those values.) One way to do that is to call the function `list` explicitly.

```
(let ((x 0) (y 1))
  (list 'foo 'bar x y)) ⇒ (foo bar 0 1)
```

You can do the same thing with quasiquote, like this:

```
(let ((x 0) (y 1))
  `(foo bar ,x ,y))  ⇒ (foo bar 0 1)
```

The difference between the two notations is that the explicit version quotes the parts that you want to insert literally into the result, while the quasiquote version *unquotes* the parts that you don’t want to quote.

The quasiquote version gets simpler and more readable when you have lots of static parts with scattered variable parts in your structure.

That’s why quasiquote is frequently used with legacy macros, which are basically a procedure that create program fragments from variable parts provided as macro arguments. See the simple-minded `my-if` macro that expands to `cond` form:

```
(define-macro (my-if test then else)
  `(cond (,test ,then)
         (else ,else)))

(macroexpand '(my-if (< n 0) n (- n)))
  ⇒ (cond ((< n 0) n) (else (- n)))
```

Note the two `else`s in the macro definition; one isn’t unquoted, thus appears literally in the output, while another is unquoted and the corresponding macro argument is inserted in its place.

Of course you can use quasiquotes unrelated to macros. It is a general way to construct structures. Some even prefer using quasiquote to explicit construction even most of the structure is variable, for quasiquoted form can be more concise. Gauche also tries to minimize runtime allocation for quasiquoted forms, so it may potentially be more efficient; see "How static are quasiquoted forms?" below.

Splicing

When `(unquote-splicing expr)` appears in a quasiquoted form, *expr* must evaluate to a list, which is *spliced* into the surrounding context. It's easier to see examples:

```
(let ((x '(1 2 3)))
  `(a ,@x b)) ⇒ (a 1 2 3 b)

(let ((x '(1 2 3)))
  `(a ,x b)) ⇒ (a (1 2 3) b)

(let ((x '(1 2 3)))
  `#(a ,@x b)) ⇒ #(a 1 2 3 b)
```

Compare the unquote version and unquote-splicing version. Splicing also works within a vector.

Multi-argument unquotes

If `unquote` or `unquote-splicing` takes multiple arguments, they are interpreted as if each of its arguments are unquoted or unquote-spliced.

```
;; This is the same result as `(,(+ 1 2) ,(+ 2 3) ,(+ 3 4))
`((unquote (+ 1 2) (+ 2 3) (+ 3 4)))
  ⇒ (3 5 7)

;; This is the same result as
;;   `(,@(list 1 2) ,@(list 2 3) ,@(list 3 4))
`((unquote-splicing (list 1 2) (list 2 3) (list 3 4)))
  ⇒ (1 2 2 3 3 4)

;; Edge cases
`((unquote))          ⇒ ()
`((unquote-splicing)) ⇒ ()
```

It is an error for zero or multiple argument `unquote`/`unquote-splicing` forms appear which you cannot splice multiple forms into.

```
;; Multiple arguments unquotes are error in non-splicing context
`(unquote 1 2)          ⇒ error
`(unquote-splicing 1 2) ⇒ error
```

Note that the abbreviated notations `,x` and `,@x` are only for single-argument forms. You have to write `unquote` or `unquote-splicing` explicitly for zero or multiple argument forms; thus you don't usually need to use them. These forms are supported mainly to make the nested unquoting forms such as `,,@` and `,@,@`—R5RS cannot handle the case the inner unquote-splicing form expands into zero or multiple forms.

How static are quasiquoted forms?

When quasiquoted form contains variable parts, what happens at runtime is just the same as when an explicit form is used: `` `(,x ,y) `` is evaluated exactly like `(list x y)`. However, Gauche tries to minimize runtime allocation when a quasiquoted form has static parts.

First of all, if there's no variable parts in quasiquoted form, like `` `(a b c) ``, the entire form is allocated statically. If there is a static tail in the structure, it is also allocated statically; e.g. `` `((,x a b) (,y c d)) `` works like `(list (cons x '(a b)) (cons y '(c d)))`.

Furthermore, when an unquoted expression is a constant expression, Gauche embeds it into the static form. If you've defined a constant like `(define-constant x 3)`, then the form `` `(,x ``

`,(+ x 1))` is compiled as the constant `'(3 4)`. (See Section 4.10 [Definitions], page 61, for the explanation of `define-constant` form.)

In general it is hard to say which part of quasiquoted form is compiled as a static datum and which part is not, so you shouldn't write a code that assumes some parts of the structure returned from quasiquote are freshly allocated. In other words, you better avoid mutating such structures.

4.10 Definitions

`define` *variable expression* [Special Form]
`define` (*variable* . *formals*) *body* . . . [Special Form]
`define` *variable* [Special Form]

[R7RS+] This form has different meanings in the toplevel (without no local bindings) or inside a local scope.

On toplevel, it defines a global binding to a symbol *variable*. In the first form, it globally binds a symbol *variable* to the value of *expression*, in the current module.

```
(define x (+ 1 2))
x ⇒ 3
(define y (lambda (a) (* a 2)))
(y 8) ⇒ 16
```

If *variable* is already bound *in the same module*, the subsequent definitions work just like assignments.

```
(define x 3)
(define (value-of-x) x)

(value-of-x x) ⇒ 3

(define x 4)

(value-of-x x) ⇒ 4
```

If *variable* is not bound in the current module, but has an imported bindings, things get interesting but complicated. See Section 4.10.1 [Into the Scheme-Verse], page 65, for the details.

The second form is a syntactic sugar of defining a procedure. It is equivalent to the following form.

```
(define (name . args) body ...)
  ≡ (define name (lambda args body ...))
```

The third form is a shorthand of `(define variable (undefined))`. It is introduced in R6RS (but not a part of R7RS). You can use that form to indicate the initial value doesn't matter.

If the form appears inside a local scope (internal define), this introduce a local binding of the variable.

Internal defines can appear in the beginning of body of `lambda` or other forms that introduces local bindings. They are equivalent to a `letrec*` form, as shown below.

```
(lambda (a b)
  (define (cube x) (* x x x))
  (define (square x) (* x x))
  (+ (cube a) (square b)))

  ≡
```

```
(lambda (a b)
  (letrec* ([cube (lambda (x) (* x x x))]
            [square (lambda (x) (* x x))])
    (+ (cube a) (square b))))
```

Since internal defines are essentially a `letrec*` form, you can write mutually recursive local functions, and you can use preceding bindings introduced in the same scope to calculate the value to be defined. However, you can't use a binding that is introduced after an internal define form to calculate its value; if you do so, Gauche may not report an error immediately, but you may get strange errors later on.

```
(lambda (a)
  (define x (* a 2))
  (define y (+ x 1))  ; ok to use x to calculate y
  (* a y))

(lambda (a)
  ;; You can refer to even? in odd?, since the value of even?
  ;; isn't used at the time odd? is defined; it is only used
  ;; when odd? is called.
  (define (odd? x) (or (= x 1) (not (even? (- x 1)))))
  (define (even? x) (or (= x 0) (not (odd? (- x 1)))))
  (odd? a))

(lambda (a)
  ;; This is not ok, for defining y needs to use the value
  ;; of x.  However, you may not get an error immediately.
  (define y (+ x 1))
  (define x (* a 2))
  (* a y))
```

Inside the body of binding constructs, internal defines must appear before any expression of the same level. The following code isn't allowed, for an expression `(print a)` precedes the `define` form.

```
(lambda (a)
  (print a)
  (define (cube x) (* x x x))  ; error!
  (cube a))
```

It is also invalid to put no expressions but internal defines inside the body of binding constructs, although Gauche don't report an error.

Note that `begin` (see Section 4.7 [Sequencing], page 55) doesn't introduce a new scope. `Define`s in the `begin` act as if `begin` and surrounding parenthesis are not there. Thus these two forms are equivalent.

```
(let ((x 0))
  (begin
    (define (foo y) (+ x y)))
  (foo 3))
 ≡
(let ((x 0))
  (define (foo y) (+ x y))
  (foo 3))
```

define-values (*var ...*) *expr* [Macro]
define-values (*var var1 ... . var2*) *expr* [Macro]
define-values *var expr* [Macro]

[R7RS base] *Expr* is evaluated, and each value of the result is bound to each *vars*. In the first form, it is an error unless *expr* yields the same number of values as *vars*.

```
(define-values (lo hi) (min&max 3 -1 15 2))

lo ⇒ -1
hi ⇒ 15
```

In the second form, *expr* may yield as many values as *var var1 ...* or more; the excess values are made into a list and bound to *var2*.

```
(define-values (a b . c) (values 1 2 3 4))

a ⇒ 1
b ⇒ 2
c ⇒ (3 4)
```

In the last form, all the values yielded by *expr* are gathered to a list and bound to *var*.

```
(define-values qr (quotient&remainder 23 5))

qr ⇒ (4 3)
```

You can use `define-values` wherever `define` is allowed; that is, you can mix `define-values` in internal defines.

```
(define (foo . args)
  (define-values (lo hi) (apply min&max args))
  (define len (length args))
  (list len lo hi))

(foo 1 4 9 3 0 7)
 ⇒ (6 0 9)
```

define-constant *variable expression* [Special Form]
define-constant (*variable . formals*) *body ...* [Special Form]

This form is only effective in toplevel.

Like top-level `define`, it defines a top-level definition of *variable* with the value of *expression*, but additionally tells the compiler that (1) the binding won't change, and (2) the value of *expression* won't change from the one computed at the compile time. So the compiler can replace references of *variable* with the compile-time value of *expression*.

An error is signaled when you use `set!` to change the value of *variable*. It is allowed to redefine *variable*, but a warning is printed.

The difference from `define-inline` below is that the value of *expression* is computed at the compile time and treated as a literal. Suppose you define *x* as follows:

```
(define-constant x (vector 1 2 3))
```

Then, the code `(list x)` is compiled to the same code as `(list '#(1 2 3))`.

This distinction is especially important when you do AOT (ahead of time) compilation.

There's no "internal `define-constant`", since the compiler can figure out whether a local binding is mutated, and optimize code accordingly, without a help of declarations.

define-inline *variable expression* [Special Form]
define-inline (*variable . formals*) *body ...* [Special Form]

The second form is a shorthand of `(define-inline variable (lambda formals body ...))`.

If this appears in the position of internal defines, it is the same as internal defines.

If it appears in the toplevel, it defines an *inlinable* binding. An inlinable binding promises the compiler that the binding won't change, but unlike constant bindings introduced by `define-constant`, the actual value of *expression* may be computed at runtime. Hence the compiler cannot simply replace the references of *variable* with the compile-time value of *expression*.

However, if the compiler can determine that the value of *expression* is to be a procedure, it may inline the procedure where it is invoked.

In the example below, the body of `dot3` is inlined where `dot3` is called. Furthermore, since the second argument of `dot3` is a constant vector, you can see `vector-ref` on it is computed at compile time (e.g. `CONST -1.0` etc.)

```
gosh> (define-inline (dot3 a b)
        (+ (* (vector-ref a 0) (vector-ref b 0))
           (* (vector-ref a 1) (vector-ref b 1))
           (* (vector-ref a 2) (vector-ref b 2))))
dot3
gosh> (disasm (^[] (dot3 x '#(-1.0 -2.0 -3.0))))
CLOSURE #<closure (#f)>
=== main_code (name=#f, code=0x28524e0, size=26, const=4 stack=6):
signatureInfo: ((#f))
     0 GREF-PUSH #<identifier user#x.20d38e0>; x
     2 LOCAL-ENV(1)             ; (dot3 x (quote #(-1.0 -2.0 -3.0)))
     3 LREF0                    ; a
     4 VEC-REFI(0)              ; (vector-ref a 0)
     5 PUSH
     6 CONST -1.0
     8 NUMMUL2                  ; (* (vector-ref a 0) (vector-ref b 0))
     9 PUSH
    10 LREF0                    ; a
    11 VEC-REFI(1)              ; (vector-ref a 1)
    12 PUSH
    13 CONST -2.0
    15 NUMMUL2                  ; (* (vector-ref a 1) (vector-ref b 1))
    16 NUMADD2                  ; (+ (* (vector-ref a 0) (vector-ref b 0))
    17 PUSH
    18 LREF0                    ; a
    19 VEC-REFI(2)              ; (vector-ref a 2)
    20 PUSH
    21 CONST -3.0
    23 NUMMUL2                  ; (* (vector-ref a 2) (vector-ref b 2))
    24 NUMADD2                  ; (+ (* (vector-ref a 0) (vector-ref b 0))
    25 RET
```

As an extreme case, if both arguments are compile-time constant, `dot3` is completely computed at compile time:

```
gosh> (disasm (^[] (dot3 '#(1 2 3) '#(4 5 6))))
CLOSURE #<closure (#f)>
=== main_code (name=#f, code=0x2a2b8e0, size=2, const=0 stack=0):
signatureInfo: ((#f))
     0 CONSTI(32)
     1 RET
```

The same inlining behavior may be achieved by making `dot3` a macro, but if you use `define-inline`, `dot3` can be used as procedures when needed:

```
(map dot3 list-of-vectors1 list-of-vectors2)
```

If `dot3` is a macro you can't pass it as a higher-order procedure.

The inline expansion pass is run top-to-bottom. Inlinable procedure must be defined before used in order to be inlined.

If you redefine an inlinable binding, Gauche warns you, since the redefinition won't affect already inlined call sites. So it should be used with care—either use it internal to the module, or use it for procedures that won't change in future. Inlining is effective for performance-critical parts. If a procedure is called sparingly, there's no point to define it inlinable.

`define-in-module` *module variable expression* [Special Form]
`define-in-module` *module* (*variable* . *formals*) *body* . . . [Special Form]

This form must appear in the toplevel. It creates a global binding of *variable* in *module*, which must be either a symbol of the module name or a module object. If *module* is a symbol, the named module must exist.

Expression is evaluated in the current module.

The second form is merely a syntactic sugar of:

```
(define-in-module module variable (lambda formals body ...))
```

Note: to find out if a symbol has definition (global binding) in the current module, you can use `global-variable-bound?` (see Section 4.13.6 [Module introspection], page 76).

4.10.1 Into the Scheme-Verse

Multiple toplevels are multiple scopes

One upon a time, the Scheme world was simple. We had one single global space we called the toplevel. Toplevel definitions can be understood as side-effects to this global space; if the name hasn't been exist there yet, create a new binding, otherwise, overwrite existing one.

The problem was that it was hard to scale, thus many implementations introduced their own module systems. One of the main agenda of R6RS was to have a module system (which is called "library" in RnRS) consistent with the design of Scheme. Especially, since Scheme's hygienic macro system captures lexical scope, it is desirable that it interacts with the module system in the same way.

In modern Scheme, "toplevel" of each module creates its own lexical scope, and the definitions are understood in `letrec*` semantics. Hence, macro systems can consistently treat identifiers as a name associated with a scope.

Suppose you see these toplevel definitions:

```
(define (odd? n)  (if (zero? x) #f (even? (- n 1))))
(define (even? n) (if (zero? x) #t (odd? (- n 1))))
```

The first appearance of `even?` in the first line is understood as the one defined in the second line. It becomes apparent when we compare it with internal defines:

```
(let ((even? error))
  (define (odd? n)  (if (zero? x) #f (even? (- n 1))))
  (define (even? n) (if (zero? x) #t (odd? (- n 1))))
  ...)
```

The `even?` in the definition of `odd?` refers to the one defined in the next line, never to the one bound by `let`.

So far, so good.

Now, consider the following toplevel code:

```
;; Invalid in RnRS, n >= 6
(import (scheme base) (scheme write))
(define orig-error error)
(define (error . args)
  (write args) (newline)
  (apply orig-error args))
```

The intention is to save the *original* value of `error`, which is imported from `(scheme base)`, into a variable `orig-error`, then redefine `error` to add logging feature. This technique was popular in pre-R6RS Scheme.

However, with our new toplevel-as-a-scope Scheme, the `error` in `(define orig-error error)` *must* refer to the one defined in the same scope, which is the new definition below; otherwise lexical scoping gets broken. The value of inner `error` hasn't been calculated when `orig-error`'s value is calculated, so the above form is an invalid program in terms of RnRS.

In fact, to avoid confusion, R6RS prohibits defining a toplevel variable that conflicts with the imported name (in R7RS the behavior of such program is undefined). In the example above, the name `error` is imported from `(scheme base)` and also defined in the toplevel, hence it's a violation.

The modern way of such augumentation is to use renaming import:

```
(import (except (scheme base) error)
        (rename (scheme base) (error r7rs:error))
        (scheme write))
(define (error . args)
  (write args) (newline)
  (apply r7rs:error args))
```

Gauche's take

Gauche's module system predates R6RS and R7RS, and it regards a module as a first-class entity and suppors class-like inheritance. It is upper-compatible to R7RS libraries, but we take freedom in interpreting R7RS undefined behaviors.

First, you can define toplevel variables that conflict with imported or inherited bindings. The new definition simply shadows the old one.

Second, if multiple toplevel forms are processed at once e.g. it is enclosed in `begin` or the file is read by `include`, we treat them in one scope. That is, if the above `orig-error` example is read by `include`, the first `error` refers to the to-be-defined `error` below. Since the value of `error` hasn't been calculated by the time it's used, you'll get the following error:

```
*** ERROR: uninitialized variable: error
```

Third, Gauche compiles and executes each individual toplevel forms (the forms that's not enclosed in other S-expressions). It is the same as REPL semantics. If each form of `orig-error` example appears individually on the toplevel, the `(define orig-error error)` line actually refers to the R7RS `error` and assign it to `orig-error`, since *we don't know yet if* `error` *will be defined in the same scope.*

The third rule is necessary to support REPL semantics, but note that the result would differ when the same file is `include`d. If you can, avoid writing such ambiguous code.

Note: The second behavior is clarified in release 0.9.9 for the better compatibility with R7RS. Before that, the behavior of such case is undefined, but some code might have expected that it works in REPL semantics (the third rule).

In order to support the transition, if you set an enviornment variable `GAUCHE_LEGACY_DEFINE`, Gauche treats definitions in the same way as 0.9.8 and before. Note that if you that, you may see Gauche can't include some valid R7RS code that has multiple libraries in one file.

4.11 Inclusions

`include` *filename* . . . [Special Form]
`include-ci` *filename* . . . [Special Form]

[R7RS base] Reads *filename* . . . at compile-time, and insert their contents as if the forms are placed in the includer's source file, surrounded by `begin`. The `include` form reads files as is, while `include-ci` reads files in case-insensitive way, as if `#!fold-case` is specified in the beginning of the file (see Section 2.4 [Case-sensitivity], page 12).

The coding magic comment in each file is honored while reading that file (see Section 2.3 [Multibyte scripts], page 11).

If *filename* is absolute, the file is just searched. If it is relative, the file is first searched relative to the file containing the `include` form, then the directories in `*load-path*` are tried.

Example: Suppose a file `a.scm` contains the following code:

```
(define x 0)
(define y 1)
```

You can include this file into another source, like this:

```
(define (foo)
  (include "a.scm")
  (list x y))
```

It works as if the source is written as follows:

```
(define (foo)
  (begin
   (define x 0)
   (define y 1))
  (list x y))
```

(Note: In version 0.9.4, `include` behaved differently when pathname begins with either `./` or `../`—in which case the file is searched relative to the current working directory of the compiler. It is rather an artifact of `include` sharing file search routine with `load`. But unlike `load`, which is a run-time operation, using relative path to the current directory won't make much sense for `include`, so we changed the behavior in 0.9.5.)

Gauche has other means to incorporate source code from another files. Here's the comparison.

`require` (`use` and `extend` calls `require` internally)

- Both `require` and `include` work at compile-time.
- `Require` works only in toplevel context, while `include` can be anywhere.
- `Require` reads the file only once (second and later `require` on the same file becomes no-op), while `include` reads the file every place it appears.
- The file is searched from `*load-path*`. The location of the file `require` form appears doesn't matter. (You can add directories relative to the requiring file using the `:relative` flag in `add-load-path`, though).
- Even if the current module is changed by `select-module` inside the required file, it is only effective while the required file is read. On the other hand, `include` inserts any S-expressions in the included file to the place `include` appears, so the effect of `select-module` persists after `include` form (Note: Encoding magic comment and `#!fold-case`/`#!no-fold-case` are dealt with by the reader, so those effect is contained in the file even with `include`).
- It is forbidden to the file loaded by `require` to insert a toplevel binding without specifying a module. In other words, the file you require should generally use `define-module`, `select-module` or `define-library`. See Section 6.23.3

[Require and provide], page 241, for further discussion. On the other hand, `include` has no such restrictions.

`load`

- Works at runtime, while `include` works at compile-time.
- Works only in toplevel context, while `include` can be anywhere.
- The file is searched from `*load-path*`, except when the file begins with `./` or `../`, in which case it is first tried relative to the current directory before being searched from `*load-path*`.
- As the case with `require`, change of the current module won't persist after `load`.

Usually, `require` (or `use` and `extend`) are better way to incorporate sources in other files. The `include` form is mainly for the tricks that can't be achieved with `require`. For example, you have a third-party R5RS code and you want to wrap it with Gauche module system. Using `include`, you place the following small source file along the third-party code, and you can load the code with `(use third-party-module)` without changing the original code at all.

```
(define-module third-party-module
  (export proc ...)
  (include "third-party-source.scm"))
```

4.12 Feature conditional

The `cond-expand` macro

Sometimes you need to have a different piece of code depending on available features provided by the implementation and/or platform. For example, you may want to switch behavior depending on whether networking is available, or to embed an implementation specific procedures in otherwise-portable code.

In C, you use preprocessor directives such as `#ifdef`. In Common Lisp, you use reader macro `#+` and `#-`. In Scheme, you have `cond-expand`:

`cond-expand` (*feature-requirement command-or-definition . . .*) *. . .* [Macro]

[R7RS base] This macro expands to *command-or-definition . . .* if *feature-requirement* is supported by the current platform.

feature-requirement must be in the following syntax:

```
feature-requirement
  : feature-identifier
  | (and feature-requirement ...)
  | (or  feature-requirement ...)
  | (not feature-requirement)
  | (library library-name)
```

The macro tests each *feature-requirement* in order, and if one is satisfied, the macro itself expands to the corresponding *command-or-definition . . .*.

The last clause may have `else` in the position of *feature-requirement*, to make the clause expanded if none of the previous feature requirement is fulfilled.

If there's neither a satisfying clause nor `else` clause, `cond-expand` form throws an error. It is to detect the case early that the platform doesn't have required features. If the feature you're testing is optional, that is, your program works without the feature as well, add empty `else` clause as follows.

```
(cond-expand
```

```
[feature expr]  ; some optional feature
[else])
```

feature-identifier is a symbol that indicates a feature. If such a feature is supported in the current platform, it satisfies the *feature-requirement*. You can do boolean combination of *feature-requirement*s to compose more complex conditions.

The form `(library` *`library-name`*`)` is added in R7RS, and it is fulfilled when the named library is available. Since this is R7RS construct, you have to use R7RS-style library name—list of symbols/integers, e.g. `(gauche net)` instead of `gauche.net`.

Here's a typical example: Suppose you want to have implementation-specific part for Gauche, Chicken Scheme and ChibiScheme. Most modern Scheme implementations defines a feature-identifier to identify itself. You can write the conditional part as follows:

```
(cond-expand
 [gauche  (gauche-specific-code)]
 [(or chicken chibi) (chicken-chibi-specific-code)]
 [else    (fallback-code)]
 )
```

It is important that the conditions of `cond-expand` is purely examined at the macro-expansion time, and unfulfilled clauses are discarded. Thus, for example, you can include macro calls or language extensions that may not be recognized on some implementations. You can also conditionally define global bindings.

Compare that to `cond`, which examines conditions at runtime. If you include unsupported macro call in one of the conditions, it may raise an error at macro expansion time, even if that clause will never be executed on the platform. Also, it is not possible to conditionally define global bindings using `cond`.

There's a caveat, though. Suppose you want to save the result of macro expansion, and run the expanded result later on other platforms. The result code is based on the features of the platform the macro expansion takes place, which may not agree with the features of the platform the code will run. (This issue always arises in cross-compiling situation in general.)

See below for the list of feature identifiers defined in Gauche.

Gauche-specific feature identifiers

`gauche`
`gauche-X.X.X`
: Indicates you're running on Gauche. It is useful to put Gauche-specific code in a portable program. `X.X.X` is the gauche's version (e.g. `gauche-0.9.4`), in case you want to have code for specific Gauche version. (Such feature identifier is suggested by R7RS; but it might not be useful if we don't have means to compare versions. Something to consider in future versions.)

`gauche.os.windows`
`gauche.os.cygwin`
: Defined on Windows-native platform and Cygwin/Windows platform, respectively. If neither is defined you can assume it's a unix variant. (Cygwin is supposedly unix variant, but corners are different enough to deserve it's own feature identifier.)

`gauche.ces.utf8`
`gauche.ces.eucjp`
`gauche.ces.sjis`
`gauche.ces.none`
: Either one of these is defined based on Gauche's native character encoding scheme. See Section 2.2 [Multibyte strings], page 11, for the details.

`gauche.net.tls`
`gauche.net.tls.axtls`
`gauche.net.tls.mbedtls`

Defined if the runtime supports TLS in networking. The two sub feature identifiers, `gauche.net.tls.axtls` and `gauche.net.tls.mbedtls`, are defined if each subsystem `axTLS` and `mbedTLS` is supported, respectively.

`gauche.net.ipv6`

Defined if the runtime supports IPv6. Note that this only indicates Gauche has been built with IPv6 support; the OS may not allow IPv6 features, in that case you'll get system error when you try to use IPv6.

`gauche.sys.threads`
`gauche.sys.pthreads`
`gauche.sys.wthreads`

If the runtime supports multithreading, `gauche.sys.threads` is defined (see Section 9.33 [Threads], page 457). Multithreading is based on either POSIX pthreads or Windows threads. The former defines `gauche.sys.pthreads`, and the latter defines `gauche.sys.wthreads`.

`gauche.sys.sigwait`
`gauche.sys.setenv`
`gauche.sys.unsetenv`
`gauche.sys.clearenv`
`gauche.sys.getloadavg`
`gauche.sys.getrlimit`
`gauche.sys.lchown`
`gauche.sys.getpgid`
`gauche.sys.nanosleep`
`gauche.sys.crypt`
`gauche.sys.symlink`
`gauche.sys.readlink`
`gauche.sys.select`
`gauche.sys.fcntl`
`gauche.sys.syslog`
`gauche.sys.setlogmask`
`gauche.sys.openpty`
`gauche.sys.forkpty`

Those are defined based on the availability of these system features of the platform.

R7RS feature identifiers

`r7rs` Indicates the implementation complies r7rs.

`exact-closed`

Exact arithmetic operations are closed; that is, dividing an exact number by a non-zero exact number always yields an exact number.

`ieee-float`

Using IEEE floating-point number internally.

`full-unicode`

Full unicode support.

`ratios` Rational number support

`posix`
`windows` Either one is defined, according to the platform.

`big-endian`
`little-endian`
    Either one is defined, according to the platform.

4.13 Modules

This section describes the semantics of Gauche modules and its API. See also Section 3.7 [Writing Gauche modules], page 32, for the conventions Gauche is using for its modules.

For R7RS programs, they are called "libraries" and have different syntax than Gauche modules. See Section 10.2.1 [R7RS library form], page 503, for the details.

4.13.1 Module semantics

Module is an object that maps symbols onto *bindings*, and affects the resolution of global variable reference.

Unlike CommonLisp's packages, which map names to symbols, in Gauche symbols are `eq?` in principle if two have the same name (except uninterned symbols; see Section 6.7 [Symbols], page 137). However, Gauche's symbol doesn't have a 'value' slot in it. From a given symbol, a module finds its binding that keeps a value. Different modules can associate different bindings to the same symbol, that yield different values.

```
;; Makes two modules A and B, and defines a global variable 'x' in them
(define-module A (define x 3))
(define-module B (define x 4))

;;  #<symbol 'x'> ---[module A]--> #<binding that has 3>
(with-module A x) ⇒ 3

;;  #<symbol 'x'> ---[module B]--> #<binding that has 4>
(with-module B x) ⇒ 4
```

A module can *export* a part or all of its bindings for other module to use. A module can *import* other modules, and their exported bindings become visible to the module. A module can import any number of modules.

```
(define-module A
  (export pi)
  (define pi 3.1416))

(define-module B
  (export e)
  (define e 2.71828))

(define-module C
  (import A B))

(select-module C)
(* pi e) ⇒ 8.539748448
```

A module can also be *inherited*, that is, you can extend the existing module by inheriting it and adding new bindings and exports. From the new module, all ancestor's bindings (including non-exported bindings) are visible. (A new module inherits the `gauche` module by default, which is why the built-in procedures and syntax of `gauche` are available in the new module). From outside, the new module looks like having all exported bindings of the original module plus the newly defined and exported bindings.

```
;; Module A defines and exports deg->rad.
```

```
;; A binding of pi is not exported.
(define-module A
  (export deg->rad)
  (define pi 3.1416)   ;; not exported
  (define (deg->rad deg) (* deg (/ pi 180))))

;; Module Aprime defines and exports rad->deg.
;; The binding of pi is visible from inside Aprime.
(define-module Aprime
  (extend A)
  (export rad->deg)
  (define (rad->deg rad) (* rad (/ 180 pi))))

;; Module C imports Aprime.
(define-module C
  (import Aprime)
  ;; Here, both deg->rad and rad->deg are visible,
  ;; but pi is not visible.
  )
```

At any moment of the compilation, there is one "current module" available, and the global variable reference is looked for from the module. If there is a visible binding of the variable, the variable reference is compiled to the access of the binding. If the compiler can't find a visible binding, it marks the variable reference with the current module, and delays the resolution of binding at the time the variable is actually used. That is, when the variable is referenced at run time, the binding is again looked for from the marked module (*not* the current module at the run time) and if found, the variable reference code is replaced for the the code to access the binding. If the variable reference is not found even at run time, an 'undefined variable' error is signaled.

Once the appropriate binding is found for the global variable, the access to the binding is hard-wired in the compiled code and the global variable resolution will never take place again.

The definition special form such as `define` and `define-syntax` inserts the binding to the current module. Thus it may shadow the binding of imported or inherited modules.

The resolution of binding of a global variable happens like this. First, the current module is searched. Then, each imported module is taken in the reverse order of import, and searched, including each module's ancestors. Note that import is not transitive; imported module list is not chased recursively. Finally, ancestors of the current module are searched in order.

This order is important when more than one modules defines the same name and your module imports both. Assuming your module don't define that name, if you first import a module `A` then a module `B`, you'll see `B`'s binding.

If you import `A`, then `B`, then `A` again, the last import takes precedence; that is, you'll see `A`'s binding.

If two modules you want to use exports bindings of the same name and you want to access both, you can add prefix to either one (or both). See Section 4.13.4 [Using modules], page 73, for the details.

4.13.2 Modules and libraries

Modules are run-time data structure; you can procedurally create modules with arbitrary names at run-time.

However, most libraries use modules to create their own namespace, so that they can control which bindings to be visible from library users. (This "library" is a general term, broader than R7RS "library").

Usually a library is provided in the form of one or more Scheme source file(s), so it is convenient to have a convention to map module names to file names, and vice versa; then, you can load a library file and import its module by one action with `use` macro, for example.

For the time being, Gauche uses a simple rules for this mapping: Module names are organized hierarchically, using period '`.`' for separator, e.g. `gauche.mop.validator`. If such a module is requested and doesn't exist in the current running environment, Gauche maps the module name to a pathname by replacing periods to directory separator, i.e. `gauche/mop/validator`, and look for `gauche/mop/validator.scm` in the load paths.

Note that this is just a default behavior. Theoretically, one Scheme source file may contain multiple modules, or one module implementation may span to multiple files. In future, there may be some hook to customize this mapping for special cases. So, when you are writing routines that deal with modules and library files, do not apply the above default rule blindly. Gauche provides two procedures, `module-name->path` and `path->module-name`, to do mapping for you (see Section 4.13.6 [Module introspection], page 76, for details).

4.13.3 Defining and selecting modules

`define-module` *name body . . .* [Special Form]

Name must be a symbol. If a module named *name* does not exist, create one. Then evaluates *body* sequentially in the module.

`select-module` *name* [Special Form]

Makes a module named *name* as the current module. It is an error if no module named *name* exists.

If `select-module` is used in the Scheme file, its effect is limited inside the file, i.e. even if you load/require a file that uses `select-module` internally, the current module of requirer is not affected.

`with-module` *name body . . .* [Special Form]

Evaluates *body* sequentially in the module named *name*. Returns the last result(s). If no module named *name*, an error is signaled.

`current-module` [Special Form]

Evaluates to the current module in the compile context. Note that this is a special form, not a function. Module in Gauche is statically determined at compile time.

```
(define-module foo
  (export get-current-module)
  (define (get-current-module) (module-name (current-module))))

(define-module bar
  (import foo)
  (get-current-module)) ⇒ foo ; not bar
```

4.13.4 Using modules

`export` *spec . . .* [Special Form]

[R7RS base] Makes bindings specified by each *spec* available to modules that imports the current module.

Each *spec* can be either one of the following forms, where *name* and *exported-name* are symbols.

name The binding with *name* is exported.

(rename *name* *exported-name*)
: The binding with *name* is exported under an alias *exported-name*.

Note: In Gauche, `export` is just a special form you can put in the middle of the program, whereas R7RS defines `export` as a library declaration, that can only appear immediately below `define-library` form. See Section 10.2.1 [R7RS library form], page 503, for the details.

`export-all` [Special Form]

Makes all bindings in the current module available to modules that imports it.

`import` *import-spec . . .* [Special Form]

Makes all or some exported bindings in the module specified by *import-spec* available in the current module. The syntax of *import-spec* is as follows.

```
<import-spec> : <module-name>
              | (<module-name> <import-option> ...)

<import-option> : :only (<symbol> ...)
                | :except (<symbol> ...)
                | :rename ((<symbol> <symbol>) ...)
                | :prefix <symbol>

<module-name> : <symbol>
```

The module named by *module-name* should exist when the compiler sees this special form.

Imports are not transitive. The modules that *module-name*s are importing are not automatically imported to the current module. This keeps modules' modularity; a library module can import whatever modules it needs without worrying about polluting the namespace of the user of the module.

import-option can be used to change how the bindings are imported. With `:only`, only the bindings with the names listed in `<symbol> ...` are imported. With `:except`, the exported bindings except the ones with the listed names are imported. With `:rename`, the binding of each name in the first of two-symbol list is renamed to the second of it. With `:prefix`, the exported bindings are visible with the names that are prefixed by the symbol to the original names. Without import options, all the exported bindings are imported without a prefix.

```
(define-module M (export x y)
  (define x 1)
  (define y 2)
  (define z 3))

(import M)

x ⇒ 1
z ⇒ error. z is not exported from M

(import (M :only (y)))

x ⇒ error. x is not in :only list.
```

```
(import (M :except (y)))

y ⇒ error. y is excluded by :except.

(import (M :prefix M:))

x ⇒ error
M:x ⇒ 1
M:y ⇒ 2
```

If more than one import option are given, it is processed as the order of appearance. That is, if `:prefix` comes first, then `:only` or `:except` has to list the name with prefix.

Note: R7RS has `import` form, which has slightly different syntax and semantics. See Section 10.1.2 [Three forms of import], page 501, for the details.

`use` *name :key only except rename prefix* [Macro]

A convenience macro that combines module imports and on-demand file loading. Basically, `(use foo)` is equivalent to the following two forms:

```
(require "foo")
(import foo)
```

That is, it loads the library file named “`foo`” (if not yet loaded) which defines a module named `foo` in it, and then import the module `foo` into the current module.

The keyword argument *only*, *except*, and *prefix* are passed to `import` as the import options.

```
(use srfi-1 :only (iota) :prefix srfi-1:)

(srfi-1:iota 3) ⇒ (0 1 2)
```

Although the files and modules are orthogonal concept, it is practically convenient to separate files by modules. Gauche doesn't force you to do so, and you can always use `require` and `import` separately. However, all modules provided with Gauche are arranged so that they can be used by `use` macro.

If a module is too big to fit in one file, you can split them into several subfiles and one main file. The main file defines the module, and either loads, requires, or autoloads subfiles.

Actually, the file pathname of the given module name is obtained by the procedure `module-name->path` below. The default rule is to replace periods ‘`.`’ in the *name* for ‘`/`’; for example, `(use foo.bar.baz)` is expanded to:

```
(require "foo/bar/baz")
(import foo.bar.baz)
```

This is not very Scheme-ish way, but nevertheless convenient. In future, there may be some mechanism to customize this mapping.

The file to be `use`'d must have explicit module selection to have any toplevel definitions (usually via `define-module`/`select-module` pair or `define-library`). If you get an error saying “Attempted to create a binding in a sealed module: module: #<module gauche.require-base>”, that's because the file lacks module selection. See Section 6.23.3 [Require and provide], page 241, for further discussion.

4.13.5 Module inheritance

The export-import mechanism doesn't work well in some cases, such as:

- You want to create a module that is mostly the same as the existing one, but adding or altering some definitions.

- You wrote a bunch of related modules that are often used together, and not want your users to repeat a bunch of 'use' forms every time they use your module.

You can use module inheritance in these cases.

`extend` *module-name ...* [Macro]

Makes the current module inherit from named modules. The current inheritance information is altered by the inheritance information calculated from given modules.

A new module inherits from `gauche` module when created. If you put `(extend scheme)` in that module, for example, the module resets to inherit directly from `scheme` module that has only bindings defined in R5RS, hence, after the export form, you can't use 'import' or any other `gauche`-specific bindings in the module.

If a named module is not defined yet, `extend` tries to load it, using the same convention `use` macro does.

A module can inherit multiple modules, exactly the same way as a class can inherit from multiple classes. The resolution of order of inheritance needs to be explained a bit.

Each module has a *module precedence list*, which lists modules in the order of how they are searched. When the module inherits multiple modules, module precedence lists of inherited modules are merged into a single list, keeping the constraints that: (1) if a module A appears before module B in some module precedence list, A has to appear before B in the resulting module precedence list; and (2) if a module A appears before module B in `extend` form, A has to appear before B in the resulting module precedence list. If no precedence list can be constructed with these constraints, an error is signaled.

For example, suppose you wrote a library in modules `mylib.base`, `mylib.util` and `mylib.system`. You can bundle those modules into one module by creating a module `mylib`, as follows:

```
(define-module mylib
  (extend mylib.system mylib.util mylib.base))
```

The user of your module just says `(use mylib)` and all exported symbols from three submodules become available.

4.13.6 Module introspection

This subsection lists procedures that operates on modules at run-time. With these procedures you can introspect the modules, create new modules procedurally, or check the existence of certain modules/libraries, for example. However, don't forget that modules are primarily compile-time structures. Tweaking modules at run-time is only for those who know what they are doing.

`<module>` [Builtin Class]

A module class.

`module?` *obj* [Function]

Returns true if *obj* is a module.

`find-module` *name* [Function]

Returns a module object whose name is a symbol *name*. If the named module doesn't exist, `#f` is returned.

`make-module` *name :key if-exists* [Function]

Creates and returns a module that has symbol *name*. If the named module already exists, the behavior is specified by *if-exists* keyword argument. If it is `:error` (default), an error is signaled. If it is `#f`, `#f` is returned.

Note that creating modules on-the-fly isn't usually necessary for ordinal scripts, since to execute already written program requires modules to be specified by name, i.e. syntax `define-module`, `import`, `extend`, `with-module` all take module names, not module objects. It is because module are inherently compile-time structures. However, there are some cases that dynamically created modules are useful, especially the program itself is dynamically created. You can pass a module to `eval` to compile and evaluate such dynamically created programs in it (see Section 6.21 [Eval and repl], page 216).

You can also pass `#f` to *name* to create *anonymous* module. Anonymous modules can't be looked up by `find-module`, nor can be imported or inherited (since `import` and `extend` take module names, not modules). It is useful when you want to have a temporary, segregated namespace dynamically—for example, you can create an anonymous module to evaluate code fragments sent from other program, and discards the module when the connection is terminated. Anonymous modules are not registered in the system dictionary and are garbage collected when nobody keeps reference to it.

R7RS provides another way to create a transient module with `environment` procedure. see Section 10.2.7 [R7RS eval], page 508, for the details.

`all-modules` [Function]

Returns a list of all named modules. Anonymous modules are not included.

`module-name` *module* [Function]
`module-imports` *module* [Function]
`module-exports` *module* [Function]
`module-table` *module* [Function]

Accessors of a module object. Returns the name of the module (a symbol), list of imported modules, list of exported symbols, and a hash table that maps symbols to bindings, of the *module* are returned, respectively.

If the *module* exports all symbols, `module-exports` returns `#t`.

It is an error to pass a non-module object.

`module-parents` *module* [Function]
`module-precedence-list` *module* [Function]

Returns the information of module inheritance. `Module-parents` returns the modules *module* directly inherits from. `Module-precedence-list` returns the module precedence list of *module* (see Section 4.13.5 [Module inheritance], page 75).

`global-variable-bound?` *module symbol* [Function]

Returns true if *symbol*'s global binding is visible from *module*. *Module* must be a module object or a symbol name of an existing module.

Note: there used to be the `symbol-bound?` procedure to check whether a global variable is bound. It is deprecated and the new code should use `global-variable-bound?` instead. The reason of change is that because of the name `symbol-bound?` and the fact that it assumes current-module by default, it gives an illusion as if a global bound value is somewhat 'stored' in a symbol itself (like CommonLisp's model). It caused a lot of confusion when the current module differs between compile-time and runtime. The new name and API made it clear that you are querying module's property.

`global-variable-ref` *module symbol :optional default* [Function]

Returns a value globally bound to the *symbol* visible from *module*. *Module* must be a module object or a symbol name of an existing module. If there's no visible global binding from *module* for *symbol*, an error is signaled, unless the *default* argument is provided, in which case it is returned instead.

`module-name->path` *symbol* [Function]

Converts a module name *symbol* to a fragment of pathname string (which you use for `require` and `provide`).

`path->module-name` *string* [Function]

Reverse function of `module-name->path`.

If you want to find out specific libraries and/or modules are installed in the system and available from the program, see Section 6.23.5 [Operations on libraries], page 243.

4.13.7 Predefined modules

Several modules are predefined in Gauche.

`null` [Builtin Module]

This module corresponds to the null environment referred in R5RS. This module contains only syntactic bindings of R5RS syntax.

`scheme` [Builtin Module]

This module contains all the binding of `null` module, and the binding of procedures defined in R5RS.

Note that if you change the current module to `null` or `scheme` by `select-module`, there will be no way to switch back to other modules, since module-related syntaxes and procedures are not visible from `null` and `scheme` modules.

`gauche` [Builtin Module]

This module contains all the bindings of `scheme` module, plus Gauche specific built-in procedures.

`user` [Builtin Module]

This module is the default module the user code is compiled. all the bindings of `gauche` module is imported.

`gauche.keyword` [Builtin Module]
`keyword` [Builtin Module]

When Gauche is running with `GAUCHE_KEYWORD_IS_SYMBOL` mode (default) keywords (symbols beginning with `:`) is automatically bound to itself in these modules. (see Section 6.8 [Keywords], page 139, for the details.)

The `keyword` module doesn't export those bindings, while `gauche.keyword` does. The former is intended to be used internally; the programmer need to know the latter.

If you use the default module inheritance, you don't need to use this module, since the `keyword` module is included in the inheritance chain. If you don't inherit `gauche` module, however, importing the `gauche.keyword` module gives you access to the keywords without quotes. For example, R7RS programs and libraries would require either `(import (gauche keyword))` or `(import (gauche base))` (the latter inherits `gauche.keyword`), or you have to quote all keywords.

The following R7RS program imports `gauche.base`; it makes gauche built-in identifiers, *and* all self-bound keywords, available:

```
;; R7RS program
(import (scheme base)
        (gauche base))  ; import gauche builtins and keywords

;; You can use :directory without quote, for it is bound to itself.
```

```
(sys-exec "ls" '("ls" "-l") :directory "/")
```

If you use more sophisticated import tricks, however, keep in mind that keywords are just imported symbols by default. The following code imports Gauche builtin identifiers with prefix `gauche/`. That causes keywords, imported via inheritance, also get the same prefix; if you don't want to bother adding prefix to all keywords or quote them, import `gauche.keyword` separately.

```
;; R7RS program
(import (scheme base)
        (prefix (gauche base) gauche/) ; use gauche builtin with gauche/ prefix
        (gauche keyword))              ; imports keywords

;; Without importing gauche.keyword,
;; you need to write ':directory
(gauche/sys-exec "ls" '("ls" "-l") :directory "/")
```

5 Macros

Macro of Lisp-family language is very different feature from ones of other languages, such as C preprocessor macros. It allows you to extend the original language syntax. You can use macros to change Gauche syntax so that you can run a Scheme program written to other Scheme implementations, and you can even design your own mini-language to solve your problem easily.

Gauche supports hygienic macros, which allows to write safe macros by avoiding name collisions. If you know traditional Lisp macros but new to hygienic macros, they might seem confusing at first. We have an introductory section (Section 5.1 [Why hygienic?], page 80) for those who are not familiar with hygienic macros; if you know what they are, you can skip the section.

5.1 Why hygienic?

Lisp macro is a programmatic transformation of source code. A *macro transformer* is a procedure that takes a subtree of source code, and returns a reconstructed tree of source code.

The traditional Lisp macros take the input source code as an S-expression, and returns the output as another S-expression. Gauche supports that type of macro, too, with `define-macro` form. Here's the simple definition of `when` with the traditional macro.

```
(define-macro (when test . body)
  `(if ,test (begin ,@body)))
```

For example, if the macro is used as `(when (zero? x) (print "zero") 'zero)`, the above macro transformer rewrites it to `(if (zero? x) (begin (print "zero") 'zero))`. So far, so good.

But what if the `when` macro is used in an environment where the names `begin` or `if` is bound to nonstandard values?

```
(let ([begin list])
  (when (zero? x) (print "zero") 'zero))
```

The expanded result would be as follows:

```
(let ([begin list])
  (if (zero? x) (begin (print "zero") 'zero)))
```

This obviously won't work as the macro writer intended, since `begin` in the expanded code refers to the locally bound name.

This is a form of *variable capture*. Note that, when Lisp people talk about variable capture of macros, it often means another form of capture, where the temporary variables inserted by a macro would unintentionally capture the variables passed to the macro. That kind of variable capture can be avoided easily by naming the temporary variables something that never conflict, using `gensym`.

On the other hand, the kind of variable capture in the above example can't be avoided by `gensym`, because `(let ([begin list]) ...)` part isn't under macro writer's control. As a macro writer, you can do nothing to prevent the conflict, just hoping the macro user won't do such a thing. Sure, rebinding `begin` is a crazy idea that nobody perhaps wants to do, but it can happen on *any* global variable, even the ones you define for your library.

Various Lisp dialects have tried to address this issue in different ways. Common Lisp somewhat relies on the common sense of the programmer—you can use separate packages to reduce the chance of accidental conflict but can't prevent the user from binding the name in the same package. (The Common Lisp spec says it is undefined if you locally rebind names of CL standard symbols; but it doesn't prevent you from locally rebinding symbols that are provided by user libraries.)

Clojure introduced a way to directly refer to the toplevel variables by a namespace prefix, so it can bypass whatever local bindings of the same name (also, it has a sophisticated quasiquote form that automatically renames free variables to refer to the toplevel ones). It works, as far as there are no local macros. With local macros, you need a way to distinguish different local bindings of the same name, as we see in the later examples. Clojure's way can only distinguish between local and toplevel bindings. It's ok for Clojure which doesn't have local macros, but in Scheme, we prefer uniform and orthogonal axioms—if functions can be defined locally with lexical scope, why not macros?

Let's look at the local macro with lexical scope. For the sake of explanation, suppose we have *hypothetical* local macro binding form, `let-macro`, that binds a local identifiers to a macro transformer. (We don't actually have `let-macro`; what we have is `let-syntax` and `letrec-syntax`, which have slightly different way to call macro transformers. But here `let-macro` may be easier to understand as it is similar to `define-macro`.)

```
(let ([f (^x (* x x))])
  (let-macro ([m (^[expr1 expr2] `(+ (f ,expr1) (f ,expr2)))])
    (let ([f (^x (+ x x))])
      (m 3 4))))    ; [1]
```

The local identifier *m* is bound to a macro transformer that takes two expressions, and returns an S-expression. So, the `(m 3 4)` form [1] would be expanded into `(+ (f 3) (f 4))`. Let's rewrite the above expression with the expanded form. (After expansion, we no longer need `let-macro` form, so we don't include it.)

```
(let ([f (^x (* x x))])
  (let ([f (^x (+ x x))])
    (+ (f 3) (f 4))))  ; [2]
```

Now, the question. Which binding `f` in the expanded form [2] should refer? If we literally interpret the expansion, it would refer to the inner binding `(^x (+ x x))`. However, following the Scheme's scoping principle, the outer code should be fully understood regardless of inner code:

```
(let ([f (^x (* x x))])
  (let-macro ([m (^[expr1 expr2] `(+ (f ,expr1) (f ,expr2)))])
    ;; The code here isn't expected to accidentally alter
    ;; the behavior defined outside.
    ))
```

The macro writer may not know the inner `let` shadows the binding of `f` (the inner forms may be `included`, or may be changed by other person who didn't fully realize the macro expansion needs to refer outer `f`).

To ensure the local macro to work regardless of what's placed inside `let-macro`, we need a sure way to refer the outer `f` in the result of macro expansion. The basic idea is to "mark" the names inserted by the macro transformer `m`—which are `f` and `+`—so that we can distinguish two `f`'s.

For example, if we would rewrite the entire form and *renames* corresponding local identifiers as follows:

```
(let ([f_1 (^x (* x x))])
  (let-macro ([m (^[expr1 expr2] `(+ (f_1 ,expr1) (f_1 ,expr2)))])
    (let ([f_2 (^x (+ x x))])
      (m 3 4))))
```

Then the naive expansion would correctly preserve scopes; that is, expansion of `m` refers `f_1`, which wouldn't conflict with inner name `f_2`:

```
(let ([f_1 (^x (* x x))])
```

```
(let ([f_2 (^x (+ x x))])
  (+ (f_1 3) (f_1 4))))
```

(You may notice that this is similar to lambda calculus treating lexical bindings with higher order functions.)

The above example deal with avoiding `f` referred from the macro *definition* (which is, in fact, `f_1`) from being shadowed by the binding of `f` at the macro *use* (which is `f_2`).

Another type of variable capture (the one most often talked about, and can be avoided by `gensym`) is that a variable in macro use site is shadowed by the binding introduced by a macro definition. We can apply the same renaming strategy to avoid that type of capture, too. Let's see the following example:

```
(let ([f (^x (* x x))])
  (let-macro ([m (^[expr1] `(let ([f (^x (+ x x))]) (f ,expr1)))])
    (m (f 3))))
```

The local macro inserts binding of `f` into the expansion. The macro use `(m (f 3))` also contains a reference to `f`, which should be the outer `f`, since the macro use is lexically outside of the `let` inserted by the macro.

We could rename `f`'s according to its lexical scope:

```
(let ([f_1 (^x (* x x))])
  (let-macro ([m (^[expr1] `(let ([f_2 (^x (+ x x))]) (f_2 ,expr1)))])
    (m (f_1 3))))
```

Then expansion unambiguously distinguish two `f`'s.

```
(let ([f_1 (^x (* x x))])
  (let ([f_2 (^x (+ x x))])
    (f_2 (f_1 3))))
```

This is, in principle, what hygienic macro is about (well, almost). In reality, we don't rename everything in batch. One caveat is in the latter example—we statically renamed `f` to `f_2`, but it is possible that the macro recursively calls itself, and we have to distinguish `f`'s introduced in every individual expansion of `m`. So macro expansion and renaming should work together.

There are multiple strategies to implement it, and the Scheme standard doesn't want to bind implementations to single specific strategy. The standard only states the properties the macro system should satisfy, in two concise sentences:

> If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers.
>
> If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that surround the use of the macro.

Just from reading this, it may not be obvious *how* to realize those properties, and the existing hygienic macro mechanisms (e.g. `syntax-rules`) hide the "how" part. That's probably one of the reason some people feel hygienic macros are difficult to grasp. It's like continuations—its description is concise but at first you have no idea how it works; then, through experience, you become familiarized yourself to it, and then you reread the original description and understand it says exactly what it is.

This introduction may not answer *how* the hygienic macro realizes those properties, but I hope it showed *what* it does and *why* it is needed. In the following chapters we introduce a couple of hygienic macro mechanisms Gauche supports, with examples, so that you can familiarize yourself to the concept.

5.2 Hygienic macros

Macro bindings

The following forms establish bindings of *name* and a macro transformer created by *transformer-spec*. The binding introduced by these forms shadows a binding of *name* established in outer scope, if there's any.

For toplevel bindings, it will shadow bindings of *name* imported or inherited from other modules (see Section 4.13 [Modules], page 71). (Note: This toplevel shadowing behavior is Gauche's extension; in R7RS, you shouldn't redefine imported bindings, so the portable code should avoid it.)

The effect is undefined if you bind the same name more than once in the same scope.

The *transformer-spec* can be either one of `syntax-rules` form, `er-macro-transformer` form, or another macro keyword or syntactic keyword. We'll explain them later.

`define-syntax` *name transformer-spec* [Special Form]

[R7RS base] If this form appears in toplevel, it binds toplevel *name* to a macro transformer defined by *transformer-spec*.

If this form appears in the *declaration* part of body of `lambda` (internal define-syntax), `let` and other similar forms, it binds *name* locally within that body. Internal `define-syntax`es are converted to `letrec-syntax`, just like internal `define`s are converted to `letrec*`.

`let-syntax` ((*name transformer-spec*) . . .) *body* [Special Form]
`letrec-syntax` ((*name transformer-spec*) . . .) *body* [Special Form]

[R7RS base] Defines local macros. Each *name* is bound to a macro transformer as specified by the corresponding *transformer-spec*, then `body` is expanded. With `let-syntax`, *transformer-spec* is evaluated with the scope surrounding `let-syntax`, while with `letrec-syntax` the bindings of *name*s are included in the scope where *transformer-spec* is evaluated. Thus `letrec-syntax` allows mutually recursive macros.

Transformer specs

The *transformer-spec* is a special expression that evaluates to a macro transformer. It is evaluated in a different phase than the other expressions, since macro transformers must be executed during compiling. So there are some restrictions.

At this moment, only one of the following expressions are allowed:

1. A `syntax-rules` form. This is called "high-level" macro, for it uses pattern matching entirely, which is basically a different declarative language from Scheme, thus putting the complication of the phasing and hygiene issues completely under the hood. Some kind of macros are easier to write in `syntax-rules`. See Section 5.2.1 [Syntax-rules macro transformer], page 84, for further description.
2. An `er-macro-transformer` form. This employs *explicit-renaming* (ER) macro, where you can use arbitrary Scheme code to transform the program, with required renaming to keep hygienity. The legacy Lisp macro can also be written with ER macro if you don't use renaming. See Section 5.2.2 [Explicit-renaming macro transformer], page 86, for the details.
3. Macro or syntax keyword. This is Gauche's extension, and can be used to define alias of existing macro or syntax keyword.

```
(define-syntax si if)
(define écrivez write)

(si (< 2 3) (écrivez "oui"))
```

5.2.1 Syntax-rules macro transformer

`syntax-rules` (*literal* ...) *clause clause2* ... [Special Form]
`syntax-rules` *ellipsis* (*literal* ...) *clause clause2* ... [Special Form]

[R7RS base] This form creates a macro transformer by pattern matching.

Each *clause* has the following form:

```
(pattern template)
```

A *pattern* denotes a pattern to be matched to the macro call. It is an S-expression that matches if the macro call has the same structure, except that symbols in *pattern* can match a whole subtree of the input; the matched symbol is called a *pattern variable*, and can be referenced in the *template*.

For example, if a pattern is `(_ "foo" (a b))`, it can match the macro call `(x "foo" (1 2))`, or `(x "foo" (1 (2 3)))`, but does not match `(x "bar" (1 2))`, `(x "foo" (1))` or `(x "foo" (1 2) 3)`. You can also match repeating structure or literal symbols; we'll discuss it fully later.

Clauses are examined in order to see if the macro call form matches its pattern. If matching pattern is found, the corresponding *template* replaces the macro call form. A pattern variable in the template is replaced with the subtree of input that is bound to the pattern variable.

Here's a definition of `when` macro in Section 5.1 [Why hygienic?], page 80, using `syntax-rules`:

```
(define-syntax when
  (syntax-rules ()
    [(_ test body ...) (if test (begin body ...))]))
```

The pattern is `(_ test body ...)`, and the template is `(if test (begin body ...))`. The ellipsis `...` is a symbol; we're not omitting code here. It denotes that the previous pattern (`body`) may repeat zero or more times.

So, if the `when` macro is called as `(when (zero? x) (print "huh?") (print "we got zero!"))`, the macro expander first check if the input matches the pattern.

- The *test* in pattern matches the input `(zero? x)`.
- The *body* in pattern matches the input `(print "huh?")` *and* `(print "we got zero!")`.

The matching of *body* is a bit tricky; as a pattern variable, you may think that *body* works like an array variable, each element holds each match—and you can use them in similarly repeating substructures in template. Let's see the template, now that the input fully matched the pattern.

- In the template, `if` and `begin` are not pattern variable, since they are not appeared in the pattern. So they are inserted as identifiers—that is, hygienic symbols effectively renamed to make sure to refer to the global `if` and `begin`, and will be unaffected by the macro use environment.
- The *test* in the template is a pattern variable, so it is replaced for the matched value, `(zero? x)`.
- The *body* is also a pattern variable. The important point is that it is also followed by ellipsis. So we repeat *body* as many times as the number of matched values. The first value, `(print "huh?")`, and the second value, `(print "we got zero!")`, are expanded here.
- Hence, we get `(if (zero? x) (begin (print "huh?") (print "we got zero!")))` as the result of expansion. (With the note that `if` and `begin` refers to the identifiers visible from the macro definition environment.)

The expansion of ellipses is quite powerful. In the template, the ellipses don't need to follow the sequence-valued pattern variable immediately; the variable can be in a substructure, as long as the substructure itself is followed by an ellipsis. See the following example:

```
(define show
  (syntax-rules ()
    [(_ expr ...)
     (begin
       (begin (write 'expr) (display "=") (write expr) (newline))
       ...)]))
```

If you call this macro as follows:

```
(show (+ 1 2) (/ 3 4))
```

It is expanded to the following form, modulo hygienity:

```
(begin
  (begin (write '(+ 1 2)) (display "=") (write (+ 1 2)) (newline))
  (begin (write '(/ 3 4)) (display "=") (write (/ 3 4)) (newline)))
```

So you'll get this output.

```
(+ 1 2)=3
(/ 3 4)=3/4
```

You can also match with a repetition of substructures in the pattern. The following example is a simplified `let` that expands to `lambda`:

```
(define-syntax my-let
  (syntax-rules ()
    [(_ ((var init) ...) body ...)
     ((lambda (var ...) body ...) init ...)]))
```

If you call it as `(my-let ((a expr1) (b expr2)) foo)`, then *var* is matched to `a` and `b`, while *init* is matched to `expr1` and `expr2`, respectively. They can be used separately in the template.

Suppose "level" of a pattern variable means the number of nested ellipses that designate repetition of the pattern variable. A subtemplate can be followed as many ellipses as the maximum level of pattern variables in the subtemplate. In the following example, the level of pattern variable `a` is 1 (it is repeated by the last ellipsis in the pattern), while the level of `b` is 2 (repeated by the last two ellipses), and the level of `c` is 3 (repeated by all the ellipses).

```
(define-syntax ellipsis-test
  (syntax-rules ()
    [(_ (a (b c ...) ...) ...)
     '((a ...)
       (((a b) ...) ...)
       ((((a b c) ...) ...) ...))]))
```

In this case, the subtemplate `a` must be repeated by one level of ellipsis, `(a b)` must be repeated by two, and `(a b c)` must be repeated by three.

```
(ellipsis-test (1 (2 3 4) (5 6)) (7 (8 9 10 11)))
 ⇒ ((1 7)
    (((1 2) (1 5)) ((7 8)))
    ((((1 2 3) (1 2 4)) ((1 5 6))) (((7 8 9) (7 8 10) (7 8 11)))))
```

In the template, more than one ellipsis directly follow a subtemplate, splicing the leaves into the surrounding list:

```
(define-syntax my-append
  (syntax-rules ()
```

```
    [(_ (a ...) ...)
     '(a ... ...)]))

(my-append (1 2 3) (4) (5 6))
  ⇒ (1 2 3 4 5 6)

(define-syntax my-append2
  (syntax-rules ()
    [(_ ((a ...) ...) ...)
     '(a ... ... ...)]))

(my-append2 ((1 2) (3 4)) ((5) (6 7 8)))
  ⇒ (1 2 3 4 5 6 7 8)
```

Note: Allowing multiple ellipses to directly follow a subtemplate, and a pattern variable in a subtemplate to be enclosed within more than the variable's level of nesting of ellipses, are extension to R7RS, and defined in SRFI-149. In the above examples, `ellipsis-test`, `my-append` and `my-append2` are outside of R7RS.

Identifiers in a pattern is treated as pattern variables. But sometimes you want to match a specific identifier in the input. For example, the built-in `cond` and `case` detects an identifier `else` as a special identifier. You can use *literal* ... for that. See the following example.

```
(define-syntax if+
  (syntax-rules (then else)
    [(_ test then expr1 else expr2) (if test expr1 expr2)]))
```

The identifiers listed as the literals don't become pattern variables, but literally match the input. If the input doesn't have the same identifier in the position, match fails.

```
(if+ (even? x) then (/ x 2) else (/ (+ x 1) 2))
 expands into (if (even? x) (/ x 2) (/ (+ x 1) 2))

(if+ (even? x) foo (/ x 2) bar (/ (+ x 1) 2))
 ⇒ ERROR: malformed if+
```

We've been saying identifiers instead of symbols. Roughly speaking, an identifier is a symbol with the surrounding syntactic environment, so that they can keep identity under renaming of hygiene macro.

The following example fails, because the `else` passed to the `if+` macro is the one locally bound by `let`, which is different from the global `else` when `if+` was defined, hence they don't match.

```
(let ((else #f))
  (if+ (even? x) then (/ x 2) else (/ (+ x 1) 2))
  ⇒ ERROR: malformed if+
```

5.2.2 Explicit-renaming macro transformer

`er-macro-transformer` *procedure-expr* [Special Form]

Creates a macro transformer from the given *procedure-expr*. The created macro transformer has to be bound to the syntactic keyword by `define-syntax`, `let-syntax` or `letrec-syntax`. Other use of macro transformers is undefined.

The *procedure-expr* must evaluate to a procedure that takes three arguments; *form*, *rename* and *id=?*.

The *form* argument receives the S-expression of the macro call. The *procedure-expr* must return an S-expression as the result of macro expansion. This part is pretty much like the traditional lisp macro. In fact, if you ignore *rename* and *id=?*, the semantics is the same as the traditional (unhygienic) macro. See the following example (Note the use of `match`; it is a good tool to decompose macro input):

```
(use util.match)

;; Unhygienic 'when-not' macro
(define-syntax when-not
  (er-macro-transformer
    (^[form rename id=?]
      (match form
        [(_ test expr1 expr ...)
         `(if (not ,test) (begin ,expr1 ,@expr))]
        [_ (error "malformed when-not:" form)]))))

(macroexpand '(when-not (foo) (print "a") 'boo))
  ⇒ (if (not (foo)) (begin (print "a") 'boo))
```

This is ok as long as you know you don't need hygiene—e.g. when you only use this macro locally in your code, knowing all the macro call site won't contain name conflicts. However, if you provide your `when-not` macro for general use, you have to protect namespace pollution around the macro use. For example, you want to make sure your macro work even if it is used as follows:

```
(let ((not values))
  (when-not #t (print "This shouldn't be printed")))
```

The *rename* argument passed to *procedure-expr* is a procedure that takes a symbol (or, to be precise, a symbol or an identifier) and *effectively renames* it to a unique identifier that keeps identity within the macro definition environment and won't be affected in the macro use environment.

As a rule of thumb, you have to pass *all new identifiers you insert into macro output* to the *rename* procedure to keep hygiene. In our `when-not` macro, we insert `if`, `not` and `begin` into the macro output, so our hygienic macro would look like this:

```
(define-syntax when-not
  (er-macro-transformer
    (^[form rename id=?]
      (match form
        [(_ test expr1 expr ...)
         `(,(rename 'if) (,(rename 'not) ,test)
            (,(rename 'begin) ,expr1 ,@expr))]
        [_ (error "malformed when-not:" form)]))))
```

This is cumbersome and makes it hard to read the macro, so Gauche provides an auxiliary macro `quasirename`, which works like `quasiquote` but renaming identifiers in the form. See the entry of `quasirename` below for the details. You can write the hygienic `when-not` as follows:

```
(define-syntax when-not
  (er-macro-transformer
    (^[form rename id=?]
      (match form
        [(_ test expr1 expr ...)
         (quasirename rename
           `(if (not ,test) (begin ,expr1 ,@expr)))]
        [_ (error "malformed when-not:" form)]))))
```

You can intentionally break hygiene by inserting a symbol without renaming. The following code implements *anaphoric* `when`, meaning the result of the test expression is available in the *expr1 exprs* . . . with the name `it`. Since the binding of the identifier `it` does not exist in the macro use site, but rather injected into the macro use site by the macro expander, it is unhygienic.

```
(define-syntax awhen
  (er-macro-transformer
    (^[form rename id=?]
      (match form
        [(_ test expr1 expr ...)
         `(,(rename 'let1) it ,test     ; 'it' is not renamed
             (,(rename 'begin) ,expr1 ,@expr))]))))
```

If you use `quasirename`, you can write `,'it` to prevent `it` from being renamed:

```
(define-syntax awhen
  (er-macro-transformer
    (^[form rename id=?]
      (match form
        [(_ test expr1 expr ...)
         (quasirename rename
           `(let1 ,'it ,test
              (begin ,expr1 ,@expr)))]))))
```

Here's an example:

```
(awhen (find odd? '(0 2 8 7 4))
  (print "Found odd number:" it))
 ⇒ prints Found odd number:7
```

Finally, the *id=?* argument to the *procedure-expr* is a procedure that takes two arguments, and returns `#t` iff both are identifiers and either both are referring to the same binding or both are free. It can be used to compare literal syntactic keyword (e.g. `else` in `cond` and `case` forms) hygienically.

The following `if=>` macro behaves like `if`, except that it accepts `(if=> test => procedure)` syntax, in which `procedure` is called with the value of `test` if it is not false (similar to `(cond [test => procedure])` syntax). The symbol `=>` must match hygienically, that is, it must refer to the same binding as in the macro definition.

```
(define-syntax if=>
  (er-macro-transformer
    (^[form rename id=?]
      (match form
        [(_ test a b)
         (if (id=? (rename '=>) a)
           (quasirename rename
             `(let ((t ,test))
                (if t (,b t))))
           (quasirename rename
             `(if ,test ,a ,b)))]))))
```

The call `(rename '=>)` returns an identifier that captures the binding of `=>` in the macro definition, and using `id=?` with the thing passed to the macro argument checks if both refer to the same binding.

```
(if=> 3 => list)  ⇒ (3)
(if=> #f => list) ⇒ #<undef>
```

```
;; If the second argument isn't =>, if=> behaves like ordinary if:
(if=> #t 1 2)     ⇒ 1

;; The binding of => in macro use environment differs from
;; the macro definition environment, so this if=> behaves like
;; ordinary if, instead of recognizing literal =>.
(let ((=> 'oof)) (if=> 3 => list)) ⇒ oof
```

`quasirename` *renamer quasiquoted-form* [Macro]

It works like quasiquote, except that the symbols and identifiers that appear in the "literal" portion of *form* (i.e. outside of `unquote` and `unquote-splicing`) are replaced by the result of applying *rename* on themselves.

The *quasiquote-form* argument must be a quasiquoted form. The outermost quasiquote ` is consumed by `quasirename` and won't appear in the output. The reason we require it is to make nested quasiquotes/quasirenames work.

For example, a form:

```
(quasirename r `(a ,b c "d"))
```

would be equivalent to write:

```
(list (r 'a) b (r 'c) "d")
```

This is not specifically tied to macros; the *renamer* can be any procedure that takes one symbol or identifier argument:

```
(quasirename (^[x] (symbol-append 'x: x)) `(+ a ,(+ 1 2) 5))
  ⇒ (x:+ x:a 3 5)
```

However, it comes pretty handy to construct the result form in ER macros. Compare the following two:

```
(use util.match)

;; using quasirename
(define-syntax swap
  (er-macro-transformer
    (^[f r c]
      (match f
        [(_ a b) (quasirename r
                   `(let ((tmp ,a))
                      (set! ,a ,b)
                      (set! ,b tmp)))]))))

;; not using quasirename
(define-syntax swap
  (er-macro-transformer
    (^[f r c]
      (match f
        [(_ a b) `((r'let) (((r'tmp) ,a))
                     ((r'set!) ,a ,b)
                     ((r'set!) ,b (r'tmp)))]))))
```

Note: In Gauche 0.9.7 and before, `quasirename` didn't use quasiquoted form as the second argument; you can write `(quasirename r form)` instead of `(quasirename r `form)`.

For the backward compatibility, we support the form without quasiquote by default for a while.

If you already have a quasirename form that does intend to produce a quasiquoted form, you have to rewrite it with double quasiquote: `(quasirename r ``form)`.

To help transition, the handling of quasiquote in of `quasirename` can be customized with the environment variable `GAUCHE_QUASIRENAME_MODE`. It can have one of the following values:

`legacy`
: `Quasirename` behaves the same way as 0.9.7 and before; use this to run code for 0.9.7 without any change.

`compatible`
: `Quasirename` behaves as described in this entry; if *form* lacks a quasiquote, it silently assumes one. Existing code should work, except the rare case when you intend to return a quasiquoted form.

`warn`
: `Quasirename` behaves as described in this entry, but warns if *form* lacks a quasiquote.

`strict`
: `Quasirename` raises an error if *form* lacks a quasiquote. This will be the default behavior in future.

5.3 Traditional macros

`define-macro` *name procedure* [Special Form]
`define-macro` (*name* . *formals*) *body* . . . [Special Form]

Defines *name* to be a global macro whose transformer is *procedure*. The second form is a shorthand notation of the following form:

```
(define-macro name (lambda formals body ...))
```

When a form `(name arg ...)` is seen by the compiler, it calls *procedure* with *arg* . . . . When *procedure* returns, the compiler inserts the returned form in place of the original form, and compile it again.

To avoid name conflict with the bindings inserted by the macro, you can use `gensym`, just like traditional Lisp macros (see Section 6.7 [Symbols], page 137).

```
(define-macro (if-let1 var test then else)
  (let1 tmp (gensym)
    `(let ((,tmp ,test))
       (if ,tmp ,then ,else))))

(macroexpand '(if-let1 v (find odd? '(2 4 6 7 8))
                 (* v v)
                 #f))
  ⇒ (let ((#:G1013 (find odd? (quote (2 4 6 7 8)))))
          (if #:G1013 (* v v) #f))
```

Note that `gensym` can't protect name conflict with global bindings inserted by the macro. Section 5.1 [Why hygienic?], page 80, discusses this issue.

5.4 Debugging macros

Macro expansion happens at the compile time, which makes it difficult to debug. The best way to avoid headache of macro debugging is not to write macros unless they're absolutely necessary, and keep them as simple as possible if you need to write ones.

However, if you find yourself in an unfortunate situation that you have to untangle hairy macros, Gauche has some tools to help.

5.4.1 Tracing macro expansion

Macro tracing shows the input to the macro expander and the result of its expansion on selected macros. Suppose you have the following macro definition. It's essentially the same as shown in the definition of `letrec` in R7RS section 7.3:

```
(define-syntax my-letrec
  (syntax-rules ()
    [(_ ((var init) ...) body ...)
     (my-letrec "tmps" (var ...) () ((var init) ...) body ...)]
    [(_ "tmps" () (tmp ...) ((var init) ...) body ...)
     (let ((var 'undefined) ...)
       (let ((tmp init) ...)
         (set! var tmp) ...
         body ...))]
    [(_ "tmps" (x y ...) (tmp ...) binds body ...)
     (my-letrec "tmps" (y ...) (newtmp tmp ...) binds body ...)]))
```

The `my-letrec` macro uses an idiom to generate temporary variables by looping with `"tmps"` tag. You can see how the macro is expanded step by step, by tracing `my-letrec`:

```
gosh> (trace-macro 'my-letrec)
(my-letrec)
gosh> (my-letrec [(ev? (^n (if (= n 0) #t (od? (- n 1)))))
                  (od? (^n (if (= n 0) #f (ev? (- n 1)))))]
        (ev? 3))
Macro input>>>
(my-letrec
 ((ev? (^n (if (= n 0) #t (od? (- n 1)))))
  (od? (^n (if (= n 0) #f (ev? (- n 1))))))
 (ev? 3))

Macro output<<<
(my-letrec
 "tmps"
 (ev? od?)
 ()
 ((ev? (^n (if (= n 0) #t (od? (- n 1)))))
  (od? (^n (if (= n 0) #f (ev? (- n 1))))))
 (ev? 3))

Macro input>>>
(my-letrec
 "tmps"
 (ev? od?)
 ()
 ((ev? (^n (if (= n 0) #t (od? (- n 1)))))
  (od? (^n (if (= n 0) #f (ev? (- n 1))))))
 (ev? 3))

Macro output<<<
(my-letrec
 "tmps"
 (od?)
 (newtmp.0)
```

```
 ((ev? (^n (if (= n 0) #t (od? (- n 1)))))
  (od? (^n (if (= n 0) #f (ev? (- n 1))))))
 (ev? 3))

Macro input>>>
(my-letrec
 "tmps"
 (od?)
 (newtmp.0)
 ((ev? (^n (if (= n 0) #t (od? (- n 1)))))
  (od? (^n (if (= n 0) #f (ev? (- n 1))))))
 (ev? 3))

Macro output<<<
(my-letrec
 "tmps"
 ()
 (newtmp.1 newtmp.0)
 ((ev? (^n (if (= n 0) #t (od? (- n 1)))))
  (od? (^n (if (= n 0) #f (ev? (- n 1))))))
 (ev? 3))

Macro input>>>
(my-letrec
 "tmps"
 ()
 (newtmp.0 newtmp.1)
 ((ev? (^n (if (= n 0) #t (od? (- n 1)))))
  (od? (^n (if (= n 0) #f (ev? (- n 1))))))
 (ev? 3))

Macro output<<<
(let
 ((ev? (quote undefined)) (od? (quote undefined)))
 (let
  ((newtmp.0 (^n (if (= n 0) #t (od? (- n 1)))))
   (newtmp.1 (^n (if (= n 0) #f (ev? (- n 1))))))
  (set! ev? newtmp.0)
  (set! od? newtmp.1)
  (ev? 3)))

#f
```

In the above example, the S-expressions after `gosh>` prompt is what you type; all other things are Gauche's answer, including `Macro input` and `Macro output` S-expressions.

The S-expression shown with `Macro input` is the input of the macro expander, and the one with `Macro output` is the expanded result. Actual macro output has syntactic information attached, but the tracer strips them off for the legibility.

Note that the loop introduces new temporary variables with the same name (`newtemp`), but they are treated as different identifiers in the macro expansion.

Once you're done debugging, don't forget to call `untrace-macro` with no arguments to remove macro traces. If there's a macro trace set, all macro expansions get some overhead, so don't leave macro traces.

```
gosh> (untrace-macro)
#f
```

`trace-macro` [Function]
`trace-macro` *boolean* [Function]
`trace-macro` *name-or-pattern ...* [Function]

Get/set current macro trace setting. Macro trace setting can be one of the following values:

`#f` Macro tracing is off. This is the default setting.

`#t` All macro expansions are traced.

`(name-or-pattern ...)`
: Trace macros that match any one of *name-or-pattern*, which is either a symbol or a regexp. If it's a symbol, a macro whose name is the same as the symbol is traced. If it's a regexp, macros whose name match the regexp are traced.

When called without arguments, `trace-macro` doesn't change the setting; it returns the current setting.

When called with single boolean value, it sets the current setting to that value. Returns the updated setting.

When called with one or more *name-or-pattern*, it *adds* them to the current setting. Note that if the current setting is `#t`, it remains `#t`, for all macros are already traced. Returns the updated setting.

If macro trace settings is not `#f`, it incurs overhead for every macro expansion. Be careful not to leave macro trace set.

The trace information is output to the current error port.

`untrace-macro` [Function]
`untrace-macro` *name-or-pattern ...* [Function]

When called without arguments, it turns macro trace off.

When called with one or more *name-or-pattern*, which is either a symbol or a regexp, `untrace-macro` removes them from the currently traced macros. Note that if the current macro trace setting is `#t` (trace all macros), you can't remove traced macro individually.

It returns the updated macro trace setting.

5.4.2 Expanding macros manually

`macroexpand` *form :optional env* [Function]
`macroexpand-1` *form :optional env* [Function]

If *form* is a list and its first element is a variable globally bound to a macro, `macroexpand-1` invokes its macro transformer and returns the expanded form. Otherwise, returns *form* as is.

`macroexpand` repeats `macroexpand-1` until the outermost expression of *form* can't be expanded. (It doesn't expand macros other than outermost one. If you want to expand all the macros within *form*, use `macroexpand-all`).

These procedures can be used to expand globally defined macros.

Internally, hygienic macro expansion wraps symbols in *form* with syntactic information to keep hygiene. However, such information is hard to read, and not suitable when you just want to expand a macro in REPL to check its result. So, by default, these procedures strips syntactic information. For the identifiers introduced in the macro, it renames them to avoid name conflicts.

The following example expands `my-letrec` macro (see Section 5.4.1 [Tracing macro expansion], page 91, for the definition) and results shows temporary variable introduced by the macro (*newtemp*) to be renamed.

```
(macroexpand
  '(my-letrec [(ev? (^n (if (= n 0) #t (od? (- n 1)))))
               (od? (^n (if (= n 0) #f (ev? (- n 1)))))]
     (ev? 3)))

⇒

(let
 ((ev? (quote undefined)) (od? (quote undefined)))
 (let
  ((newtmp.0 (^n (if (= n 0) #t (od? (- n 1)))))
   (newtmp.1 (^n (if (= n 0) #f (ev? (- n 1))))))
  (set! ev? newtmp.0)
  (set! od? newtmp.1)
  (ev? 3)))
```

If you pass a module to the *env* argument, it is used as the macro use environment. You can also pass `#t` to let it use the current *runtime* environment as the macro use environment. In those cases, syntactic information in the output won't be stripped.

If you want to use the output of `macroexpand` as a program, e.g. embed it into another macro expansion, you need syntactic information preserved.

`macroexpand-all` *form :optional env* [Function]

Fully expand macros inside *form*. The result only contains function calls and Gauche's built-in syntax.

By default, or `#t` is passed to *env*, the *form* is assumed to be a toplevel form within the current runtime module. You can also pass a module to *env* to specify the alternative toplevel environment.

Any local variables introduced in *form* is renamed to avoid collision. Since each local variable has unique name, all `let` forms become `letrec` forms (we can safely replace `let` with `letrec` if no bindings introduced by `let` shadows outer bindings.)

NB: If a macro in *form* inserts a reference to a global variable which belongs to other module, the information is lost in the current implementation. There are a few ways to address this issue; we may leave such reference as an identifier object, convert it to `with-module` form, or introduce a special syntax to represent such case. It's undecided currently, so do not rely too much on the current behavior. For the time being, it's best to use this feature only for interactive macro testing.

```
(macroexpand-all
 '(letrec-syntax
      [(when-not (syntax-rules ()
                   [(_ test . body) (if test #f (begin . body))]))]
    (let ([if list])
      (define x (expt foo))
      (let1 x 3
        (when-not (bar) (if x))))))
 ⇒ (letrec ((if.0 list))
```

```
(letrec ((x.1 (expt foo)))
  (letrec ((x.2 '3))
   (if (bar) '#f (if.0 x.2)))))
```

`%macroexpand` *form* [Special Form]
`%macroexpand-1` *form* [Special Form]

5.5 Macro utilities

`syntax-error` *msg arg . . .* [Macro]
`syntax-errorf` *fmt arg . . .* [Macro]

Signal an error. They are same as `error` and `errorf` (see Section 6.20.2 [Signaling exceptions], page 207), except that the error is signaled at macro-expansion time (i.e. compile time) rather than run time.

They are useful to tell the user the wrong usage of macro in the comprehensive way, instead of the cryptic error from the macro transformer. Because of the purpose, *arg . . .* are first passed to `unwrap-syntax` described below, to strip off the internal syntactic binding informations.

```
(define-syntax my-macro
  (syntax-rules ()
    ((_ a b)   (foo2 a b))
    ((_ a b c) (foo3 a b c))
    ((_ . ?)
     (syntax-error "malformed my-macro" (my-macro . ?)))))

(my-macro 1 2 3 4)
  ⇒ error: "malformed my-macro: (my-macro 1 2 3 4)"
```

`unwrap-syntax` *form* [Function]

Removes internal syntactic information from *form*. In order to implement a hygienic macro, macro transformer replaces symbols in the macro form for identifiers, which captures the syntactic environment where they are defined. Although it is necessary information for the compiler, it is rather confusing for users if it appears in the messages. This function replaces occurrences of identifiers in *form* to the original symbols.

6 Core library

6.1 Types and classes

Scheme is a dynamically and strongly typed language. That is, every value *knows* its type at run-time, and the type determines what kind of operations can be applied on the value.

In Gauche, *classes* are used to describe types. A class is also an object you can handle at runtime. You can also create a new class in order to have objects with distinct types.

Since R6RS, Scheme got a standard way to define a new type, through define-record-type. You can use record types in Gauche as well, via gauche.record module. See Section 9.26 [Record types], page 433. Internally a record type is implemented as a class.

In this section we introduce the most basic interface to the Gauche's type system. See Chapter 7 [Object system], page 279, for the details of how to define your own classes and creates values (instances).

Predefined classes are bound to a global variable; Gauche's convention is to name the variable that holds a class with brackets < and >, e.g. <string>. (It's nothing syntactically special with these brackets; they're valid characters to consist of variable names). We'll introduce classes for each built-in type as we go through this chapter. Here are a few basic classes to start with:

<top> [Builtin Class]

This class represents the supertype of all the types in Gauche. That is, for any class X, (subtype? X <top>) is #t, and for any object x, (is-a? x <top>) is #t.

<bottom> [Builtin Class]

This class represents the subtype of all the types in Gauche. For any class X, (subtype? <bottom> X) is #t, and for any object x, (is-a? x <bottom>) is #f.

There's no instance of <bottom>.

Note: Although <bottom> is subtype of other types, the class precedence list (CPL) of <bottom> only contains <bottom> and <top>. It's because it isn't always possible to calculate a linear list of all the types. Even if it is possible, it would be expensive to check and update the CPL of <bottom> every time a new class is defined or an existing class is redefined. Procedures subtype? and is-a? treat <bottom> specially.

One of use case of <bottom> is applicable? procedure. See Section 6.18.1 [Procedure class and applicability], page 188.

<object> [Builtin Class]

This class represents a supertype of all user-defined classes.

class-of *obj* [Function]

Returns a class metaobject of *obj*.

```
(class-of 3)         ⇒ #<class <integer>>
(class-of "foo")     ⇒ #<class <string>>
(class-of <integer>) ⇒ #<class <class>>
```

Note: In Gauche, you can redefine existing user-defined classes. If the new definition has different configuration of the instance, class-of on existing instance triggers instance updates; see Section 7.2.5 [Class redefinition], page 292, for the details. Using current-class-of suppresses instance updates (see Section 7.3.2 [Accessing instance], page 296).

is-a? *obj class* [Function]

Returns true if *obj* is an instance of *class* or an instance of descendants of *class*.

```
(is-a? 3 <integer>)   ⇒ #t
```

```
(is-a? 3 <real>)       ⇒ #t
(is-a? 5+3i <real>)    ⇒ #f
(is-a? :foo <symbol>) ⇒ #f
```

Note: If *obj*'s class has been redefined, `is-a?` also triggers instance update. See Section 7.2.5 [Class redefinition], page 292, for the details.

`subtype?` *sub super* [Function]

Returns `#t` if a class *sub* is a subclass of a class *super* (includes the case that *sub* is *super*). Otherwise, returns `#f`.

(The name `subtype?` is taken from Common Lisp's procedure `subtypep`.)

6.2 Equality and comparison

Comparing two objects seems trivial, but if you look into deeper, there are lots of subtleties hidden in the corners. What should it mean if two procedures are equal to each other? How to order two complex numbers? It all depends on your purpose; there's no single generic answer. So Scheme (and Gauche) provides several options, as well as the way to make your own.

6.2.1 Equality

Scheme has three different general equality test predicates. Other than these, some types have their own comparison predicates.

`eq?` *obj1 obj2* [Function]

[R7RS base] This is the fastest and finest predicate. Returns `#t` if *obj1* and *obj2* are identical objects—that is, if they represents the same object on memory or in a register. Notably, you can compare two symbols or two keywords with `eq?` to check if they are the same or not. You can think `eq?` as a pointer comparison for any heap-allocated objects.

Booleans can be compared with `eq?`, but you can't compare characters and numbers reliably—objects with the same numerical value may or may not `eq?` to each other. If you identity comparison needs to include those objects, use `eqv?` below.

```
(eq? #t #t)               ⇒ #t
(eq? #t #f)               ⇒ #f
(eq? 'a 'a)               ⇒ #t
(eq? 'a 'b)               ⇒ #f
(eq? (list 'a) (list 'a)) ⇒ #f
(let ((x (list 'a)))
  (eq? x x))              ⇒ #t
```

`eqv?` *obj1 obj2* [Function]

[R7RS base] When *obj1* and *obj2* are both exact or both inexact numbers (except NaN), `eqv?` returns `#t` iff `(= obj1 obj2)` is true. When *obj1* and *obj2* are both characters, `eqv?` returns `#t` iff `(char=? obj1 obj2)` is true. Otherwise, `eqv?` is the same as `eq?` on Gauche.

```
(eqv? #\a #\a)             ⇒ #t
(eqv? #\a #\b)             ⇒ #f
(eqv? 1.0 1.0)             ⇒ #t
(eqv? 1 1)                 ⇒ #t
(eqv? 1 1.0)               ⇒ #f
(eqv? (list 'a) (list 'a)) ⇒ #f
(let ((x (list 'a)))
  (eqv? x x))              ⇒ #t
```

Note that comparison of NaNs has some peculiarity. Any numeric comparison fails if there's at least one NaN in its argument. Therefore, `(= +nan.0 +nan.0)` is always `#f`. However, Gauche *may* return `#t` for `(eq? +nan.0 +nan.0)` or `(eqv? +nan.0 +nan.0)`.

`equal?` *obj1 obj2* [Function]

[R7RS+] If *obj1* and *obj2* are both aggregate types, `equal?` compares its elements recursively. Otherwise, `equal?` behaves the same as `eqv?`.

If *obj1* and *obj2* are not `eqv?` to each other, not of builtin types, and the class of both objects are the same, `equal?` calls the generic function `object-equal?`. By defining the method, users can extend the behavior of `equal?` for user-defined classes.

```
(equal? (list 1 2) (list 1 2)) ⇒ #t
(equal? "abc" "abc")           ⇒ #t
(equal? 100 100)               ⇒ #t
(equal? 100 100.0)             ⇒ #f
```

Note: This procedure correctly handles the case when both *obj1* and *obj2* have cycles through pairs and vectors, as required by R6RS and R7RS. However, if the cycle involves user-defined classes, `equal?` may fail to terminate.

`object-equal?` *obj1 obj2* [Generic Function]

This generic function is called when `equal?` is called on the objects it doesn't know about. You can define this method on your class so that `equal?` can check equivalence. This method is supposed to return `#t` if *obj1* is equal to *obj2*, `#f` otherwise. If you want to check equivalence of elements recursively, do not call `object-equal?` directly; call `equal?` on each element.

```
(define-class <foo> ()
  ((x :init-keyword :x)
   (y :init-keyword :y)))

(define-method object-equal? ((a <foo>) (b <foo>))
  (and (equal? (slot-ref a 'x) (slot-ref b 'x))
       (equal? (slot-ref a 'y) (slot-ref b 'y))))

(equal? (make <foo> :x 1 :y (list 'a 'b))
        (make <foo> :x 1 :y (list 'a 'b)))
  ⇒ #t

(equal? (make <foo> :x 1 :y (make <foo> :x 3 :y 4))
        (make <foo> :x 1 :y (make <foo> :x 3 :y 4)))
  ⇒ #t
```

`object-equal?` (*obj1* <*top*>) (*obj2* <*top*>) [Method]

This method catches `equal?` between two objects of a user-defined class, in case the user doesn't define a specialized method for the class.

When called, it scans the registered default comparators that can handle both *obj1* and *obj2*, and if it finds one, use the comparator's equality predicate to see if two arguments are equal to each other. When no matching comparators are found, it just returns `#f`. See Section 6.2.4.3 [Predefined comparators], page 106, about the default comparators: Look for the entries of `default-comparator` and `comparator-register-default!`.

Note: If you define `object-equal?` with exactly the same specializers of this method, you'll replace it and that breaks `default-comparator` operation. Future versions of Gauche will prohibit such redefinition. For now, be careful not to redefine it accidentally.

Sometimes you want to test if two aggregate structures are topologically equal, i.e., if one has a shared substructure, the other has a shared substructure in the same way. `Equal?` can't handle it; module `util.isomorph` provides a procedure `isomorphic?` which does the job (see Section 12.65 [Determine isomorphism], page 797).

6.2.2 Comparison

Equality only concern about whether two objects are equivalent or not. However, sometimes we want to see the order among objects. Again, there's no single "universal order". It doesn't make mathematical sense to ask if one complex number is greater than another, but having some artificial order is useful when you want a consistent result of sorting a list of objects including numbers.

`compare` *obj1 obj2* [Function]

A general comparison procedure. Returns -1 if *obj1* is less than *obj2*, 0 if *obj1* is equal to *obj2*, and 1 if *obj1* is greater than *obj2*.

If *obj1* and *obj2* are incomparable, an error is signalled. However, `compare` defines total order between most Scheme objects, so that you can use it on wide variety of objects. The definition is upper-compatible to the order defined in `srfi-114`.

Some built-in types are handled by this procedure reflecting "natural" order of comparison if any (e.g. real numbers are compared by numeric values, characters are compared by `char<` etc.) For convenience, it also defines superficial order between objects that doesn't have natural order; complex numbers are ordered first by their real part, then their imaginary part, for example. That is, `1+i` comes before `2-i`, which comes before `2`, which comes before `2+i`.

Boolean false comes before boolean true.

Lists are ordered by dictionary order: Take the common prefix. If either one is `()` and the other is not, `()` comes first. If both tails are not empty, compare the heads of the tails. (This makes empty list the "smallest" of all lists).

Vectors (including uniform vectors) are compared first by their lengths, and if they are the same, elements are compared from left to right. Note that it's different from lists and strings.

```
(compare '(1 2 3) '(1 3))
  ⇒ -1  ; (1 2 3) is smaller
(compare '#(1 2 3) '#(1 3))
  ⇒ 1   ; #(1 3) is smaller
(compare "123" "13")
  ⇒ -1  ; "123" is smaller
```

If two objects are of subclasses of `<object>`, a generic function `object-compare` is called.

If two objects are of different types and at least one of them isn't `<object>`, then they are ordered by their types. `Srfi-114` defines the order of builtin types as follows:

1. Empty list.
2. Pairs.
3. Booleans.
4. Characters.
5. Strings.
6. Symbols.
7. Numbers.
8. Vectors.
9. Uniform vectors (u8 < s8 < u16 < s16 < u32 < s32 < u64 < s64 < f16 < f32 < f64)
10. All other objects.

`object-compare` *obj1 obj2* [Generic Function]

Specializing this generic function extends `compare` procedure for user-defined classes.

This method must return either `-1` (*obj1* precedes *obj2*), `0` (*obj1* equals to *obj2*), `1` (*obj1* succeeds *obj2*), or `#f` (*obj1* and *obj2* cannot be ordered).

`object-compare` *(obj1 <top>) (obj2 <top>)* [Method]

This method catches `compare` between two objects of a user-defined class, in case the user doesn't define a specialized method for the class.

When called, it scans the registered default comparators that can handle both *obj1* and *obj2*, and if it finds one, use the comparator's compare procedure to determine the order of *obj1* and *obj2*. When no matching comparators are found, it returns `#f`, meaning two objects can't be ordered. See Section 6.2.4.3 [Predefined comparators], page 106, about the default comparators: Look for the entries of `default-comparator` and `comparator-register-default!`.

Note: If you define `object-compare` with exactly the same specializers of this method, you'll replace it and that breaks `default-comparator` operation. Future versions of Gauche will prohibit such redefinition. For now, be careful not to redefine it accidentally.

`eq-compare` *obj1 obj2* [Function]

Returns -1 (less than), 0 (equal to) or 1 (greater than) according to a certain total ordering of *obj1* and *obj2*. Both arguments can be any Scheme objects, and can be different type of objects. The following properties are guaranteed.

- `(eq-compare x y)` is 0 iff `(eq? x y)` is `#t`.
- The result is consistent within a single run of the process (but may differ between runs).

Other than these, no actual semantics are given to the ordering.

This procedure is useful when you need to order arbitrary Scheme objects, but you don't care the actual order as far as it's consistent.

6.2.3 Hashing

Hash functions have close relationship with equality predicate, so we list them here.

`eq-hash` *obj* [Function]
`eqv-hash` *obj* [Function]

These are hash functions suitable to be used with `eq?` and `eqv?`, respectively. The returned hash value is system- and process-dependent, and can't be carried over the boundary of the running process.

Note: don't hash numbers by `eq-hash`. Two numbers are not guaranteed to be `eq?` even if they are numerically equal.

`default-hash` *obj* [Function]

[R7RS+] This is a hash function suitable to be used with `equal?`. In R7RS, this is defined in `scheme.comparator` (originally in `srfi-128`).

If *obj* is either a number, a boolean, a character, a symbol, a keyword, a string, a list, a vector or a uniform vector, internal hash function is used to calculate the hash value. If *obj* is other than that, a generic function `object-hash` is called to calculate the hash value (see below).

The hash value also depends on `hash-salt`, which differs for every run of the process.

`portable-hash` *obj salt* [Function]

Sometimes you need to calculate a hash value that's "portable", in a sense that the value won't change across multiple runs of the process, nor between different platforms. Such hash value can be used with storing objects externally to share among processes.

This procedure calculates a hash value of *obj* with such characteristics; the hash value is the same for the same object and the same salt value. Here "same object" roughly means having the same external representation. Objects `equal?` to each other are same. If you write out an object with `write`, and read it back, they are also the same objects in this sense.

This means objects without read/write invariance, such as ports, can't be handled with `portable-hash`. It is caller's responsibility that *obj* won't contain such objects.

The *salt* argument is a nonnegative fixnum and gives variations in the hash function. You have to use the same salt to get consistent results.

If *obj* is other than a number, a boolean, a character, a symbol, a keyword, a string, a list, a vector, or a uniform vector, this procedure calls a generic function `object-hash` is called to calculate the hash value (see below).

`legacy-hash` *obj* [Function]

Up to 0.9.4, Gauche had a hash function called `hash` that was used in both `equal?`-hashtable *and* for the portable hash function. It had a problem, though.

1. There was no way to salt the hash function, which makes the hashtables storing externally provided data vulnerable to collision attack.
2. The hash function behaves poorly, especially on flonums.
3. There are bugs in bignum and flonum hashing code that have produced different results on different architectures.

Since there are existing hash values calculated with the old hash function, we preserve the behavior of the original `hash` function as `legacy-hash`. Use this when you need to access old data. (The `hash` function also behaves as `legacy-hash` by default, but it has tweaks; see below.)

The new code that needs portable hash value should use `portable-hash` instead.

`object-hash` *obj rec-hash* [Generic Function]

By defining a method for this generic function, objects of user-defined types can have a hash value and can be used in a `equal?` hash table.

The method has to return an exact non-negative integer, and must return the same value for two object which are `equal?`. Furthermore, the returned value must not rely on the platform or state of the process, if *obj* is a portable object (see `portable-hash` above for what is portable.)

If the method needs to get hash value of *obj*'s elements, it has to call *rec-hash* on them. It guarantees that the proper hash function is called recursively. So you can count on *rec-hash* to calculate a portable hash value when `object-hash` itself is called from `portable-hash`.

If *obj* has several elements, you can call `combine-hash-value` on the elements' hash values.

```
(define-class <myclass> () (x y))

;; user-defined equality function
(define-method object-equal? ((a <myclass>) (b <myclass>))
  (and (equal? (ref a 'x) (ref b 'x))
       (= (abs (ref a 'y)) (abs (ref b 'y)))))

;; user-defined hash function
(define-method object-hash ((a <myclass>) rec-hash)
  (combine-hash-value (rec-hash (ref a 'x))
                      (rec-hash (abs (ref a 'y)))))
```

`object-hash` (*obj* `<top>`) *rec-hash* [Method]
`object-hash` (*obj* `<top>`) [Method]

These two methods are defined by the system and ensures the backward compatibility and the behavior of `default-comparator`. Be careful not to replace these methods by defining the exactly same specializers. In future versions of Gauche, attempts to replace these methods will raise an error.

`combine-hash-value` *ha hb* [Function]

Returns a hash value which is a combination of two hash values, *ha* and *hb*. The guaranteed invariance is that if `(= ha1 ha2)` and `(= hb1 hb2)` then `(= (combine-hash-value ha1 hb1) (combine-hash-value ha2 hb2))`. This is useful to write user-defined `object-hash` method.

`hash` *obj* [Function]

This function is deprecated.

Calculate a hash value of *obj* suitable for `equal?` hash. By default, it returns the same value as `legacy-hash`. However, if this is called from `default-hash` or `portable-hash` (via `object-hash` method), it recurses to the calling hash function.

The behavior is to keep the legacy code work. Until 0.9.5, `hash` is the only hash function to be used for both portable hash and `equal?`-hash, and `object-hash` method takes single argument (an object to hash) and calls `hash` recursively whenever it needs to get a hash value of other objects pointed from the argument.

As of 0.9.5 we have more than one hash functions that calls `object-hash`, so the method takes the hash function as the second argument to recurse. However, we can't just break the legacy code; so there's a default method defined in `object-hash` which is invoked when no two-arg method is defined for the given object, and dispatches to one-arg method. As far as the legacy `object-hash` code calls `hash`, it calls proper function. The new code shouldn't rely on this behavior, and must use the second argument of `object-hash` instead.

`boolean-hash` *bool* [Function]
`char-hash` *char* [Function]
`char-ci-hash` *char* [Function]
`string-hash` *str* [Function]
`string-ci-hash` *str* [Function]
`symbol-hash` *sym* [Function]
`number-hash` *num* [Function]

[R7RS comparator] These are hash functions for specific type of objects, defined in R7RS `scheme.comparator`. In Gauche, these procedures are just a wrapper of `default-hash` with type checks (and case folding when relevant). These are mainly provided to conform `scheme.comparator`; in your code you might just want to use `default-hash` (or `eq-hash`/`eqv-hash`, depending on the equality predicate).

The case-folding versions, `char-ci-hash` and `string-ci-hash`, calls `char-foldcase` and `string-foldcase` respectively, on the argument before passing it to `hash`. (See Section 6.10 [Characters], page 143, for `char-foldcase`. See Section 9.35.3 [Full string case conversion], page 475, for `string-foldcase`).

`hash-bound` [Function]
`hash-salt` [Function]

[R7RS comparator] Both evaluates to an exact nonnegative integers. In R7RS, these are defined in `scheme.comparator`.

(Note: `scheme.comparator` defines these as macros, in order to allow implementations optimize runtime overhead. In Gauche we use procedures but the overhead is negligible.)

User-defined hash functions can limit the range of the result between 0 and `(hash-bound)`, respectively, without worrying to lose quality of hash function. (User-defined hash functions don't need to honor `(hash-bound)` at all; hashtables takes modulo when necessary.)

User-defined hash function can also take into account of the value `(hash-salt)` into hash calculation; the salt value may differ between runs of the Scheme processes, or even between hash table instances. It is to avoid collision attack. Built-in hash functions already takes the salt value into account, so if your hash function is combining the hash values of primitive types, you don't need to worry about salt values.

6.2.4 Basic comparators

Equality and comparison procedures are parameters in various data structures. A treemap needs to order its keys; a hashtable needs to see if the keys are the same or not, and it also need a hash function consistent with the equality predicate.

If we want to work on generic data structures, we need to abstract those variations of comparison schemes. So here comes the comparator, a record that bundles closely-related comparison procedures together.

There are two SRFIs that define comparators. The one that was originally called `srfi-128` has now become a part of R7RS large as `scheme.comparator`, and we recommend new code to use it. Gauche has all of `scheme.comparator` procedures built-in. The older, and rather complex one is `srfi-114`; Gauche also supports it mainly for the backward compatibility. Importantly, Gauche's native `<comparator>` object is compatible to both `scheme.comparator` and `srfi-114` comparators.

6.2.4.1 Comparator class and constructors

`<comparator>` [Builtin Class]

A comparator record that bundles the following procedures:

Type test predicate
: Checks if an object can be compared with this comparator.

Equality predicate
: See if given two objects are equal to each other; returns a boolean value.

Ordering predicate
: Compare given two objects, and returns true iff the first one is strictly precedes the second one. That is, this is a less-than predicate.

Comparison procedure
: Compare given two objects, and returns either -1 (the first one is less than the second), 0 (they are equal), or 1 (the first one is greater than the second).

Hash function
: Returns a hash value of the given object.

`Scheme.comparator`'s comparators use the ordering predicate, while SRFI-114 comparators use the comparison procedure. Gauche's `<comparator>` supports both by automatically generating the missing one; that is, if you create a comparator with `scheme.comparator` interface, by giving an ordering predicate, Gauche automatically fills the comparison procedure, and if you create one with SRFI-114 interface by giving a comparison procedure, Gauche generates the ordering predicate.

A comparator may not have an ordering predicate / comparison procedure, and/or a hash function. You can check if the comparator can be used for ordering or hashing by `comparator-ordered?` and `comparator-hashable?`, respectively.

Some built-in data types such as hashtables (see Section 6.15 [Hashtables], page 177) and treemaps (see Section 6.16 [Treemaps], page 182), take a comparator in their constructors. The sort and merge procedures also accept comparators (see Section 6.24 [Sorting and merging], page 244).

`make-comparator` *type-test equal order hash :optional name* [Function]

[R7RS comparator] Creates a new comparator form the given *type-test*, *equal*, *order* and *hash* functions, and returns it. In R7RS, this is defined in `scheme.comparator`

See the description of `<comparator>` above for the role of those procedures.

Note: Both `scheme.comparator` and `srfi-114` defines `make-comparator`, but where `scheme.comparator` takes *order* argument, `srfi-114` takes *compare* argument. Since `scheme.comparator` is preferable, we adopt it for the built-in interface, and give a different name (`make-comparator/compare`) for SRFI-114 constructor.

Actually, some arguments can be non-procedures, to use predefined procedures, for the convenience. Even if non-procedure arguments are passed, the corresponding accessors (e.g. `comparator-type-test-procedure` for the `type-test` procedure) always return a procedure—either the given one or the predefined one.

The *type-test* argument must be either `#t` or a predicate taking one argument to test suitability of the object for comparing by the resulting comparator. If it is `#t`, a procedure that always return `#t` is used.

The *equal* argument must a predicate taking two arguments to test equality.

the *order* argument must be either `#f` or a procedure taking two arguments and returning a boolean value. It must return `#t` iff the first argument strictly precedes the second one. If `#f` is passed, the comparator can not be used for ordering.

The *hash* argument must be either `#f`, or a procedure taking one argument and returning nonnegative exact integer. If `#f` is given, it indicates the comparator can't hash objects; the predefined procedure just throws an error.

The fifth, optional argument *name*, is Gauche's extension. It can be any object but usually a symbol; it is only used when printing the comparator, to help debugging.

`make-comparator/compare` *type-test equal compare hash :optional name* [Function]

This is SRFI-114 comparator constructor. In SRFI-114, this is called `make-comparator`. Avoiding name conflict, we renamed it. If you `(use srfi-114)` you get the original name `make-comparator` (and the built-in `make-comparator` is shadowed). This is provided for the backward compatibility, and new code should use built-in `make-comparator` above.

It's mostly the same as `make-comparator` above, except the following:

- The third argument (*compare*) is a comparison procedure instead of an ordering predicate. It must be either `#f`, or a procedure taking two arguments and returning either -1, 0, or 1, depending on whether the first argument is less than, equal to, or greater than the second argument. If it is `#f`, it indicates the comparator can't order objects.
- You can pass `#t` to the *equal* argument when you give a comparison procedure. In that case, equality is determined by calling the comparison procedure and see if the result is 0.

6.2.4.2 Comparator predicates and accessors

`comparator?` *obj* [Function]

[R7RS comparator] Returns true iff *obj* is a comparator. In R7RS, this is provided from `scheme.comparator`.

`object-equal?` (*a* <*comparator*>) (*b* <*comparator*>) [Method]

Comparing two comparators by `equal?` compares their contents, via this method. Even *a* and *b* are comparators created separately, they can be `equal?` if all of their slots are the same.

This is Gauche's extension. The standard says nothing about equality of comparators, but it is sometimes useful if you can compare two.

```
(equal? (make-comparator #t equal? #f hash 'foo)
        (make-comparator #t equal? #f hash 'foo))
  ⇒ #t
```

```
;; The following may be #t or #f, depending on how the anonymous
;; procedure is allocated.
(equal? (make-comparator (^x x) eq? #f #f)
        (make-comparator (^x x) eq? #f #f))
```

`comparator-flavor` *cmpr* [Function]

Returns a symbol `ordering` if *cmpr* is created with `scheme.comparator` constructor, and returns `comparison` if *cmpr* is created with SRFI-114 constructor.

Usually applications don't need to distinguish these two kinds of comparators, for either kind of comparators can behave just as another kind. This procedure is for some particular cases when one wants to optimize for the underlying comparator implementation.

`comparator-ordered?` *cmpr* [Function]
`comparator-hashable?` *cmpr* [Function]

[R7RS comparator] Returns true iff a comparator *cmpr* can be used to order objects, or to hash them, respectively. In R7RS, this is provided from `scheme.comparator`.

`comparator-type-test-procedure` *cmpr* [Function]
`comparator-equality-predicate` *cmpr* [Function]
`comparator-ordering-predicate` *cmpr* [Function]
`comparator-hash-function` *cmpr* [Function]

[R7RS comparator] Returns type test procedure, equality predicate, ordering procedure and hash function of comparator *cmpr*, respectively. In R7RS, this is provided from `scheme.comparator`.

These accessors always return procedures; if you give `#f` to the *order* or *hash* argument of the constructor, `comparator-ordering-predicate` and `comparator-hash-function` still return a procedure, which will just raise an error.

`comparator-comparison-procedure` *cmpr* [Function]

[SRFI-114] This is a SRFI-114 procedure, but sometimes handy with `scheme.comparator` comparators. Returns a procedure that takes two objects that satisfy the type predicates of *cmpr*. The procedure returns either -1, 0 or 1, depending on whether the first object is less than, equal to, or greater than the second. The comparator must be ordered, that is, it must have an ordering predicate (or a comparison procedure, if it is created by SRFI-114 constructor).

`comparator-test-type` *cmpr obj* [Function]
`comparator-check-type` *cmpr obj* [Function]

[R7RS comparator] Test whether *obj* can be handled by a comparator *cmpr*, by applying *cmpr*'s type test predicate. The former (`comparator-test-type`) returns a boolean values, while the latter (`comparator-check-type`) signals an error when *obj* can't be handled.

In R7RS, this is provided from `scheme.comparator`.

`=?` *cmpr obj obj2 obj3 ...* [Function]
`<?` *cmpr obj obj2 obj3 ...* [Function]
`<=?` *cmpr obj obj2 obj3 ...* [Function]
`>?` *cmpr obj obj2 obj3 ...* [Function]
`>=?` *cmpr obj obj2 obj3 ...* [Function]

[R7RS comparator] Compare objects using a comparator *cmpr*. All of *obj*, *obj2*, *obj3 ...* must satisfy the type predicate of *cmpr*. When more than two objects are given, the order of comparison is undefined.

In order to use `<?`, `<=?`, `>?` and `>=?`, comparator must be ordered.

In R7RS, this is provided from `scheme.comparator`.

comparator-hash *cmpr obj* [Function]

[R7RS comparator] Returns a hash value of *obj* with the hash function of a comparator *cmpr*. The comparator must be hashable, and *obj* must satisfy comparator's type test predicate.

In R7RS, this is provided from `scheme.comparator`.

comparator-compare *cmpr a b* [Function]

[SRFI-114] Order two objects *a* and *b* using *cmpr*, and returns either one of -1 (*a* is less than *b*), 0 (*a* equals to *b*), or 1 (*a* is greater than *b*). Objects must satisfy *cmpr*'s type test predicate.

A simple comparison can be done by `<?` etc, but sometimes three-way comparison comes handy. So we adopt this procedure from srfi-114.

6.2.4.3 Predefined comparators

default-comparator [Variable]

[SRFI-114] This variable bounds to a comparator that is used by default in many context.

It can compare most of Scheme objects, even between objects with different types. In fact, it is defined as follows:

```
(define default-comparator
  (make-comparator/compare #t equal? compare default-hash
                           'default-comparator))
```

As you see in the definition, equality, ordering and hashing are handled by `equal?`, `compare` and `default-hash`, respectively. They takes care of builtin objects, and also `equal?` and `compare` handle the case when two objects () are of different types.

For objects of user-defined classes, those procedures call generic functions `object-equal?`, `object-compare`, and `object-hash`, respectively. Defining methods for them automatically extended the domain of `default-comparator`.

`Scheme.comparator` defines another way to extend `default-comparator`. See `comparator-register-default!` below for the details.

comparator-register-default! *comparator* [Function]

[R7RS comparator] In R7RS, this is provided from `scheme.comparator`. This is the `scheme.comparator` way for user programs to extend the behavior of the `default-comparator` (which is what `make-default-comparator` returns).

Note that, in Gauche, you can also extend default comparator's behavior by defining specialized methods for `object-equal?`, `object-compare` and `object-hash`. See the description of `default-comparator` above, for the details.

In fact, Gauche uses those generic functions to handle the registered comparators; methods specialized for `<top>` are defined for these generic functions, which catches the case when `default-comparator` is applied on object(s) of user-defined classes that don't have specialized methods defined for those generic functions. The catching method examines registered comparators to find one that can handle passed argument(s), and if it finds one, use it.

You might frown at this procedure having a global side-effect. Well, `scheme.comparator` explicitly prohibits comparators registered by this procedure alters the behavior of the default comparator in the existing domain—it is only allowed to handle objects that aren't already handled by the system's original default comparator and other already registered comparators. So, the only effect of adding new comparator should make the default comparator work on objects that had been previously raised an error.

In reality, it is impossible to enforce the condition. If you register a comparator whose domain overlaps overlaps the domain the default comparator (and its extensions via Gauche's methods), the program becomes non-portable at that moment. In the current version, the comparators registered by `comparator-register-default!` has the lowest precedence on the dispatch mechanism, but you shouldn't count on that.

`eq-comparator` [Variable]
`eqv-comparator` [Variable]
`equal-comparator` [Variable]

[SRFI-114] Built-in comparators that uses `eq?`, `eqv?` and `equal?` for the equality predicate, respectively. They accept any kind of Scheme objects. Each has corresponding hash functions (i.e. `eq-hash` for `eq-comparator`, `eqv-hash` for `eqv-comparator` and `default-hash` for `equal-comparator`). Only `eq-comparator` is ordered, using `eq-compare` to order the objects (see Section 6.2.2 [Comparison], page 99, for `eq-compare`).

Note that `eq-comparator` and `eqv-comparator` are not equivalent from what `make-eq-comparator` and `make-eqv-comparator` return, respectively. The latter two are defined in `scheme.comparator` and specified to use `default-hash` for the hash function. It is heavier than `eq-hash`/`eqv-hash`, and it can't be used for circular objects, nor for the mutable objects with which you want to hash them by identity. We provide `eq-comparator` and `eqv-comparator` in case you want to avoid limitations of `default-hash`.

`boolean-comparator` [Variable]
`char-comparator` [Variable]
`char-ci-comparator` [Variable]
`string-comparator` [Variable]
`string-ci-comparator` [Variable]

[SRFI-114] Compare booleans, characters, and strings, respectively. The `*-ci-*` variants uses case-insensitive comparison. All have appropriate hash functions, too.

The string case-insensitive comparison uses Unicode full-string case conversion (see Section 9.35.3 [Full string case conversion], page 475).

`exact-integer-comparator` [Variable]
`integer-comparator` [Variable]
`rational-comparator` [Variable]
`real-comparator` [Variable]
`complex-comparator` [Variable]
`number-comparator` [Variable]

[SRFI-114] Compare exact integers, integers, rational numbers, real numbers, complex numbers and general numbers, respectively. In Gauche `number-comparator` is the same as `complex-comparator`.

The equality are determined by `=`. For exact integer, integer, rational and real comparators, the order is the numerical order. Two complex numbers are compared first by their real components, and then their imaginary components only if the real components are the same.

Note that those comparator rejects NaN. You need `make-inexact-real-comparator` in `srfi-114` module to compare NaNs with your own discretion. See Section 11.24 [Comparators], page 626, for the details.

`pair-comparator` [Variable]
`list-comparator` [Variable]
`vector-comparator` [Variable]
`uvector-comparator` [Variable]
`bytevector-comparator` [Variable]

[SRFI-114] The default comparators to compare pairs, lists, vectors, uniform vectors and bytevectors (which is synonym to `u8vector`). Their respective elements are compared with the default comparators.

Note that lists are compared by dictionary order (`(1 2 3)` comes before `(1 3)`), while in vector-families shorter ones are ordered first (`#(1 3)` comes before `#(1 2 3)`).

6.2.4.4 Combining comparators

`make-default-comparator` [Function]

[R7RS comparator] Returns a default comparator. In Gauche, this returns the `default-comparator` object. In R7RS, this is provided from `scheme.comparator`.

`make-eq-comparator` [Function]
`make-eqv-comparator` [Function]

[R7RS comparator] Returns comparators that use `eq?` and `eqv?` for its equality predicate, respectively. Note that they use `default-hash` for hash functions, as specified by `scheme.comparator`, which has a few drawbacks: You can't use it if you want to hash based on identity of mutable objects, it diverges on circular objects, and it is slow if applied on a large structures. We recommend to use `eq-comparator` or `eqv-comparator` if possible (see Section 6.2.4.3 [Predefined comparators], page 106).

In R7RS, this is provided from `scheme.comparator`.

`make-reverse-comparator` *cmpr* [Function]

[SRFI-114] Returns a comparator with the same type test predicate, equality procedure, and hash function as the given comparator, but the comparison procedure is flipped.

`make-key-comparator` *cmpr test key* [Function]

Suppose you have some kind of structure, but you only need to look at one part of it to compare them.

Returns a new comparator that uses *test* as type test predicate. Its equality predicate, comparison procedure and hash function are constructed by applying *key* to the argument(s) then passing the result to the corresponding procedure of *cmpr*. If *cmpr* lacks comparison procedure and/or hash function, so does the returned comparator.

In the following example, the tree-map `users` compares the given user records only by the `username` slots:

```
(use gauche.record)

(define-record-type user #t #t
  username      ; string
  password-hash ; string
  comment)      ; string

(define users   ; table of users, managed by tree-map
  (make-tree-map
    (make-key-comparator string-comparator user? user-username)))
```

`make-tuple-comparator` *cmpr1 cmpr2 . . .* [Function]

Creates a comparator that compares lists of the form `(x1 x2 ...)`, where each element is compared with the corresponding comparator. For example, `(make-tuple-comparator c1 c2 c3)` will compare three-element list, whose first elements are compared by *c1*, second elements by *c2* and third elements by *c3*.

6.3 Numbers

Gauche supports the following types of numbers

multi-precision exact integer
: There's no limit of the size of number except the memory of the machine.

multi-precision exact non-integral rational numbers.
: Both denominator and numerator are represented by exact integers. There's no limit of the size of number except the memory of the machine.

inexact floating-point real numbers
: Using double-type of underlying C compiler, usually IEEE 64-bit floating point number.

inexact floating-point complex numbers
: Real part and imaginary part are represented by inexact floating-point real numbers.

6.3.1 Number classes

<number> [Builtin Class]
<complex> [Builtin Class]
<real> [Builtin Class]
<rational> [Builtin Class]
<integer> [Builtin Class]

These classes consist a class hierarchy of number objects. <complex> inherits <number>, <real> inherits <complex>,<rational> inherits <real> and <integer> inherits <rational>.

Note that these classes do not exactly correspond to the number hierarchy defined in R7RS. Especially, only exact integers are the instances of the <integer> class. That is,

```
(integer? 1)        ⇒ #t
(is-a? 1 <integer>) ⇒ #t
(is-a? 1 <real>)    ⇒ #t

(integer? 1.0)        ⇒ #t
(is-a? 1.0 <integer>) ⇒ #f
(is-a? 1.0 <real>)    ⇒ #t

(class-of (expt 2 100)) ⇒ #<class <integer>>
(class-of (sqrt -3)) ⇒ #<class <complex>>
```

6.3.2 Numerical predicates

number? *obj* [Function]
complex? *obj* [Function]
real? *obj* [Function]
rational? *obj* [Function]
integer? *obj* [Function]

[R7RS base] Returns #t if *obj* is a number, a complex number, a real number, a rational number or an integer, respectively. In Gauche, a set of numbers is the same as a set of complex numbers. A set of rational numbers is the same as a set of real numbers, except +inf.0, -inf.0 and +nan.0 (since we have only limited-precision floating numbers).

```
(complex? 3+4i)   ⇒ #t
(complex? 3)      ⇒ #t
(real? 3)         ⇒ #t
(real? -2.5+0.0i) ⇒ #t
(real? #e1e10)    ⇒ #t
(integer? 3+0i)   ⇒ #t
(integer? 3.0)    ⇒ #t

(real? +inf.0)     ⇒ #t
```

```
(real? +nan.0)     ⇒ #t
(rational? +inf.0) ⇒ #f
(rational? +nan.0) ⇒ #f
```

Note: R6RS adopts more strict definition on exactness, and notably, it defines a complex number with non-exact zero imaginary part is not a real number. Currently Gauche doesn't have exact complex numbers, and automatically coerces complex numbers with zero imaginary part to a real number. Thus R6RS code that relies on the fact that `(real? 1+0.0i)` is `#f` won't work with Gauche.

`real-valued?` *obj* [Function]
`rational-valued?` *obj* [Function]
`integer-valued?` *obj* [Function]

[R6RS] In Gauche these are just an alias of `real?`, `rational?` and `integer?`. They are provided for R6RS compatibility.

The difference of those and non `-valued` versions in R6RS is that these returns `#t` if *obj* is a complex number with nonexact zero imaginary part. Since Gauche doesn't distinguish complex numbers with zero imaginary part and real numbers, we don't have the difference.

`exact?` *obj* [Function]
`inexact?` *obj* [Function]

[R7RS base] Returns `#t` if *obj* is an exact number and an inexact number, respectively.

```
(exact? 1)       ⇒ #t
(exact? 1.0)     ⇒ #f
(inexact? 1)     ⇒ #f
(inexact? 1.0)   ⇒ #t

(exact? (modulo 5 3)) ⇒ #t
(inexact? (modulo 5 3.0)) ⇒ #f
```

`exact-integer?` *obj* [Function]

[R7RS base] Same as `(and (exact? obj) (integer? obj))`, but more efficient.

`zero?` *z* [Function]

[R7RS base] Returns `#t` if a number *z* equals to zero.

```
(zero? 1)        ⇒ #f
(zero? 0)        ⇒ #t
(zero? 0.0)      ⇒ #t
(zero? 0.0+0.0i) ⇒ #t
```

`positive?` *x* [Function]
`negative?` *x* [Function]

[R7RS base] Returns `#t` if a real number *x* is positive and negative, respectively. It is an error to pass a non-real number.

`finite?` *z* [Function]
`infinite?` *z* [Function]
`nan?` *z* [Function]

[R7RS inexact] For real numbers, returns `#f` iff the given number is finite, infinite, or NaN, respectively.

For non-real complex numbers, `finite?` returns `#t` iff both real and imaginary components are finite, `infinite?` returns `#t` if at least either real or imaginary component is infinite, and `nan?` returns `#t` if at least either real or imaginary component is NaN. (Note: It is incompatible to R6RS, in which these procedures must raise an error if the given argument is non-real number.)

In R7RS, these procedures are in `(scheme inexact)` library.

`odd?` *n* [Function]
`even?` *n* [Function]

[R7RS base] Returns `#t` if an integer *n* is odd and even, respectively. It is an error to pass a non-integral number.

```
(odd? 3)      ⇒ #t
(even? 3)     ⇒ #f
(odd? 3.0)    ⇒ #t
```

`fixnum?` *n* [Function]
`bignum?` *n* [Function]

[R7RS fixnum] Returns `#t` iff *n* is an exact integer whose internal representation is *fixnum* and *bignum*, respectively. R7RS-large defines `fixnum?` in `scheme.fixnum` library; `bignum?` is Gauche's extension. Portable Scheme programs don't need to care about the internal representation of integer. These are for certain low-level routines that does particular optimization.

`flonum?` *x* [Function]

[R7RS flonum] Returns `#t` if *x* is a number represented by a floating-point number, `#f` otherwise. In Gauche, inexact real numbers are flonums.

See Section 10.3.21 [R7RS flonum], page 581, for comprehensive flonum library.

6.3.3 Numerical comparison

`=` *z1 z2 z3 . . .* [Function]

[R7RS base] If all the numbers *z* are equal, returns `#t`.

```
(= 2 2)          ⇒ #t
(= 2 3)          ⇒ #f
(= 2 2.0)        ⇒ #t
(= 2 2.0 2.0+0i) ⇒ #t
(= 2/4 1/2)      ⇒ #t
```

`<` *x1 x2 x3 . . .* [Function]
`<=` *x1 x2 x3 . . .* [Function]
`>` *x1 x2 x3 . . .* [Function]
`>=` *x1 x2 x3 . . .* [Function]

[R7RS base] Returns `#t` If all the real numbers *x* are monotonically increasing, monotonically nondecreasing, monotonically decreasing, or monotonically nonincreasing, respectively.

`max` *x1 x2 . . .* [Function]
`min` *x1 x2 . . .* [Function]

[R7RS base] Returns a maximum or minimum number in the given real numbers, respectively. If any of the arguments are NaN, NaN is returned.

See also `find-min` and `find-max` in Section 9.5.2 [Selection and searching in collection], page 347.

`min&max` *x1 x2 . . .* [Function]

Returns a maximum and minimum number in the given real numbers.

See also `find-min&max` in Section 9.5.2 [Selection and searching in collection], page 347.

`approx=?` *x y :optional relative-tolerance absolute-tolerance* [Function]

Returns `#t` iff two numbers are approximately equal within the given error tolerance.

- If at least one of *x* or *y* is NaN, returns `#f`.
- If either one is infinity, returns `#t` iff the other one is also infinity of the same sign.
- Otherwise, return a boolean value computed as follows:

```
(<= (abs (- x y))
    (max (* (max (abs x) (abs y)) relative-tolerance)
         absolute-tolerance))
```

If at least one of *x* or *y* are non-real complex number, `magnitude` is used in place of `abs`.

When omitted, *relative-tolerance* is assumed to be `(flonum-epsilon)`, and *absolute-tolerance* is `(flonum-min-denormalized)`. That is, by default, `approx=?` tolerates 1 ULP (unit in the last place) error.

The *absolute-tolerance* argument is useful when arguments are close to zero, in which case relative tolerance becomes too small.

`flonum-epsilon` [Function]
`flonum-min-normalized` [Function]
`flonum-min-denormalized` [Function]

Returns flonums with the following characteristics, respectively:

`flonum-epsilon`
: Returns the least positive flonum e such that, for a normalized flonum `x`, `x` and `(* x (+ 1.0 e))` are distinguishable.

`flonum-min-normalized`
: Returns the least positive flonum representable as normalized floating-point number.

`flonum-min-denormalized`
: Returns the least positive flonum representable as denormalized floating-point number. If the platform doesn't support denormalized flonum, it returns the least positive normalized floating number.

6.3.4 Arithmetics

`+` *z . . .* [Function]
`*` *z . . .* [Function]

[R7RS base] Returns the sum or the product of given numbers, respectively. If no argument is given, `(+)` yields 0 and `(*)` yields 1.

`-` *z1 z2 . . .* [Function]
`/` *z1 z2 . . .* [Function]

[R7RS base] If only one number *z1* is given, returns its negation and reciprocal, respectively.

If more than one number are given, returns:

```
z1 - z2 - z3 ...
z1 / z2 / z3 ...
```

respectively.

```
(- 3)       ⇒ -3
(- -3.0)    ⇒ 3.0
(- 5+2i)    ⇒ -5.0-2.0i
(/ 3)       ⇒ 1/3
(/ 5+2i)    ⇒ 0.172413793103448-0.0689655172413793i
```

```
(- 5 2 1)       ⇒ 2
(- 5 2.0 1)     ⇒ 2.0
(- 5+3i -i)     ⇒ 5.0+2.0i
(/ 14 6)        ⇒ 7/3
(/ 6+2i 2)      ⇒ 3.0+1.0i
```

Note: Gauche didn't have exact rational number support until 0.8.8; before that, `/` coerced the result to inexact even if both divisor and dividend were exact numbers, when the result wasn't a whole number. It is not the case anymore.

If the existing code relies on the old behavior, it runs very slowly on the newer versions of Gauche, since the calculation proceeds with exact rational arithmetics that is much slower than floating point arithmetics. You want to use `/.` below to use fast inexact arithmetics (unless you need exact results).

`+.` *z ...* [Function]
`*.` *z ...* [Function]
`-.` *z1 z2 ...* [Function]
`/.` *z1 z2 ...* [Function]

Like `+`, `*`, `-`, and `/`, but the arguments are coerced to inexact number. So they always return inexact number. These are useful when you know you don't need exact calculation and want to avoid accidental overhead of bignums and/or exact rational numbers.

`abs` *z* [Function]

[R7RS+] For real number *z*, returns an absolute value of it. For complex number *z*, returns the magnitude of the number. The complex part is Gauche extension.

```
(abs -1)   ⇒ 1
(abs -1.0) ⇒ 1.0
(abs 1+i)  ⇒ 1.4142135623731
```

`quotient` *n1 n2* [Function]
`remainder` *n1 n2* [Function]
`modulo` *n1 n2* [Function]

[R7RS base] Returns the quotient, remainder and modulo of dividing an integer *n1* by an integer *n2*. The result is an exact number only if both *n1* and *n2* are exact numbers.

Remainder and modulo differ when either one of the arguments is negative. Remainder *R* and quotient *Q* have the following relationship.

```
n1 = Q * n2 + R
```

where `abs(Q) = floor(abs(n1)/abs(n2))`. Consequently, *R*'s sign is always the same as *n1*'s.

On the other hand, modulo works as expected for positive *n2*, regardless of the sign of *n1* (e.g. `(modulo -1 n2) == n2 - 1`). If *n2* is negative, it is mapped to the positive case by the following relationship.

```
modulo(n1, n2) = -modulo(-n1, -n2)
```

Consequently, *modulo*'s sign is always the same as *n2*'s.

```
(remainder 10 3)    ⇒ 1
(modulo 10 3)       ⇒ 1

(remainder -10 3)   ⇒ -1
(modulo -10 3)      ⇒ 2

(remainder 10 -3)   ⇒ 1
```

```
(modulo 10 -3)       ⇒ -2

(remainder -10 -3)   ⇒ -1
(modulo -10 -3)      ⇒ -1
```

quotient&remainder *n1 n2* [Function]

Calculates the quotient and the remainder of dividing integer *n1* by integer *n2* simultaneously, and returns them as two values.

div *x y* [Function]
mod *x y* [Function]
div-and-mod *x y* [Function]
div0 *x y* [Function]
mod0 *x y* [Function]
div0-and-mod0 *x y* [Function]

[R6RS] These are integer division procedures introduced in R6RS. Unlike `quotient`, `modulo` and `remainder`, these procedures can take non-integral values. The dividend *x* can be an arbitrary real number, and the divisor *y* can be non-zero real number.

`div` returns an integer *n*, and `mod` returns a real number *m*, such that:

- $x = n\ y$ + m, and
- 0 <= m < |y|.

Examples:

```
(div 123 10)      ⇒ 12
(mod 123 10)      ⇒ 3

(div 123 -10)     ⇒ -12
(mod 123 -10)     ⇒ 3

(div -123 10)     ⇒ -13
(mod -123 10)     ⇒ 7

(div -123 -10)    ⇒ 13
(mod -123 -10)    ⇒ 7

(div 123/7 10/9) ⇒ 15
(mod 123/7 10/9) ⇒ 19/21
;; 123/7 = 10/9 * 15 + 19/21

(div 14.625 3.75) ⇒ 3.0
(mod 14.625 3.75) ⇒ 3.375
;; 14.625 = 3.75 * 3.0 + 3.375
```

For a nonnegative integer *x* and an integer *y*, The results of `div` and `mod` matches those of `quotient` and `remainder`. If *x* is negative, they differ, though.

`div-and-mod` calculates both `div` and `mod` and returns their results in two values.

div0 and *mod0* are similar, except the range of *m*:

- $x = n\ y$ + m
- -|y|/2 <= m < |y|/2

```
(div0 123 10)    ⇒ 12
(mod0 123 10)    ⇒ 3
```

```
(div0 127 10)    ⇒ 13
(mod0 127 10)    ⇒ -3

(div0 127 -10)   ⇒ -13
(mod0 127 -10)   ⇒ -3

(div0 -127 10)   ⇒ -13
(mod0 -127 10)   ⇒ 3

(div0 -127 -10)  ⇒ 13
(mod0 -127 -10)  ⇒ 3
```

`div0-and-mod0` calculates both `div0` and `mod0` and returns their results in two values.

Here's a visualization of R6RS and R7RS division and modulo operations: `http://blog.practical-scheme.net/gauche/20100618-integer-divisions` It might help to grasp how they works.

`floor-quotient` *n d* [Function]
`floor-remainder` *n d* [Function]
`floor/` *n d* [Function]
`truncate-quotient` *n d* [Function]
`truncate-remainder` *n d* [Function]
`truncate/` *n d* [Function]

[R7RS base] These are integer division operators introduced in R7RS. The names explicitly indicate how they behave when numerator and/or denominator is/are negative.

The arguments *n* and *d* must be an integer. If any of them are inexact, the result is inexact. If all of them are exact, the result is exact. Also, *d* must not be zero.

Given numerator *n*, denominator *d*, quotient *q* and remainder *r*, the following relations are always kept.

```
r = n - dq
abs(r) < abs(d)
```

Now, `(floor-quotient n d)` and `(truncate-quotient n d)` are the same as `(floor (/ n d))` and `(truncate (/ n d))`, respectively. The `*-remainder` counterparts are derived from the above relation.

The `/`-suffixed version, `floor/` and `truncate/`, returns corresponding quotient and remainder as two values.

```
(floor-quotient 10 -3)      ⇒ -4
(floor-remainder 10 -3)     ⇒ -2
(truncate-quotient 10 -3)   ⇒ -3
(truncate-remainder 10 -3)  ⇒ 1
```

R7RS division library (`scheme.division`) introduces other variation of integer divisions (see Section 10.3.18 [R7RS integer division], page 574).

`gcd` *n* . . . [Function]
`lcm` *n* . . . [Function]

[R7RS base] Returns the greatest common divisor or the least common multiplier of the given integers, respectively

Arguments must be integers, but doesn't need to be exact. If any of arguments is inexact, the result is inexact.

`continued-fraction` *x* [Function]

Returns a lazy sequence of regular continued fraction expansion of finite real number *x*. An error is raised if *x* is infinite or NaN, or not a real number. The returned sequence is lazy, so the terms are calculated as needed.

```
(continued-fraction 13579/2468)
  ⇒ (5 1 1 122 1 9)

(+ 5 (/ (+ 1 (/ (+ 1 (/ (+ 122 (/ (+ 1 (/ 9))))))))))
  ⇒ 13579/2468

(continued-fraction (exact 3.141592653589793))
  ⇒ (3 7 15 1 292 1 1 1 2 1 3 1 14 3 3 2 1 3 3 7 2 1 1 3 2 42 2)

(continued-fraction 1.5625)
  ⇒ (1.0 1.0 1.0 3.0 2.0)
```

`numerator` *q* [Function]

`denominator` *q* [Function]

[R7RS base] Returns the numerator and denominator of a rational number *q*.

`rationalize` *x ebound* [Function]

[R7RS base] Returns the simplest rational approximation *q* of a real number *x*, such that the difference between *x* and *q* is no more than the error bound *ebound*.

Note that Gauche doesn't have inexact rational number, so if *x* and/or *ebound* is inexact, the result is coerced to floating point representation. If you want an exact result, coerce the arguments to exact number first.

```
(rationalize 1234/5678 1/1000) ⇒ 5/23

(rationalize 3.141592653589793 1/10000)
  ⇒ 3.141509433962264
(rationalize (exact 3.141592653589793) 1/10000)
  ⇒ 333/106
(rationalize (exact 3.141592653589793) 1/10000000)
  ⇒ 75948/24175

;; Some edge cases
(rationalize 2 +inf.0) ⇒ 0
(rationalize +inf.0 0) ⇒ +inf.0
(rationalize +inf.0 +inf.0) ⇒ +nan.0
```

`floor` *x* [Function]

`ceiling` *x* [Function]

`truncate` *x* [Function]

`round` *x* [Function]

[R7RS base] The argument *x* must be a real number. `Floor` and `ceiling` return a maximum integer that isn't greater than *x* and a minimum integer that isn't less than *x*, respectively. `Truncate` returns an integer that truncates *x* towards zero. `Round` returns an integer that is closest to *x*. If fractional part of *x* is exactly 0.5, `round` returns the closest even integer.

Following Scheme's general rule, the result is inexact if *x* is an inexact number; e.g. `(round 2.3)` is `2.0`. If you need an exact integer by rounding an inexact number, you have to use `exact` on the result, or use one of the following procedure (`(floor->exact` etc).

`floor->exact` *x* [Function]
`ceiling->exact` *x* [Function]
`truncate->exact` *x* [Function]
`round->exact` *x* [Function]

These are convenience procedures of the popular phrase `(exact (floor x))` etc.

`clamp` *x :optional min max* [Function]

Returns

```
min if x < min
x   if min <= x <= max
max if max < x
```

If *min* or *max* is omitted or `#f`, it is regarded as `-inf.0` or `+inf.0`, respectively. Returns an exact integer only if all the given numbers are exact integers.

```
(clamp 3.1 0.0 1.0) ⇒ 1.0
(clamp 0.5 0.0 1.0) ⇒ 0.5
(clamp -0.3 0.0 1.0) ⇒ 0.0
(clamp -5 0)         ⇒ 0
(clamp 3724 #f 256) ⇒ 256
```

`exp` *z* [Function]
`log` *z* [Function]
`log` *z1 z2* [Function]
`sin` *z* [Function]
`cos` *z* [Function]
`tan` *z* [Function]
`asin` *z* [Function]
`acos` *z* [Function]
`atan` *z* [Function]
`atan` *y x* [Function]

[R7RS inexact] Transcendental functions. Work for complex numbers as well. In R7RS, these procedures are in the `(scheme inexact)` module.

The two-argument version of `log` is added in R6RS, and returns base-*z2* logarithm of *z1*.

The two-argument version of `atan` returns `(angle (make-rectangular x y))` for the real numbers `x` and `y`.

`sinh` *z* [Function]
`cosh` *z* [Function]
`tanh` *z* [Function]
`asinh` *z* [Function]
`acosh` *z* [Function]
`atanh` *z* [Function]

Hyperbolic trigonometric functions. Work for complex numbers as well.

`sqrt` *z* [Function]

[R7RS inexact] Returns a square root of a complex number *z*. The branch cut scheme is the same as Common Lisp. For real numbers, it returns a positive root.

If *z* is the square of an exact real number, the return value is also an exact number.

```
(sqrt 2)       ⇒ 1.4142135623730951
(sqrt -2)      ⇒ 0.0+1.4142135623730951i
(sqrt 256)     ⇒ 16
(sqrt 256.0)   ⇒ 16.0
(sqrt 81/169) ⇒ 9/13
```

`exact-integer-sqrt` *k* [Function]

[R7RS base] Given an exact nonnegative integer *k*, returns two exact nonnegative integer *s* and *r* that satisfy the following equations:

```
k = (+ (* s s) r)
k < (* (+ s 1) (+ s 1))
```

```
(exact-integer-sqrt 782763574)
  ⇒ 27977 and 51045
```

`square` *z* [Function]

[R7RS base] Returns `(* z z)`.

`expt` *z1 z2* [Function]

[R7RS base] Returns *z1*^*z2* (*z1* powered by *z2*), where *z1* and *z2* are complex numbers.

Scheme standard defines `(expt 0 0)` as 1 for convenience.

`expt-mod` *base exponent mod* [Function]

Calculates `(modulo (expt base exponent) mod)` efficiently.

The next example shows the last 10 digits of a mersenne prime M_74207281 (2^74207281 - 1)

```
(- (expt-mod 2 74207281 #e1e10) 1)
 ⇒ 1086436351
```

`gamma` *x* [Function]
`lgamma` *x* [Function]

Gamma function and natural logarithmic of absolute value of Gamma function.

NB: Mathematically these functions are defined in complex domain, but currently we only supports real number argument.

`fixnum-width` [Function]
`greatest-fixnum` [Function]
`least-fixnum` [Function]

[R6RS] These procedures return the width of fixnum (*w*), the greatest integer representable by fixnum (`2^(w-1) - 1`), and the least integer representable by fixnum (`- 2^(w-1)`), respectively. You might want to care the fixnum range when you are writing a performance-critical section.

These names are defined in R6RS. Common Lisp and ChezScheme have `most-positive-fixnum` and `most-negative-fixnum`.

NB: Before 0.9.5, `fixnum-width` had a bug to return one smaller than the supposed value.

6.3.5 Numerical conversions

`make-rectangular` *x1 x2* [Function]
`make-polar` *x1 x2* [Function]

[R7RS complex] Creates a complex number from two real numbers, *x1* and *x2*. `make-rectangular` returns *x1* + **i***x2*. `make-polar` returns *x1***e**^(**i***x2*).

In R7RS, these procedures are in the `(scheme complex)` library.

`real-part` *z* [Function]
`imag-part` *z* [Function]
`magnitude` *z* [Function]

angle *z* [Function]

[R7RS complex] Decompose a complex number *z* and returns a real number. `real-part` and `imag-part` return *z*'s real and imaginary part, respectively. `magnitude` and `angle` return *z*'s magnitude and angle, respectively.

In R7RS, these procedures are in the `(scheme complex)` library.

decode-float *x* [Function]

For a given finite floating-point number, returns a vector of three exact integers, `#(m, e, sign)`, where

```
x = (* sign m (expt 2.0 e))
sign is either 1, 0 or -1.
```

If *x* is `+inf.0` or `-inf.0`, *m* is `#t`. If *x* is `+nan.0`, *m* is `#f`.

The API is taken from ChezScheme.

```
(decode-float 3.1415926)
 ⇒ #(7074237631354954 -51 1)
(* 7074237631354954 (expt 2.0 -51))
 ⇒ 3.1415926

(decode-float +nan.0)
 ⇒ #(#f 0 -1)
```

encode-float *vector* [Function]

This is an inverse of `decode-float`. *Vector* must be a three-element vector as returned from `decode-float`.

```
(encode-float '#(7074237631354954 -51 1))
 ⇒ 3.1415926

(encode-float '#(#t 0 1))
 ⇒ +inf.0
```

fmod *x y* [Function]
modf *x* [Function]
frexp *x* [Function]
ldexp *x n* [Function]

[POSIX] These procedures can be used to compose and decompose floating point numbers. `Fmod` computes the remainder of dividing *x* by *y*, that is, it returns *x*-*n***y* where *n* is the quotient of *x*/*y* rounded towards zero to an integer. `Modf` returns two values; a fractional part of *x* and an integral part of *x*. `Frexp` returns two values, *fraction* and *exponent* of *x*, where *x* = *fraction* * 2^*exponent*, and 0.5 <= |*fraction*| < 1.0, unless *x* is zero. (When *x* is zero, both *fraction* and *exponent* are zero). *Ldexp* is a reverse operation of `frexp`; it returns a real number *x* * 2^*n*.

```
(fmod 32.1 10.0)  ⇒ 2.1
(fmod 1.5 1.4)    ⇒ 0.1
(modf 12.5)       ⇒ 0.5 and 12.0
(frexp 3.14)      ⇒ 0.785 and 2
(ldexp 0.785 2)   ⇒ 3.14
```

exact *z* [Function]
inexact *z* [Function]

[R7RS base] Returns an exact or an inexact representation of the given number *z*, respectively. Passing an exact number to `exact`, and an inexact number to `inexact`, are no-op.

Gauche doesn't have exact complex number with non-zero imaginary part, nor exact infinites and NaNs, so passing those to `exact` raises an error.

```
(inexact 1)    ⇒ 1.0
(inexact 1/10) ⇒ 0.1
```

If an inexact finite real number is passed to `exact`, the simplest exact rational number within the precision of the floating point representation is returned.

```
(exact 1.0)      ⇒ 1
(exact 0.1)      ⇒ 1/10
(exact (/ 3.0))  ⇒ 1/3
```

For all finite inexact real number *x*, `(inexact (exact x))` is always `eqv?` to the original number *x*.

(Note that the inverse doesn't hold, that is, an exact number *n* and `(exact (inexact n))` aren't necessarily the same. It's because many (actually, infinite number of) exact numbers can be mapped to one inexact number.)

To specify the error tolerance when converting inexact real numbers to exact rational numbers, use `rationalize` or `real->rational`.

`exact->inexact` *z* [Function]
`inexact->exact` *z* [Function]

[R5RS] Converts exact number to inexact one, and vice versa.

In fact, `exact->inexact` returns the argument as is if an inexact number is passed, and `inexact->exact` returns the argument if an exact number is passed, so in Gauche they are equivalent to `inexact` and `exact`, respectively. Note that other R5RS implementation may raise an error if passing an inexact number to `exact->inexact`, for example.

Generally `exact` and `inexact` are preferred, for they are more concise, and you don't need to care whether the argument is exact or inexact numbers. These procedures are for compatibility with R5RS programs.

`real->rational` *x :optional hi lo open?* [Function]

Find the simplest rational representation of a finite real number *x* within the specified error bounds. This is the low-level routine called by `rationalize` and `exact`. Typically you want to use `rationalize` (see Section 6.3.4 [Arithmetics], page 112) for this purpose. Use `real->rational` only when you need finer control of error bounds.

The result rational value *r* satisfies the following condition:

```
(<= (- x lo) r (+ x hi))   ; when open? is #f
(<  (- x lo) r (+ x hi))   ; otherwise
```

Note that both *hi* and *lo* must be nonnegative.

If *hi* and/or *lo* is omitted, it is determined by *x*: if *x* is exact, *hi* and *lo* are defaulted to zero; if *x* is inexact, *hi* and *lo* depend on the precision of the floating point representation of *x*. In the latter case, the *open?* also depends on *x*—it is true if the mantissa of *x* is odd, and false otherwise, reflecting the round-to-even rule. So, if you call `real->rational` with one finite number, you'll get the same result as `exact`:

```
(real->rational 0.1) ⇒ 1/10
```

Passing zeros to the error bounds makes it return the exact conversion of the floating number itself (that is, the exact calculation of `(* sign mantissa (expt 2 exponent))`).

```
(real->rational 0.1 0 0) ⇒ 3602879701896397/36028797018963968
```

(If you give both *hi* and *lo*, but omit *open?*, we assume closed range.)

`number->string` *z :optional radix use-upper?* [Function]
`string->number` *string :optional radix* [Function]

[R7RS+] These procedures convert a number and its string representation in radix *radix* system. *radix* must be between 2 and 36 inclusive. If *radix* is omitted, 10 is assumed.

`Number->string` takes a number *z* and returns a string. If *z* is not an exact integer, *radix* must be 10. For the numbers with radix more than 10, lower case alphabet character is used for digits, unless the optional argument *use-upper?* is true, in that case upper case characters are used. The argument *use-upper?* is Gauche's extension.

`String->number` takes a string *string* and parses it as a number in radix *radix* system. If the number looks like non-exact number, only radix 10 is allowed. If the given string can't be a number, `#f` is returned.

`x->number` *obj* [Generic Function]
`x->integer` *obj* [Generic Function]

Generic coercion functions. Returns 'natural' interpretation of *obj* as a number or an exact integer, respectively. The default methods are defined for numbers and strings; a string is interpreted by `string->number`, and if the string can't be interpreted as a number, 0 is returned. Other *obj* is simply converted to 0. If *obj* is naturally interpreted as a number that is not an exact integer, `x->integer` uses `round` and `inexact->exact` to obtain an integer.

Other class may provide a method to customize the behavior.

6.3.6 Basic bitwise operations

These procedures treat integers as half-open bit vectors. If an integer is positive, it is regarded as if infinite number of zeros are padded to the left. If an integer is negative, it is regarded in 2's complement form, and infinite number of 1's are padded to the left.

In regard to the names of those operations, there are two groups in the Scheme world; Gauche follows the names of the original SLIB's "logical" module, which was rooted in CL. Another group uses a bit long but descriptive name such as `arithmetic-shift`.

R7RS bitwise library (see Section 10.3.19 [R7RS bitwise operations], page 575) provides additional bitwise operations.

`ash` *n count* [Function]

[SRFI-60] Shifts integer *n* left with *count* bits. If *count* is negative, `ash` shifts *n* right with −*count* bits.

```
; Note: 6  ≡ [...00110], and
;      -6  ≡ [...11010]
(ash 6 2)    ⇒ 24  ;[...0011000]
(ash 6 -2)   ⇒ 1   ;[...0000001]
(ash -6 2)   ⇒ -24 ;[...1101000]
(ash -6 -2)  ⇒ -2  ;[...1111110]
```

`logand` *n1 . . .* [Function]
`logior` *n1 . . .* [Function]
`logxor` *n1 . . .* [Function]

[SRFI-60] Returns bitwise and, bitwise inclusive or and bitwise exclusive or of integers *n1 . . .* . If no arguments are given, `logand` returns `-1`, and `logior` and `logxor` returns `0`.

`lognot` *n* [Function]

[SRFI-60] Returns bitwise not of an integer *n*.

`logtest` *n1 n2 . . .* [Function]

[SRFI-60] ≡ `(not (zero? (logand n1 n2 ...)))`

`logbit?` *index n* [Function]

[SRFI-60] Returns `#t` if *index*-th bit of integer *n* is 1, `#f` otherwise.

`bit-field` *n start end* [Function]

[R7RS bitwise] Extracts *start*-th bit (inclusive) to *end*-th bit (exclusive) from an exact integer *n*, where *start* `<` *end*.

`copy-bit` *index n bit* [Function]

[R7RS bitwise] If *bit* is true, sets *index*-th bit of an exact integer *n*. If *bit* is false, resets *index*-th bit of an exact integer *n*.

`copy-bit-field` *n from start end* [Function]

[SRFI-60] Returns an exact integer, each bit of which is the same as *n* except the *start*-th bit (inclusive) to *end*-th bit (exclusive), which is a copy of the lower `(end-start)`-th bits of an exact integer *from*.

```
(number->string (copy-bit-field #b10000000 -1 1 5) 2)
  ⇒ "10011110"

(number->string (copy-bit-field #b10000000 #b010101010 1 7) 2)
  ⇒ "11010100"
```

Note: The API of this procedure was originally taken from SLIB, and at that time, the argument order was `(copy-bit-field n start end from)`. During the discussion of SRFI-60 the argument order was changed for the consistency, and the new versions of SLIB followed it. We didn't realize the change until recently - before 0.9.4, *this procedure had the old argument order*. Code that is using this procedure needs to be fixed. If you need your code to work with both versions of Gauche, have the following definition in your code.

```
(define (copy-bit-field to from start end)
  (if (< start end)
    (let1 mask (- (ash 1 (- end start)) 1)
      (logior (logand to (lognot (ash mask start)))
              (ash (logand from mask) start)))
    from))
```

`logcount` *n* [Function]

[SRFI-60] If *n* is positive, returns the number of `1`'s in the bits of *n*. If *n* is negative, returns the number of `0`'s in the bits of 2's complement representation of *n*.

```
(logcount 0)      ⇒ 0
(logcount #b0010) ⇒ 1
(logcount #b0110) ⇒ 2
(logcount #b1111) ⇒ 4

(logcount #b-0001) ⇒ 0  ;; 2's complement:  ....111111
(logcount #b-0010) ⇒ 1  ;; 2's complement:  ....111110
(logcount #b-0011) ⇒ 1  ;; 2's complement:  ....111101
(logcount #b-0100) ⇒ 2  ;; 2's complement:  ....111100
```

`integer-length` *n* [Function]

[R7RS bitwise] Returns the minimum number of bits required to represent an exact integer *n*. Negative integer is assumed to be in 2's complement form. A sign bit is not considered.

```
(integer-length 255)  ⇒ 8
(integer-length 256)  ⇒ 9
```

```
(integer-length -256)  ⇒ 8
(integer-length -257)  ⇒ 9
```

`twos-exponent` *n* [Function]

If *n* is a power of two, that is, `(expt 2 k)` and `k >= 0`, then returns `k`. Returns `#f` if *n* is not a power of two.

`twos-exponent-factor` *n* [Function]

Returns maximum *k* such that `(expt 2 k)` is a factor of *n*. In other words, returns the number of consecutive zero bits from LSB of *n*. When *n* is zero, we return `-1` for the consistency of the following equivalent expression.

This can be calculated by the following expression; this procedure is for speed to save creating intermediate numbers when *n* is bignum.

```
(- (integer-length (logxor n (- n 1))) 1)
```

This procedure is also equivalent to srfi-60's `log2-binary-factors` and `first-set-bit` (see Section 11.15 [Integers as bits], page 616).

6.3.7 Endianness

In the Scheme world you rarely need to know about how the numbers are represented inside the machine. However, it matters when you have to exchange data to/from the outer world in binary representation.

Gauche's binary I/O procedures, such as in the `binary.io` module (see Section 12.1 [Binary I/O], page 638) and `write-uvector/read-uvector!` (see Section 9.36 [Uniform vectors], page 476), take optional *endian* argument to specify the endianness.

Currently Gauche recognizes the following endiannesses.

`big-endian`
: Big endian. With this endianness, a 32-bit integer `#x12345678` will be written out as an octet sequence `#x12 #x34 #x56 #x78`.

`little-endian`
: Little endian. With this endianness, a 32-bit integer `#x12345678` is written out as an octet sequence `#x78 #x56 #x34 #x12`.

`arm-little-endian`
: This is a variation of `little-endian`, and used in ARM processors in some specific modes. It works just like `little-endian`, except reading/writing double-precision floating point number (`f64`), which is written as two little-endian 32bit words ordered by big-endian (e.g. If machine register's representation is `#x0102030405060708`, it is written as `#x04 #x03 #x02 #x01 #x08 #x07 #x06 #x05`.

When the *endian* argument is omitted, those procedures use the parameter `default-endian`:

`default-endian` [Parameter]

This is a dynamic parameter (see Section 9.22 [Parameters], page 411) to specify the endianness the binary I/O routines use when its *endian* argument is omitted. The initial value of this parameter is the system's native endianness.

The system's native endianness can be queried with the following procedure:

`native-endian` [Function]

Returns a symbol representing the system's endianness.

6.4 Booleans

<boolean> [Builtin Class]

A boolean class. Only #t and #f belong to this class.

not *obj* [Function]

[R7RS base] Returns #t if and only if *obj* is #f, and returns #f otherwise.

boolean? *obj* [Function]

[R7RS base] Returns #t if *obj* is a boolean value.

boolean *obj* [Function]

Returns #f iff *obj* is #f, and returns #t otherwise. Convenient to coerce a value to boolean.

boolean=? *a b c ...* [Function]

[R7RS base] Every argument must be a boolean value. Returns #t iff all values are the same, #f otherwise.

6.5 Undefined values

While working with Gauche, sometimes you encounter a value printed as #<undef>, an *undefined* value.

```
gosh> (if #f #t)
#<undef>
```

It is a value used as a filler where the actual value doesn't matter, or there's no other suitable value, or the binding hasn't been calculated.

Do not confuse undefined values with unbound variables; A variable can be bound to #<undef>, for it is just an ordinary first-class value. On the other hand, an unbound variable means there's no value associated with the variable.

However, #<undef> may be used in certain occasions to indicate that a value is not provided for the variable. For example, the toplevel variable can be bound to #<undef> if it is defined by (define variable) form (see Section 4.10 [Definitions], page 61). An optional procedure parameter without default value is bound to #<undef> if an actual argument is not given (see Section 4.3 [Making procedures], page 42).

Note that it cannot be distinguished from the case a value is actually provided, and the value just happens to be #<undef>. If you get an #<undef>, you can say at most is that the value doesn't matter. You shouldn't let it carry too much meanings.

The #<undef> value is counted as true value in generalized boolean context, since it is not #f. However, branching based on #<undef> is dangerous—a procedure that is defined to return unspecified value may merely returning #<undef> as a provisional value; it will change the return value in future. Since the return value isn't specified, no one should be using it. The code that tests such result value as a generalized boolean may break if the procedure changes the return value.

In fact, we've found that there are quite a few code that accidentally tests #<undef> return value in conditionals. They can be seeds for future bugs, so we added a feature to warn when #<undef> value is used in the test of branches. You can turn it on with setting the environment variable GAUCHE_CHECK_UNDEFINED_TEST. In future, we may turn it on while testing.

One typical case of such accidental use of #<undef> branching is in and-let*; the following code assumes print always return #<undef>, which is counted as a true value, and expects the control to proceeed to the next clause. It'll break if print ever changes so that it may return #f in some cases.

```
(and-let* ([var (foo x y z)]
```

```
              [ (print var) ]    ;; branch on #<undef>
              [baz (bar var)])
       ...)
```

Being said that, there are a couple of procedures to deal with undefined values.

`undefined?` *obj* [Function]

Returns `#t` iff *obj* is an undefined value.

`undefined` [Function]

Returns an undefined value.

6.6 Pairs and lists

Pairs and lists are one of the most fundamental data structure in Scheme. Gauche core provides all standard list procedures, plus some useful procedures that are commonly supported in lots of implementations. If they are not enough, you can find more procedures in the modules described in Section 10.3.1 [R7RS lists], page 512, and Section 12.62 [Combination library], page 794. See also Section 9.5 [Collection framework], page 344, and Section 9.29 [Sequence framework], page 441, for generic collection/sequence operations.

6.6.1 Pair and null class

`<list>` [Builtin Class]

An abstract class represents lists. A parent class of `<null>` and `<pair>`. Inherits `<sequence>`.

Note that a circular list is also an instance of the `<list>` class, while `list?` returns false on the circular lists and dotted lists.

```
(use srfi-1)
(list? (circular-list 1 2)) ⇒ #f
(is-a? (circular-list 1 2) <list>) ⇒ #t
```

`<null>` [Builtin Class]

A class of empty list. `()` is the only instance.

`<pair>` [Builtin Class]

A class of pairs.

6.6.2 List predicates

`pair?` *obj* [Function]

[R7RS base] Returns `#t` if *obj* is a pair, `#f` otherwise.

`null?` *obj* [Function]

[R7RS base] Returns `#t` if *obj* is an empty list, `#f` otherwise.

`null-list?` *obj* [Function]

[R7RS list] Returns `#t` if *obj* is an empty list, `#f` if *obj* is a pair. If *obj* is neither a pair nor an empty list, an error is signaled.

This can be used instead of `null?` to check the end-of-list condition when you want to be more picky about non-proper lists.

`list?` *obj* [Function]

[R7RS base] Returns `#t` if *obj* is a proper list, `#f` otherwise. This function returns `#f` if *obj* is a dotted or circular list.

See also `proper-list?`, `circular-list?` and `dotted-list?` below.

`proper-list?` *x* [Function]

[R7RS list] Returns `#t` if x is a proper list.

`circular-list?` *x* [Function]

[R7RS list] Returns `#t` if x is a circular list.

`dotted-list?` *x* [Function]

[R7RS list] Returns `#t` if x is a finite, non-nil-terminated list. This includes non-pair, non-() values (e.g. symbols, numbers), which are considered to be dotted lists of length 0.

6.6.3 List constructors

`cons` *obj1 obj2* [Function]

[R7RS base] Constructs a pair of *obj1* and *obj2* and returns it.

```
(cons 'a 'b) ⇒ (a . b)
```

`make-list` *len :optional fill* [Function]

[R7RS base] Makes a proper list of length *len*. If optional argument *fill* is provided, each element is initialized by it. Otherwise each element is undefined.

```
(make-list 5 #t) ⇒ (#t #t #t #t #t)
```

`list` *obj* . . . [Function]

[R7RS base] Makes a list, whose elements are *obj* . . . .

```
(list 1 2 3) ⇒ (1 2 3)
(list) ⇒ ()
```

`list*` *obj1 obj2* . . . [Function]

`cons*` *obj1 obj2* . . . [Function]

[R7RS list] Like `list`, but the last argument becomes cdr of the last pair. Two procedures are exactly the same. Gauche originally had `list*`, and SRFI-1 (R7RS `(scheme list)`)defines `cons*`.

```
(list* 1 2 3) ⇒ (1 2 . 3)
(list* 1) ⇒ 1
```

`list-copy` *list* [Function]

[R7RS base] Shallow copies *list*. If *list* is circular, this function diverges.

`iota` *count :optional* (*start 0*) (*step 1*) [Function]

[R7RS list] Returns a list of *count* numbers, starting from *start*, increasing by *step*. *Count* must be a nonnegative integer. If both *start* and *step* are exact, the result is a list of exact numbers; otherwise, it is a list of inexact numbers.

```
(iota 5)         ⇒ (0 1 2 3 4)
(iota 5 1 3/7)   ⇒ (1 10/7 13/7 16/7 19/7)
(iota 5 0 -0.1)  ⇒ (0 -0.1 -0.2 -0.3 -0.4)
```

This creates a list eagerly. If the list is short it is fast enough, but if you want to count tens of thousands of numbers, you may want to do so lazily. See `liota` (see Section 6.19.2 [Lazy sequences], page 200).

`cond-list` *clause* . . . [Macro]

Construct a list by conditionally adding entries. Each *clause* has a test and expressions. When its test yields true, the result of associated expression is used to construct the resulting list. When the test yields false, nothing is inserted.

Clause must be either one of the following form:

`(`*test expr* `...)`
: *Test* is evaluated, and when it is true, *expr* ... are evaluated, and the return value becomes a part of the result. If no *expr* is given, the result of *test* is used if it is not false.

`(`*test* `=>` *proc*`)`
: *Test* is evaluated, and when it is true, *proc* is called with the value, and the return value is used to construct the result.

`(`*test* `@` *expr* `...)`
: Like `(test expr ...)`, except that the result of the last *expr* must be a list, and it is spliced into the resulting list, like unquote-splicing.

`(`*test* `=> @` *proc*`)`
: Like `(test => proc)`, except that the result of *proc* must be a list, and and it is spliced into the resulting list, like unquote-splicing.

```
(let ((alist '((x 3) (y -1) (z 6))))
 (cond-list ((assoc 'x alist) 'have-x)
            ((assoc 'w alist) 'have-w)
            ((assoc 'z alist) => cadr)))
  ⇒ (have-x 6)

(let ((x 2) (y #f) (z 5))
  (cond-list (x @ `(:x ,x))
             (y @ `(:y ,y))
             (z @ `(:z ,z))))
  ⇒ (:x 2 :z 5)
```

6.6.4 List accessors and modifiers

`car` *pair* [Function]
`cdr` *pair* [Function]

[R7RS base] Returns car and cdr of *pair*, respectively.

`set-car!` *pair obj* [Function]
`set-cdr!` *pair obj* [Function]

[R7RS base] Modifies car and cdr of *pair*, by *obj*, respectively.

Note: `(setter car)` ≡ `set-car!`, and `(setter cdr)` ≡ `set-cdr!`.

`caar` *pair* [Function]
`cadr` *pair* [Function]

...

`cdddar` *pair* [Function]
`cddddr` *pair* [Function]

[R7RS base][R7RS cxr] `caar` ≡ `(car (car x))`, `cadr` ≡ `(car (cdr x))`, and so on.

In R7RS, more than two-level of accessors are defined in the `(scheme cxr)` library.

The corresponding setters are also defined.

```
(let ((x (list 1 2 3 4 5)))
  (set! (caddr x) -1)
  x)
  ⇒ (1 2 -1 4 5)
```

length *list* [Function]

[R7RS base] Returns the length of a proper list *list*. If *list* is a dotted list, an error is signaled. If *list* is a circular list, this function diverges.

length+ *x* [Function]

[R7RS list] If *x* is a proper list, returns its length. For all other *x*, including a circular list, it returns `#f`.

length=? *x k* [Function]
length<? *x k* [Function]
length<=? *x k* [Function]
length>? *x k* [Function]
length>=? *x k* [Function]

Returns `#t` iff *x* is a (possibly improper) list whose length is equal to, less than, less than or equal to, greater than, or greater than or equal to *k*, respectively. This procedure only follows the list up to the *k* items, so it doesn't realize elements of lazy sequence more than needed (See Section 6.19.2 [Lazy sequences], page 200, for the lazy sequences).

Dotted lists and circular lists are allowed. For the dotted list, the cdr of the last pair isn't counted; that is, a non-pair object has length 0, and `(a . b)` has length 1. A circular list is treated as if it has infinite length.

```
(length<=? '(a b) 2)  ⇒ #t
(length<=? '(a b) 1)  ⇒ #f
(length<=? '()    0)  ⇒ #t

;; dotted list cases
(length<=? 'a       0)  ⇒ #t
(length<=? '(a . b) 0)  ⇒ #f
(length<=? '(a . b) 1)  ⇒ #t
```

NB: The name of these procedures might be misleading, for other procedures with the name `something<=?` etc. usually takes objects of the same type. We don't have any better idea now, unfortunately.

take *x i* [Function]
drop *x i* [Function]

[R7RS list] `take` returns the first i elements of list x. `drop` returns all but the first i elements of list x.

```
(take '(a b c d e) 2) => (a b)
(drop '(a b c d e) 2) => (c d e)
```

x may be any value:

```
(take '(1 2 3 . d) 2) => (1 2)
(drop '(1 2 3 . d) 2) => (3 . d)
(drop '(1 2 3 . d) 3) => d
```

`drop` is exactly equivalent to performing *i* cdr operations on *x*. The returned value shares a common tail with *x*. On the other hand, take always allocates a new list for result if the argument is a list of non-zero length.

An error is signaled if *i* is past the end of list *x*. See `take*` and `drop*` below for more tolerant version.

For generic subsequence extraction from any sequence, see `subseq` in Section 9.29.2 [Slicing sequence], page 442.

`take*` *list k :optional (fill? #f) (padding #f)* [Function]
`drop*` *list k* [Function]

More tolerant version of `take` and `drop`. They won't raise an error even if *k* is larger than the size of the given list.

If the list is shorter than *k* elements, `take*` returns a copy of *list* by default. If *fill?* is true, *padding* is added to the result to make its length *k*.

On the other hand, `drop*` just returns an empty list when the input list is shorter than *k* elements.

```
(take* '(a b c d) 3)       ⇒ (a b c)
(take* '(a b c d) 6)       ⇒ (a b c d)
(take* '(a b c d) 6 #t)    ⇒ (a b c d #f #f)
(take* '(a b c d) 6 #t 'z) ⇒ (a b c d z z)
(drop* '(a b c d) 3)       ⇒ (d)
(drop* '(a b c d) 5)       ⇒ ()
```

Note: For generic subsequence extraction from any sequence, see `subseq` in Section 9.29.2 [Slicing sequence], page 442.

`take-right` *lis k* [Function]
`drop-right` *lis k* [Function]

[R7RS list] `take-right` returns the last *k* elements of *lis*. `drop-right` returns all but the last *k* elements of *lis*.

```
(take-right '(a b c d e) 2) => (d e)
(drop-right '(a b c d e) 2) => (a b c)
```

lis may be any finite list.

```
(take-right '(1 2 3 . d) 2) => (2 3 . d)
(drop-right '(1 2 3 . d) 2) => (1)
(take-right '(1 2 3 . d) 0) => d
(drop-right '(1 2 3 . d) 0) => (1 2 3)
```

`take-right`'s return value always shares a common tail with *lis*. `drop-right` always allocates a new list if the argument is a list of non-zero length.

An error is signaled if *k* is larger than the length of *lis*. See `take-right*` and `drop-right*` below, for more tolerant version.

`take-right*` *list k :optional (fill? #f) (padding #f)* [Function]
`drop-right*` *list k* [Function]

Like `take*` and `drop*`, but counts from right of *list*. If *list* is shorter than *k* elements, they won't raise an error. Instead, `drop-right*` just returns an empty list, and `take-right*` returns *list* itself by default. If *fill?* is true for `take-right*`, *padding* is added on the left of the result to make its length *k*. The result still shares the *list*.

`take!` *lis k* [Function]
`drop-right!` *lis k* [Function]

[R7RS list] Linear update variants of `take` and `drop-right`. Those procedures may destructively modifies *lis*.

If *lis* is circular, `take!` may return a list shorter than expected.

`list-tail` *list k :optional fallback* [Function]

[R7RS base] Returns *k*-th cdr of *list*. *list* can be a proper, dotted or circular list. (If *list* is a dotted list, its last `cdr` is simply ignored).

If *k* is negative or larger than the length of *list*, the behavior depends on whether the optional *fallback* argument is given or not. If *fallback* is given, it is returned. Otherwise, an error is signaled.

`list-ref` *list k :optional fallback* [Function]

[R7RS+] Returns *k*-th element of *list*. *list* can be a proper, dotted or circular list.

By default, `list-ref` signals an error if *k* is negative, or greater than or equal to the length of *list*. However, if an optional argument *fallback* is given, it is returned for such case. This is an extension of Gauche.

`list-set!` *list k v* [Function]

[R7RS base] Modifies the *k*-th element of a *list* by *v*. It is an error unless *k* is an exact integer between 0 and one minus the length of *k*. If *list* is immutable, no error is signalled but the behavior is undefined.

`last-pair` *list* [Function]

[R7RS list] Returns the last pair of *list*. *list* can be a proper or dotted list.

```
(last-pair '(1 2 3))   ⇒ (3)
(last-pair '(1 2 . 3)) ⇒ (2 . 3)
(last-pair 1)          ⇒ error
```

`last` *pair* [Function]

[R7RS list] Returns the last element of the non-empty, finite list *pair*. It is equivalent to `(car (last-pair pair))`.

```
(last '(1 2 3))   ⇒ 3
(last '(1 2 . 3)) ⇒ 2
```

`split-at` *x i* [Function]
`split-at!` *x i* [Function]

[R7RS list] `split-at` splits the list *x* at index *i*, returning a list of the first *i* elements, and the remaining tail.

```
(split-at '(a b c d e) 2) ⇒ (a b) (c d e)
```

`split-at!` is the linear-update variant. It may destructively modifies *x* to produce the result.

`split-at*` *list k :optional* (*fill? #f*) (*padding #f*) [Function]

More tolerant version of `split-at`. Returns the results of `take*` and `drop*`.

```
(split-at* '(a b c d) 6 #t 'z)
  ⇒ (a b c d z z) and ()
```

`slices` *list k :optional fill? padding* [Function]

Splits *list* into the sublists (slices) where the length of each slice is *k*. If the length of *list* is not a multiple of *k*, the last slice is dealt in the same way as `take*`; that is, it is shorter than *k* by default, or added *padding* if *fill?* is true.

```
(slices '(a b c d e f g) 3)
  ⇒ ((a b c) (d e f) (g))
(slices '(a b c d e f g) 3 #t 'z)
  ⇒ ((a b c) (d e f) (g z z))
```

`intersperse` *item list* [Function]

Inserts *item* between elements in the *list*. (The order of arguments is taken from Haskell's intersperse).

```
(intersperse '+ '(1 2 3))  ⇒ (1 + 2 + 3)
(intersperse '+ '(1))      ⇒ (1)
(intersperse '+ '())       ⇒ ()
```

6.6.5 Walking over lists

`map` *proc list1 list2 . . .* [Function]

[R7RS+] Applies *proc* for each element(s) of given list(s), and returns a list of the results. R7RS doesn't specify the application order of `map`, but Gauche guarantees *proc* is always applied in order of the list(s). Gauche's `map` also terminates as soon as one of the list is exhausted.

```
(map car '((a b) (c d) (e f))) ⇒ (a c e)

(map cons '(a b c) '(d e f))
  ⇒ ((a . d) (b . e) (c . f))
```

Note that the `gauche.collection` module (see Section 9.5 [Collection framework], page 344) extends `map` to work on any type of collection.

`append-map` *f clist1 clist2 . . .* [Function]
`append-map!` *f clist1 clist2 . . .* [Function]

[R7RS list] Functionally equivalent to the followings, though a bit more efficient:

```
(apply append (map f clist1 clist2 ...))
(apply append! (map f clist1 clist2 ...))
```

At least one of the list arguments must be finite.

`map*` *proc tail-proc list1 list2 . . .* [Function]

Like `map`, except that *tail-proc* is applied to the `cdr` of the last pair in the argument(s) to get the `cdr` of the last pair of the result list. This procedure allows improper list to appear in the arguments. If a single list is given, *tail-proc* always receives a non-pair object.

```
(map* - / '(1 2 3 . 4)) ⇒ (-1 -2 -3 . 1/4)

(define (proper lis)
  (map* values
        (lambda (p) (if (null? p) '() (list p)))
        lis))

(proper '(1 2 3))     ⇒ (1 2 3)
(proper '(1 2 3 . 4)) ⇒ (1 2 3 4)
```

If more than one list are given, the shortest one determines how *tail-proc* is called. When `map*` reaches the last pair of the shortest list, *tail-proc* is called with cdrs of the current pairs.

```
(map* + vector '(1 2 3 4) '(1 2 . 3))
  ⇒ (2 4 . #((3 4) 3))
```

Note: The name `map*` is along the line of `list*`/`cons*` that can produce improper list (See Section 6.6.3 [List constructors], page 126, see Section 10.3.1 [R7RS lists], page 512).

`for-each` *proc list1 list2 . . .* [Function]

[R7RS base] Applies *proc* for each element(s) of given list(s) in order. The results of *proc* are discarded. The return value of `for-each` is undefined. When more than one list is given, `for-each` terminates as soon as one of the list is exhausted.

Note that the `gauche.collection` module (see Section 9.5 [Collection framework], page 344) extends `for-each` to work on any type of collection.

`fold` *kons knil clist1 clist2 . . .* [Function]

[R7RS list] The fundamental list iterator. When it is given a single list *clist1* = (*e1 e2 . . . en*), then this procedure returns

```
(kons en ... (kons e2 (kons e1 knil)) ... )
```

If *n* list arguments are provided, then the *kons* function must take *n*+1 parameters: one element from each list, and the "seed" or fold state, which is initially *knil*. The fold operation terminates when the shortest list runs out of values. At least one of the list arguments must be finite.

Examples:

```
(fold + 0 '(3 1 4 1 5 9)) ⇒ 23 ;sum up the elements
(fold cons '() '(a b c d e)) ⇒ (e d c b a) ;reverse
(fold cons* '() '(a b c) '(1 2 3 4 5))
    ⇒ (c 3 b 2 a 1) ;n-ary case
```

fold-right *kons knil clist1 clist2 . . .* [Function]

[R7RS list] The fundamental list recursion operator. When it is given a single list *clist1* = (*e1 e2 . . . en*), then this procedure returns

```
(kons e1 (kons e2 ... (kons en knil)))
```

If *n* list arguments are provided, then the *kons* function must take *n*+1 parameters: one element from each list, and the "seed" or fold state, which is initially *knil*. The fold operation terminates when the shortest list runs out of values. At least one of the list arguments must be finite.

Examples:

```
(fold-right cons '() '(a b c d e))
   ⇒ (a b c d e) ;copy list
(fold-right cons* '() '(a b c) '(1 2 3 4 5))
   ⇒ (a 1 b 2 c 3) ;n-ary case
```

fold-left *snok knil clist1 clist2 . . .* [Function]

[R6RS] This is another variation of left-associative folding. When it is given a single list *clist1* = (*e1 e2 . . . en*), then this procedure returns:

```
(snok (... (snok (snok knil e1) e2) ...) en)
```

Compare this with `fold` above; association is the same, but the order of arguments passed to the procedure *snok* is reversed from the way arguments are passed to *kons* in `fold`. If *snok* is commutative, `fold` and `fold-left` produces the same result.

```
(fold-left + 0 '(1 2 3 4 5) ⇒ 15

(fold-left cons 'z '(a b c d))
  ⇒ ((((z . a) . b) . c) . d)

(fold-left (^[a b] (cons b a)) 'z '(a b c d))
  ⇒ (a b c d z)
```

If more than one lists are given, *snok* is called with the current seed value *knil* and each corresponding element of the input lists `clist1 clist2 ...`.

```
(fold-left list 'z '(a b c) '(A B C))
  ⇒ (((z a A) b B) c C)
```

Note: Most functional languages have left- and right- associative fold operations, which correspond to `fold-left` and `fold-right`, respectively. (e.g. Haskell's `foldl` and `foldr`). In Scheme, `SRFI-1` first introduced `fold` and `fold-right`. R6RS introduced `fold-left`. (However, in R6RS the behavior is undefined if the lengths of `clist1 clist2` . . . aren't the same, while in Gauche `fold-left` terminates as soon as any one of the lists terminates.)

`reduce` *f ridentity list* [Function]
`reduce-right` *f ridentity list* [Function]

[R7RS list] Variant of `fold` and `fold-right`. *f* must be a binary operator, and *ridentity* is the value such that for any value *x* that is valid as *f*'s input,

```
(f x ridentity) ≡ x
```

These functions effectively do the same thing as `fold` or `fold-right`, respectively, but omit the first application of *f* to *ridentity*, using the above nature. So *ridentity* is used only when *list* is empty.

`filter` *pred list* [Function]
`filter!` *pred list* [Function]

[R7RS list] A procedure *pred* is applied on each element of *list*, and a list of elements that *pred* returned true on it is returned.

```
(filter odd? '(3 1 4 5 9 2 6)) ⇒ (3 1 5 9)
```

`filter!` is the linear-update variant. It may destructively modifies *list* to produce the result.

`filter-map` *f clist1 clist2 . . .* [Function]

[R7RS list] Like `map`, but only true values are saved. At least one of the list arguments must be finite.

```
(filter-map (lambda (x) (and (number? x) (* x x)))
            '(a 1 b 3 c 7))
  ⇒ (1 9 49)
```

`remove` *pred list* [Function]
`remove!` *pred list* [Function]

[R7RS list] A procedure *pred* is applied on each element of *list*, and a list of elements that *pred* returned false on it is returned.

```
(remove odd? '(3 1 4 5 9 2 6)) ⇒ (4 2 6)
```

`remove!` is the linear-update variant. It may destructively modifies *list* to produce the result.

`find` *pred clist* [Function]

[R7RS list] Applies *pred* for each element of *clist*, from left to right, and returns the first element that *pred* returns true on. If no element satisfies *pred*, `#f` is returned.

`find-tail` *pred clist* [Function]

[R7RS list] Applies *pred* for each element of *clist*, from left to right, and when *pred* returns a true value, returns the pair whose car is the element. If no element satisfies *pred*, `#f` is returned.

`any` *pred clist1 clist2 . . .* [Function]

[R7RS list] Applies *pred* across each element of *clist*s, and returns as soon as *pred* returns a non-false value. The return value of `any` is the non-false value *pred* returned. If *clist*s are exhausted before *pred* returns a non-false value, `#f` is returned.

`every` *pred clist1 clist2 . . .* [Function]

[R7RS list] Applies *pred* across each element of *clist*s, and returns `#f` as soon as *pred* returns `#f`. If all application of *pred* return a non-false value, `every` returns the last result of the applications.

`count` *pred clist1 clist2 . . .* [Function]

[R7RS list] A procedure *pred* is applied to the *n*-th element of given lists, from *n* is zero to the length of the the shortest finite list in the given lists, and the count of times *pred* returned true is returned.

```
(count even? '(3 1 4 1 5 9 2 5 6)) ⇒ 3
```

```
(count < '(1 2 4 8) '(2 4 6 8 10 12 14 16)) ⇒ 3
```

At least one of the argument lists must be finite:

```
(count < '(3 1 4 1) (circular-list 1 10)) ⇒ 2
```

`delete` *x list :optional elt=* [Function]
`delete!` *x list :optional elt=* [Function]

[R7RS list] Equivalent to

```
(remove (lambda (y) (elt= x y)) list)
(remove! (lambda (y) (elt= x y)) list)
```

The comparison procedure, *elt=*, defaults to `equal?`.

`delete-duplicates` *list :optional elt=* [Function]
`delete-duplicates!` *list :optional elt=* [Function]

[R7RS list] Removes duplicate elements from *list*. If there are multiple equal elements in *list*, the result list only contains the first or leftmost of these elements in the result. The order of these surviving elements is the same as in the original list. The comparison procedure, *elt=*, defaults to `equal?`.

6.6.6 Other list procedures

`append` *list . . .* [Function]

[R7RS base] Returns a list consisting of the elements of the first *list* followed by the elements of the other lists. The resulting list is always newly allocated, except that it shares structure with the last list argument. The last argument may actually be any object; an improper list results if the last argument is not a proper list.

`append!` *list . . .* [Function]

[R7RS list] Returns a list consisting of the elements of the first *list* followed by the elements of the other lists. The cells in the lists except the last one may be reused to construct the result. The last argument may be any object.

`concatenate` *list-of-lists* [Function]
`concatenate!` *list-of-lists!* [Function]

[R7RS list] Equivalent to `(apply append list-of-lists)` and `(apply append! list-of-lists)`, respectively, but this can be a bit efficient by skipping overhead of `apply`.

`reverse` *list :optional* (*tail* '()) [Function]
`reverse!` *list :optional* (*tail* '()) [Function]

[R7RS+] Returns a list consisting of the elements of *list* in the reverse order. While `reverse` always returns a newly allocated list, `reverse!` may reuse the cells of *list*. Even *list* is destructively modified by `reverse!`, you should use its return value, for the first cell of *list* may not be the first cell of the returned list.

If an optional argument *tail* is given, it becomes the tail of the returned list (*tail* isn't copied). It is useful in the idiom to prepend the processed results on top of already existing results.

```
(reverse '(1 2 3 4 5)) ⇒ (5 4 3 2 1)
(reverse '(1 2 3) '(a b)) ⇒ (3 2 1 a b)
```

The *tail* argument is Gauche's extension, and it isn't in the traditional Scheme's `reverse`. The rationale is the following correspondence:

```
(reverse xs)      ≡ (fold cons xs '())
(reverse xs tail) ≡ (fold cons xs tail)
```

`append-reverse` *rev-head tail* [Function]
`append-reverse!` *rev-head tail* [Function]

[R7RS list] Equivalent to the two-argument `reverse` and `reverse!`. Provided for srfi-1 (R7RS `(scheme list)`) compatibility.

`memq` *obj list* [Function]
`memv` *obj list* [Function]
`member` *obj list :optional obj=* [Function]

[R7RS base] Searches *obj* in the *list*. If `n`-th element of *list* equals to *obj* (in the sense of `eq?` for `memq`, `eqv?` for `memv`, and `equal?` for `member`), `(list-tail list n)` is returned. Otherwise, `#f` is returned.

If the optional *obj=* argument of `member` is given, it is used as a equivalence predicate instead of `equal?`.

```
(memq 'a '(a b c))          ⇒ (a b c)
(memq 'b '(a b c))          ⇒  (b c)
(memq 'a '(b c d))          ⇒ #f
(memq (list 'a) '(b (a) c)) ⇒ #f
(memv 101 '(100 101 102))   ⇒ (101 102)
```

6.6.7 Association lists

`acons` *obj1 obj2 obj3* [Function]

Returns `(cons (cons obj1 obj2) obj3)`. Useful to put an entry at the head of an associative list.

(This procedure is defined in SRFI-1 (R7RS `(scheme list)`) as `alist-cons`; see Section 10.3.1 [R7RS lists], page 512).

```
(acons 'a 'b '((c . d))) ⇒ ((a . b) (c . d))
```

`alist-copy` *alist* [Function]

[R7RS list] Returns a fresh copy of *alist*. The spine of *alist* and each cell that points a key and a value is copied.

```
(define a (list (cons 'a 'b) (cons 'c 'd)))
a ⇒ ((a . b) (c . d))

(define b (alist-copy a))
b ⇒ ((a . b) (c . d))

(set-cdr! (car a) 'z)
a ⇒ ((a . z) (c . d))
b ⇒ ((a . b) (c . d))
```

`assq` *obj list* [Function]
`assv` *obj list* [Function]
`assoc` *obj list :optional key=* [Function]

[R7RS base] Each element in *list* should be a pair (Gauche ignores non-pair element in *list*, but other R7RS implementation may raise an error, so be aware of it when you're writing a portable code). These procedures search a pair whose car matches *obj* (in the sense of `eq?` for `assq`, `eqv?` for `assv`, and `equal?` for `assoc`) from left to right, and return the leftmost matched pair if any. If no pair matches, these return `#f`.

If the optional argument of `assoc` is given, it is called instead of `equal?` to check the equivalence of *obj* and each key.

`alist-delete` *key alist :optional key=* [Function]
`alist-delete!` *key alist :optional key=* [Function]

[R7RS list] Deletes all cells in *alist* whose key is the same as *key*. Comparison is done by a procedure *key=*. The default is `eqv?`.

The linear-update version `alist-delete!` may or may not modify *alist*.

`rassoc` *key alist :optional eq-fn* [Function]
`rassq` *key alist* [Function]
`rassv` *key alist* [Function]

Reverse associations—given *key* is matched to the *cdr* of each element in *alist*, instead of the *car*. Handy to realize bidirectional associative list. `Rassoc` takes an optional comparison function, whose default is `equal?`. `Rassq` and `rassv` uses `eq?` and `eqv?`.

`assoc-ref` *alist key :optional default eq-fn* [Function]
`assq-ref` *alist key :optional default* [Function]
`assv-ref` *alist key :optional default* [Function]

These procedures provide the access to the assoc list symmetric with other `*-ref` procedures. (Note that the argument order is different from `assoc`, `assq` and `assv` – `*-ref` procedures take a container first, and an item second.)

This captures the common pattern of alist access:

```
(assoc-ref alist key default eq-fn)
  ≡
  (cond [(assoc key alist eq-fn) => cdr]
        [else default])))
```

If *default* is omitted, `#f` is used.

`Assoc-ref` takes an optional comparison function *eq-fn*, whose default is `equal?`. `Assq-ref` and `assv-ref` uses `eq?` and `eqv?`, respectively.

`rassoc-ref` *alist key :optional default eq-fn* [Function]
`rassq-ref` *alist key :optional default* [Function]
`rassv-ref` *alist key :optional default* [Function]

Reverse association version of `assoc-ref`.

```
(rassoc-ref alist key default eq-fn)
  ≡
  (cond ((rassoc key alist eq-fn) => car)
        (else default))))
```

The meanings of optional arguments are the same as `assoc-ref`.

`assoc-set!` *alist key val :optional eq-fn* [Function]
`assq-set!` *alist key val* [Function]
`assv-set!` *alist key val* [Function]

Returns an alist who has `(key . val)` pair added to the `alist`. If `alist` already has an element with *key*, the element's *cdr* is destructively modified for *val*. If *alist* doesn't have an element with *key*, a new pair is created and appended in front of *alist*; so you should use the return value to guarantee *key*-*val* pair is added.

`Assoc-set!` takes optional comparison function *eq-fn*, whose default is `equal?`. `Assq-set!` and `assv-set!` uses `eq?` and `eqv?`, respectively.

`assoc-adjoin` *alist key val :optional eq-fn* [Function]

If *alist* contains an entry with *key*, returns a new associative list where the value of the *key* is replaced for *val*. The order of entries in *alist* is preserved. If *alist* doesn't contain the entry, it returns `(acons key val alist)`.

The original *alist* is left unmodified. The returned associative list may share a part of its tail with the original *alist*, however.

The optional *eq-fn* argument is a procedure with two arguments to be used to compare the keys; the default is `equal?`.

Note the order of arguments; we have *alist* first, just as `assoc-ref` and `assoc-set!`, and other `-adjoin` procedures. It is not the same as `alist-delete` and `assoc`, which takes the key first.

`assoc-update-in` *alist keys proc :optional default eq-fn* [Function]

This procedure allows to update a nested associative list. The *alist* argument is a (possibly nested) associative list, *keys* are a list of keys, and *proc* is a procedure that takes one argument. First, the keys are looked up recursively in *alist*; then its value is passed to *proc*. The return value is a new (nested) associative list where the value pointed by *keys* is replaced with the return value of *proc*.

```
(assoc-update-in '((a (b . 1) (c . 2))) '(a c) (cut + <> 1))
  ⇒ ((a (b . 1) (c . 3))
```

The order of entries are preserved. The original *alist* is left unmodified, but the returned value may share a part of the structure with *alist*.

If *alist* doesn't have the entry specified by *keys*, a new entry is added. A new entry is added at the beginning of the sequence where specified key didn't exist.

```
(assoc-update-in '((a (b . 1) (c . 2))) '(a d e) (^_ 99))
  ⇒ ((a (d (e . 99)) (b . 1) (c . 3)))
```

The *default* argument is passed to *proc* when there's no entry with specified keys. If omitted, `#f` is assumed.

The optional *eq-fn* argument is a procedure with two arguments to be used to compare the keys; the default is `equal?`.

Note the order of arguments; we have *alist* first, just as `assoc-ref` and `assoc-set!`, and other `-adjoin` procedures. It is not the same as `alist-delete` and `assoc`, which takes the key first.

Note: For destructively updating general nested aggregate structures, setter of `~` is handy (see Section 6.18.2 [Universal accessor], page 189). You can modify an entry in a hashtable in a vector in a list, for example. Associative list is a bit special, since you can't distinguish it from lists (thus `~` can't be used), and it is mostly used in functional way. So we added a special update procedure.

6.7 Symbols

`<symbol>` [Builtin Class]

A class for symbols.

`|name|` [Reader Syntax]

[R7RS] Denotes a symbol that has weird name, including the characters that are not usually allowed in symbols. It can also include hex-escaped characters.

```
;; A symbol with spaces in its name
'|this is a symbol| ⇒ |this is a symbol|

;; Unicode codepoint can be used following backslash-x escape,
;; and terminated by semicolon.
'|\x3bb;| ⇒ λ
```

If the interpreter is running in case-insensitive mode, this syntax can be used to include uppercase characters in a symbol (see Section 2.4 [Case-sensitivity], page 12).

`#:name` [Reader Syntax]

Denotes *uninterned* symbol. Uninterned symbols can be created by `gensym` or `string->uninterned-symbol`.

Uninterned symbols are mainly for legacy macros to avoid variable conflicts. They are not registered in the internal dictionary, so such symbols with the same name can't be `eq?`.

```
(eq? '#:foo '#:foo) ⇒ #f
(eq? '#:foo 'foo) ⇒ #f
```

When an S-expression including uninterned symbols are printed, the srfi-38 syntax is used to indicate which uninterned symbol is the same (`eq?`) to which.

```
(let1 s '#:foo (list s s))
  ⇒ prints (#0=#:foo #0#)

(let ((s '#:foo) (t '#:foo)) (list s t s t))
  ⇒ prints (#0=#:foo #1=#:foo #0# #1#)
```

`symbol?` *obj* [Function]

[R7RS base] Returns true if and only if *obj* is a symbol.

```
(symbol? 'abc)     ⇒ #t
(symbol? 0)        ⇒ #f
(symbol? 'i)       ⇒ #t
(symbol? '-i)      ⇒ #f
(symbol? '|-i|)    ⇒ #t
```

`symbol-interned?` *symbol* [Function]

Returns `#t` if *symbol* is an interned symbol, `#f` if it is an uninterned symbol. An error is signaled if *symbol* is not a symbol.

`symbol=?` *a b c . . .* [Function]

[R7RS base] Every argument must be a symbol. Returns `#t` iff every pair of arguments are `eq?` to each other.

`symbol->string` *symbol* [Function]

[R7RS base] Returns the name of *symbol* in a string. Returned string is immutable.

```
(symbol->string 'foo) ⇒ foo
```

`string->symbol` *string* [Function]

[R7RS base] Returns a symbol whose name is a string *string*. *String* may contain weird characters.

```
(string->symbol "a") ⇒ a
(string->symbol "A") ⇒ A
(string->symbol "weird symbol name") ⇒ |weird symbol name|
```

`string->uninterned-symbol` *string* [Function]

Like `string->symbol`, but the created symbol is uninterned.

```
(string->uninterned-symbol "a") ⇒ #:a
```

`gensym` *:optional prefix* [Function]

Returns a fresh, uninterned symbol. The returned symbol can never be `eq?` to other symbol within the process. If *prefix* is given, which must be a string, it is used as a prefix of the name of the generated symbol. It is mainly for the convenience of debugging.

`symbol-sans-prefix` *symbol prefix* [Function]

Both *symbol* and *prefix* must be symbols. If the name of *prefix* matches the beginning part of the name of *symbol*, this procedure returns a symbol whose name is the name of *symbol* without the matched prefix. Otherwise, it returns `#f`.

```
(symbol-sans-prefix 'foo:bar 'foo:) ⇒ bar
(symbol-sans-prefix 'foo:bar 'baz:) ⇒ #f
```

`symbol-append` *interned? objs . . .* [Function]
`symbol-append` *objs . . .* [Function]

Returns a symbol with the name which is a concatenation of string representation of *objs*.

If the first argument is a boolean, it is recognized as the first form; the first argument specifies whether the resulting symbol is interned or not.

Each other argument is converted to a string as follows: If it is a keyword, its name (with the preceding `:`) is used. For all other objects, `x->string` is used. (The special treatment of keyword is to keep the consistency before and after keyword-symbol integration. See Section 6.8.1 [Keyword and symbol integration], page 141, for the details.)

This is upper-compatible to Bigloo's same name procedure, which only allows symbols as the arguments and the result is always interned.

```
(symbol-append 'ab 'cd) ⇒ abcd
(symbol-append 'ab ':c 30) ⇒ ab:c30
(symbol-append #f 'g 100) ⇒ #:g100
```

6.8 Keywords

`<keyword>` [Builtin Class]

Keywords are a subtype of symbols that are automatically bound to itself. It is extensively used in named arguments (keyword arguments), and keyword-value list.

See Section 4.3 [Making procedures], page 42, for how Gauche supports keyword arguments, and `let-keywords` macro (Section 6.18.4 [Optional argument parsing], page 192) for parsing keyword-value list manually.

Keywords used to be a disjoint type from symbols. Since it isn't conformant to R7RS, in which symbols can begin with `:`, we've introduced two modes since 0.9.5; keywords can be a disjoint type of its own, or it can be a subtype of symbols.

The behavior can be switched by environment variables. If the environment variable `GAUCHE_KEYWORD_DISJOINT` is defined when `gosh` starts up, keywords and symbols are disjoint. Otherwise, if the environment variable `GAUCHE_KEYWORD_IS_SYMBOL` is defined, keywords are a subtype of symbols.

The default behavior when neither environment variables are defined has been switched since 0.9.8. `GAUCHE_KEYWORD_DISJOINT` was assumed in 0.9.7 and before, while `GAUCHE_KEYWORD_IS_SYMBOL` is assumed in 0.9.8 and after.

Most typical code run in either mode, but there can be some code that behaves differently. See Section 6.8.1 [Keyword and symbol integration], page 141, for effect of the change.

In future we'll stop supporting `GAUCHE_KEYWORD_DISJOINT`, so we recommend you to ensure applications to run on the current default mode.

`:`*name* [Reader Syntax]

Read to a keyword whose name is *:name*.

`keyword?` *obj* [Function]

Returns `#t` if *obj* is a keyword.

`make-keyword` *name* [Function]

Returns a keyword whose name is *name* prepended by `:`. The *name* argument can be a string or a symbol.

```
(make-keyword "foo")  ⇒ :foo

(make-keyword 'foo)   ⇒ :foo
```

`keyword->string` *keyword* [Function]

Returns the name (without the initial `:`) of the keyword *keyword*, in a string.

```
(keyword->string :foo) ⇒ "foo"
```

`get-keyword` *key kv-list :optional fallback* [Function]

A useful procedure to extract a value from key-value list. A key-value list *kv-list* must contains even number of elements; the first, third, fifth ... elements are regarded as keys, and the second, fourth, sixth ... elements are the values of the preceding keys.

This procedure looks for *key* from the keys, and if it finds one, it returns the corresponding value. If there are more than one matching keys, the leftmost one is taken. If there is no matching key, it returns *fallback* if provided, or signals an error otherwise.

It is an error if *kv-list* is not a proper, even-number element list.

Actually, 'keywords' in the keyword-value list and the *key* argument need not be a keyword— it can be any Scheme object. Key comparison is done by `eq?`.

This procedure is taken from STk.

```
(get-keyword :y '(:x 1 :y 2 :z 3))
  ⇒ 2
(get-keyword 'z '(x 1 y 2 z 3))
  ⇒ 3

(get-keyword :t '(:x 1 :y 2 :z 3))
  ⇒ #<error>
(get-keyword :t '(:x 1 :y 2 :z 3) #f)
  ⇒ #f
```

`get-keyword*` *key kv-list :optional fallback* [Macro]

Like `get-keyword`, but *fallback* is evaluated only if *kv-list* does not have *key*.

`delete-keyword` *key kv-list* [Function]
`delete-keyword!` *key kv-list* [Function]

Removes all the keys and values from *kv-list* for keys that are `eq?` to *key*.

`delete-keyword` doesn't change *kv-list*, but the returned list may share the common tail of it.

`delete-keyword!` doesn't allocate, and *may* destructively changes *kv-list*. You still have to use the returned value, for the original list may not be changed if its first key matches *key*.

If there's no key that matches *key*, *kv-list* is returned.

```
(delete-keyword :y '(:x 1 :y 2 :z 3 :y 4))
 ⇒ (:x 1 :z 3)
```

`delete-keywords` *keys kv-list* [Function]
`delete-keywords!` *keys kv-list* [Function]

Similar to `delete-keyword` and `delete-keyword!`, but you can specify a list of objects in *keys*; when a key in *kv-list* matches any of *keys*, the key and the following value is removed from *kv-list*.

```
(delete-keywords '(:x :y) '(:x 1 :y 2 :z 3 :y 4))
```

```
⇒ (:z 3)
```

6.8.1 Keyword and symbol integration

In older versions of Gauche, keywords are of disjoint type from symbols, and they are self-evaluating objects. To maintain the compatibility, the current Gauche makes symbols that begins with `:` automatically bound to itself.

On the surface it won't make much difference; you can write a keyword `:key`, which evaluates to itself; so you can pass and receive keyword arguments just as they used to be. If you use `:key` as variables, however, e.g. `(define :key 3)`, the value of `:key` in your module changes (it won't affect other modules, which refer to the binding of `:key` in `gauche.keyword` module).

However, there are several subtle points that do make difference, that breaks compatibility of legacy code. We explain here how to change the code that works in both ways.

If you find a problem in new mode and want to get the old behavior until you change the code, you can set the environment variable `GAUCHE_KEYWORD_DISJOINT.`

`(symbol? :key)` used to return `#f`, now returns `#t`

`keyword?` always returns `#t` on keywords, but if you need to switch behavior depending whether an object is a symbol or a keyword, you should test keyword-ness first.

```
;; behaved differently in 0.9.7 and before
(cond
  [(symbol? x) (x-is-symbol)]
  [(keyword? x) (x-is-keyword)])

;; works on all versions
(cond
  [(keyword? x) (x-is-keyword)]
  [(symbol? x) (x-is-symbol)])
```

Literal keywords in pattern matching

In the old versions, when keywords appear in a pattern of `util.match` or `syntax-rules`, they only matched to themselves. In the current version, such keywords in a pattern are treated as pattern variables, since they are symbols.

```
;; In the old versions
(match '(a b) [(:key z) (list :key z)] [_ "nope"])
  ⇒ "nope"

;; In the current version
;; :key is treated just as a pattern variable
(match '(a b) [(:key z) (list :key z)] [_ "nope"])
  ⇒ (a b)
```

The same thing happens to the patterns in `syntax-rules`.

To make the code work in both versions, explicitly mark the keywords as literals.

- For `match`, quote the keywords you want to be treated as literals.

  ```
  (match '(a b) [(':key z) (list :key z)] [_ "nope"])
    ⇒ "nope"
  ```

- For `syntax-rules`, list the keywords as literals.

  ```
  (syntax-rules (:key)
    [(_ :key z) (list :key z)])  ;etc.
  ```

As of Gauche 0.9.5, `match` warns if you have unquoted keywords in match patterns.

Displaying keywords

`(display :key)` used to print `key` (no colon), while it now prints `:key`.

You can use `(display (keyword->string :key))` which prints `key` in both versions.

For R7RS code, quote them or import Gauche modules

Keywords (symbols beginning with `:`) are automatically bound to itself in the `gauche.keyword` module.

Gauche code inherits the `gauche` module by default, which inherits `keyword`, so you can see the binding of the keyword by default.

In R7RS code, however, you don't inherit `gauche`, so symbols beginning with `:` are just ordinary symbols by default. Usually you do `(import (gauche base))` to use Gauche built-ins, and that makes binding of `gauche.keyword` available in your code, too (since `gauche.base` inherits `gauche.keyword`). But keep this in mind just in case you want to handle keywords in your R7RS code separate from Gauche procedures—you have to either say `(import (gauche keyword))` to get just the self-bound keywords, or quote them.

```
(import (scheme base))

:foo ⇒ ERROR: unbound variable: :foo

(import (gauche base))

:foo ⇒ :foo
```

In the following example, the R7RS library `foo` imports only `copy-port` from `(gauche base)`; in that case, you have to import `(gauche keyword)` separately in order to use `:size` keyword without quoting. (Or add `:size` explicitly in the imported symbol list of `(gauche base)`.)

```
(define-library (foo)
  (import (scheme base)
          (only (gauche base) copy-port)
          (gauche keyword))
  (export cat)

  (begin
    (define (cat)
      (copy-port (current-input-port)
                 (current-output-port)
                 :size 4096))))
```

6.9 Identifiers

`<identifier>` [Builtin Class]

An identifier is an internal object to keep track of binding of variables by the compiler.

Usually it is hidden from Scheme world, but the hygienic macro expander inserts identifiers into its output, which is necessary for hygiene. But that makes reading macro expansion result difficult.

If you bothered by all the `#<identifier ...>` stuff in the macro output, remember a handy trick to pass the expansion result to `unwrap-syntax`; it converts all identifiers in the passed form to bare symbols. It does lose information—two different identifiers may be converted to a symbol with the same name—so you need some care to interpret the output, but usually the output gives a fairly good idea of what the macro is doing.

Currently, identifiers are disjoint from symbols. That might cause problems if you tweak macro output. The plan is to make identifiers just a special kind of symbols eventually, so do not assume too much about identifiers.

identifier? *obj* [Function]

identifier->symbol *identifier* [Function]

unwrap-syntax *form* [Function]

6.10 Characters

<char> [Builtin Class]

#*charname* [Reader Syntax]

[R7RS] Denotes a literal character.

When the reader reads `#\`, it fetches a subsequent character. If it is one of `()[]{}" \|;#`, this is a character literal of itself. Otherwise, the reader reads subsequent characters until it sees a non word-constituent character. If only one character is read, it is the character. Otherwise, the reader matches the read characters with predefined character names. If it doesn't match any, an error is signaled.

The following character names are recognized. These character names are case insensitive.

`space` Whitespace (ASCII #x20)

`newline, nl, lf`
Newline (ASCII #x0a)

`return, cr`
Carriage return (ASCII #x0d)

`tab, ht` Horizontal tab (ASCII #x09)

`page` Form feed (ASCII #x0c)

`escape, esc`
Escape (ASCII #x1b)

`delete, del`
Delete (ASCII #x7f)

`null` NUL character (ASCII #x00)

`x`*N* A character whose Unicode codepoint is the integer *N*, when *N* is a hexadecimal integer. This is R7RS lexical syntax. (See the compatibility note below).

`u`*N* A character whose Unicode codepoint is the integer *N*, where *N* is 4-digit or 8-digit hexadecimal number.

This is legacy Gauche lexical syntax. Use `\xN` syntax for the new code. (See the compatibility note below).

```
#\newline ⇒ #\newline ; newline character
#\x0a     ⇒ #\newline ; ditto
#\x41     ⇒ #\A       ; ASCII letter 'A'
#\x3042   ⇒ ; Hiragana letter A
#\x2a6b2  ⇒ ; JISX0213 Kanji 2-94-86
```

Compatibility note: Before 0.9.4, `\xNN` syntax uses Gauche's internal character encoding as opposed to Unicode codepoint. Both are the same if Gauche is compiled with internal encoding `utf-8` or `none` (if it's `none`, only characters up to U+00ff is supported and in this range the characters are the same as Unicode characters.) If Gauche is compiled with encoding `euc-jp` or `sjis`, the meaning of `\xNN` beyond ASCII range differs from 0.9.3.3 or before.

If you set the reader mode to `legacy` (see Section 6.22.7.2 [Reader lexical mode], page 228), `#\xNN` is read as before, keeping the compatibility (but it isn't compatible to R7RS). Alternatively, you can use `#\uNNNN`, or a character itself, to make the code work in both new and old versions of Gauche.

`char?` *obj* [Function]

[R7RS base] Returns `#t` if *obj* is a character, `#f` otherwise.

`char=?` *char1 char2 char3 . . .* [Function]
`char<?` *char1 char2 char3 . . .* [Function]
`char<=?` *char1 char2 char3 . . .* [Function]
`char>?` *char1 char2 char3 . . .* [Function]
`char>=?` *char1 char2 char3 . . .* [Function]

[R7RS base] Compares characters. Character comparison is done in internal character encoding.

`char-ci=?` *char1 char2 char3 . . .* [Function]
`char-ci<?` *char1 char2 char3 . . .* [Function]
`char-ci<=?` *char1 char2 char3 . . .* [Function]
`char-ci>?` *char1 char2 char3 . . .* [Function]
`char-ci>=?` *char1 char2 char3 . . .* [Function]

[R7RS char] Compares characters in case-insensitive way. The comparison is done in the internal character code of the foldcase of the each character; see `char-foldcase` below.

In R7RS, these procedures are in the `(scheme char)` library.

`char-alphabetic?` *char* [Function]
`char-numeric?` *char* [Function]
`char-whitespace?` *char* [Function]
`char-upper-case?` *char* [Function]
`char-lower-case?` *char* [Function]

[R7RS char] Returns true if a character *char* is an alphabetic character (Unicode character category `Lu`, `Ll`, `Lt`, `Lm`, `Lo`, `Nl`), a numeric character (Unicode character category `Nd`), a whitespace character, (Unicode character category `Zs`, `Zp`, `Zl`), an upper case character (Unicode character category `Lu`), or a lower case character (Unicode character category `Ll`), respectively.

In R7RS, these procedures are in the `(scheme char)` library.

`char-general-category` *char* [Function]

[R6RS] Returns one of the following symbols, representing the Unicode general category of *char*.

- `Cc` Other, Control
- `Cf` Other, Format
- `Cn` Other, Not Assigned
- `Co` Other, Private Use
- `Cs` Other, Surrogate
- `Ll` Letter, Lowercase
- `Lm` Letter, Modifier
- `Lo` Letter, Other
- `Lt` Letter, Titlecase
- `Lu` Letter, Uppercase
- `Mc` Mark, Spacing Combining

Me	Mark, Enclosing
Mn	Mark, Nonspacing
Nd	Number, Decimal Digit
Nl	Number, Letter
No	Number, Other
Pc	Punctuation, Connector
Pd	Punctuation, Dash
Pe	Punctuation, Close
Pf	Punctuation, Final quote
Pi	Punctuation, Initial quote
Po	Punctuation, Other
Ps	Punctuation, Open
Sc	Symbol, Currency
Sk	Symbol, Modifier
Sm	Symbol, Math
So	Symbol, Other
Zl	Separator, Line
Zp	Separator, Paragraph
Zs	Separator, Space

If Gauche is compiled with euc-jp or shift_jis encoding, there are characters that don't have corresponding Unicode codepoint (each of them are represented by one unicode character plus one unicode modifier character). A provisional category is assigned to those characters. If future versions of Unicode incorporates these characters, the category may be reassigned.

SJIS	EUC	Cat	Unicode
82F5	A4F7	Lo	U+304B U+309A (Semi-voiced Hiragana KA)
82F6	A4F8	Lo	U+304D U+309A (Semi-voiced Hiragana KI)
82F7	A4F9	Lo	U+304F U+309A (Semi-voiced Hiragana KU)
82F8	A4FA	Lo	U+3051 U+309A (Semi-voiced Hiragana KE)
82F9	A4FB	Lo	U+3053 U+309A (Semi-voiced Hiragana KO)
8397	A5F7	Lo	U+30AB U+309A (Semi-voiced Katakana KA)
8398	A5F8	Lo	U+30AD U+309A (Semi-voiced Katakana KI)
8399	A5F9	Lo	U+30AF U+309A (Semi-voiced Katakana KU)
839A	A5FA	Lo	U+30B1 U+309A (Semi-voiced Katakana KE)
839B	A5FB	Lo	U+30B3 U+309A (Semi-voiced Katakana KO)
839C	A5FC	Lo	U+30BB U+309A (Semi-voiced Katakana SE)
839D	A5FD	Lo	U+30C4 U+309A (Semi-voiced Katakana TSU)
839E	A5FE	Lo	U+30C8 U+309A (Semi-voiced Katakana TO)
83F6	A6F8	Lo	U+31F7 U+309A (Semi-voiced small Katakana FU)
8663	ABC4	Ll	U+00E6 U+0300 (Accented latin small ae)
8667	ABC8	Ll	U+0254 U+0300 (Accented latin small open o)
8668	ABC9	Ll	U+0254 U+0301 (Accented latin small open o)
8669	ABCA	Ll	U+028C U+0300 (Accented latin small turned v)
866A	ABCB	Ll	U+028C U+0301 (Accented latin small turned v)
866B	ABCC	Ll	U+0259 U+0300 (Accented latin small schwa)
866C	ABCD	Ll	U+0259 U+0301 (Accented latin small schwa)
866D	ABCE	Ll	U+025A U+0300 (Accented latin small schwa w/hook)
866E	ABCF	Ll	U+025A U+0301 (Accented latin small schwa w/hook)
8685	ABE5	Sk	U+02E9 U+02E5
8686	ABE6	Sk	U+02E5 U+02E9

`char->integer` *char* [Function]
`integer->char` *n* [Function]

[R7RS base] `char->integer` returns an exact integer that represents internal encoding of the character *char*. `integer->char` returns a character whose internal encoding is an exact integer *n*. The following expression is always true for valid character *char*:

```
(eq? char (integer->char (char->integer char)))
```

Note: R7RS defines these procedures to deal with Unicode codepoints. Gauche complies it when compiled with `utf-8` or `none` internal encoding (for the latter, only characters up to U+00ff are supported). If Gauche is compiled with `euc-jp` or `sjis` internal encoding, you need to use `char->ucs`/`ucs->char` below to convert between Unicode codepoints and characters.

The result is undefined if you pass *n* to `integer->char` that doesn't have a corresponding character.

`char->ucs` *char* [Function]
`ucs->char` *n* [Function]

Converts a character *char* to integer UCS codepoint, and integer UCS codepoint *n* to a character, respectively.

If Gauche is compiled with UTF-8 encoding, these procedures are the same as `char->integer` and `integer->char`.

When Gauche's internal encoding differs from UTF-8, these procedures implicitly loads `gauche.charconv` module to convert internal character code to UCS or vice versa (see Section 9.4 [Character code conversion], page 339). If *char* doesn't have corresponding UCS codepoint, `char->ucs` returns `#f`. If UCS codepoint *n* can't be represented in the internal character encoding, `ucs->char` returns `#f`, unless the conversion routine provides a substitution character.

`char-upcase` *char* [Function]
`char-downcase` *char* [Function]
`char-titlecase` *char* [Function]
`char-foldcase` *char* [Function]

[R7RS char] Returns the upper case, lower case, title case and folded case of *char*, respectively.

The mapping is done according to Unicode-defined character-by-character case mapping whenever possible. If the native encoding doesn't support the mapped character defined in Unicode, the operation becomes no-op. If the native encoding is 'none', we treat the characters as if they are Latin-1 (ISO-8859-1) characters. So, upcasing Latin-1 character small y with diaresis (U+00ff) maps to capital y with diaeresis (U+0178) if the internal encoding is utf-8, but it is no-op if the internal encoding is none.

R7RS doesn't have `char-titlecase`; other three procedures are defined in the `(scheme char)` library. R6RS defines all of them.

The character-by-character case mapping doesn't consider a character that may map to more than one characters; a notable example is eszett (latin small letter sharp S, U+00df), which is is mapped to two capital S's in string context, but `char-upcase #\ß` returns #\ß. To get a full mapping, use `string-upcase` etc. in `gauche.unicode` module (see Section 9.35.3 [Full string case conversion], page 475).

`digit->integer` *char :optional (radix 10) (extended-range? #f)* [Function]

If given character *char* is a valid digit character in radix *radix* number, the corresponding integer is returned. Otherwise `#f` is returned.

```
(digit->integer #\4) ⇒ 4
(digit->integer #\e 16) ⇒ 14
```

```
(digit->integer #\9 8) ⇒ #f
```

If the optional *extended-range?* argument is true, this procedure recognizes not only ASCII digits, but also all characters with `Nd` general category—such as FULLWIDTH DIGIT ZERO to NINE (U+ff10 - U+ff19).

R7RS has `digit-value`, which is equivalent to `(digit->integer char 10 #t)`.

Note: CommonLisp has a similar function in rather confusing name, `digit-char-p`.

`integer->digit` *integer :optional* (*radix 10*) (*basechar1 #\0*) (*basechar2 #\a*) [Function]

Reverse operation of `digit->integer`. Returns a character that represents the number *integer* in the radix *radix* system. If *integer* is out of the valid range, `#f` is returned.

```
(integer->digit 13 16) ⇒ #\d
(integer->digit 10) ⇒ #f
```

The optional *basechar1* argument specifies the character that stands for zero; by default, it's `#\0`. You can give alternative character, for example, U+0660 (ARABIC-INDIC DIGIT ZERO) to convert an integer to a arabic-indic digit character.

Another optional *basechar2* argument is used for integers over 10. The default value is `#\a`. You can pass `#\A` to get upper-case hex digits, for example.

Note: CommonLisp's `digit-char`.

`gauche-character-encoding` [Function]

Returns a symbol designates the native character encoding, selected at the compile time. The possible return values are those:

`euc-jp` EUC-JP

`utf-8` UTF-8

`sjis` Shift JIS

`none` No multibyte character support (8-bit fixed-length character).

To switch code at compile time according to the internal encoding, you can use feature identifiers `gauche.ces.*`–see Section 3.5 [Platform-dependent features], page 29.

`supported-character-encodings` [Function]

Returns a list of string names of character encoding schemes that are supported in the native multibyte encoding scheme.

6.11 Character Set

`<char-set>` [Builtin Class]

Character set class. Character set object represents a set of characters. Gauche provides built-in support of character set creation and a predicate that tests whether a character is in the set or not.

The class implements the collection protocol (see Section 9.5 [Collection framework], page 344), so that the standard collection methods provided in the `gauche.collection` module can be used.

An instance of `<char-set>` is applicable to a character, and works as a membership predicate; see `char-set-contains?` below.

Further operations, such as set algebra, is defined in SRFI-14 module (see Section 11.6 [Character-set library], page 600).

Char-set literal

`#[char-set-spec]` [Reader Syntax]

You can write a literal character set in this syntax. *char-set-spec* is a sequence of characters to be included in the set. You can include the following special sequences:

x-y
: Characters between *x* and *y*, inclusive. *x* must be smaller than *y* in the internal encoding.

`^`
: If *char-set-spec* begins with caret, the actual character set is a complement of what the rest of *char-set-spec* indicates.

`\xN;`
: A character whose Unicode codepoint is a hexadecimal number *N*.

`\uXXXX`
`\UXXXXXXXX`
: This is a legacy Gauche syntax, for a unicode character whose Unicode codepoint is represented by 4-digit and 8-digit hexadecimal numbers, respectively.

`\s`
: Whitespace characters (space, newline, tab, form feed, vertical tab, carriage return). Members of `char-set:ascii-whitespace`.

`\S`
: Complement of whitespace characters.

`\d`
: Decimal digit characters. Members of `char-set:ascii-digits`.

`\D`
: Complement of decimal digit characters.

`\w`
: Word constituent characters (`#[A-Za-z0-9_]`). Members of `char-set:ascii-word`.

`\W`
: Complement of word constituent characters.

`\\`
: A backslash character.

`\-`
: A minus character.

`\^`
: A caret character.

`[:alnum:] ...`
: Character set a la POSIX. See the table below for the complete list of recognized character set names. The set name must be in all lower cases. This notation only includes characters in ASCII range.

`[:^alnum:] ...`
: Complement set of `[:alnum:]` etc.

`[:ALNUM:] ...`
: Gauche's extension of character set a la POSIX; the name must be all in upper cases, and includes full Unicode range. See the table below for the recognized names.

`[:^ALNUM:] ...`
: Complement set of `[:ALNUM:]` etc.

Here's the list of POSIX-style character class names:

`:alpha:`	ASCII alphabets. `char-set:ascii-letter`, `#[A-Za-z]`
`:alnum:`	ASCII alphabets and digits. `char-set:ascii-letter+digits`, `#[0-9A-Za-z]`.
`:blank:`	ASCII blanks. `char-set:ascii-blank`, tab and space.
`:cntrl:`	ASCII control characters. `char-set:ascii-control`, U+0000 to U+001f and U+007f.
`:digit:`	ASCII digits. `char-set:ascii-digit`, `#[0-9]`.

`:graph:`	ASCII graphic characters. `char-set:ascii-graphic`.
`:lower:`	ASCII lower-case alphabets. `char-set:ascii-lower-case`, `#[a-z]`.
`:print:`	ASCII printing characters. `char-set:ascii-printing`.
`:punct:`	ASCII punctuation characters. `char-set:ascii-punctuation`.
`:space:`	ASCII whitespaces. `char-set:ascii-whitespace`.
`:upper:`	ASCII upper-case characters. `char-set:ascii-upper-case`, `#[A-Z]`.
`:word:`	ASCII word characters (not POSIX). `char-set:ascii-word`, `#[0-9A-Za-z_]`.
`:xdigit:`	Hexadecimal digits. `char-set:hex-digit`, `#[0-9a-fA-F]`.
`:ascii:`	ASCII characters (not POSIX). `char-set:ascii`.
`:ALPHA:`	Unicode letters (category `L*`). `char-set:letter`.
`:ALNUM:`	Unicode letters and digits. `char-set:letter+digits`.
`:BLANK:`	Unicode blanks (tab and category `Zs`). `char-set:blank`.
`:CNTRL:`	Unicode control characters (category `Cc`). `char-set:iso-control`.
`:DIGIT:`	Unicode digits (category `Nd`). `char-set:digit`.
`:GRAPH:`	Unicode graphic characters (letter, digits, punctuation, symbol, and category `Nl` and `No`). `char-set:graphic`.
`:LOWER:`	Unicode lower-case letters (category `Ll`). `char-set:lower-case`, `#[a-z]`.
`:PRINT:`	Unicode printing characters (graphic and whitespace). `char-set:printing`.
`:PUNCT:`	Unicode punctuation characters (category `P*`). `char-set:punctuation`.
`:SPACE:`	Unicode whitespaces (tab, LF, vertical tab, FF, CR, and category `Z*`). `char-set:whitespace`.
`:TITLE:`	Unicode titlecase letters (category `Lt`). `char-set:title-case`.
`:UPPER:`	Unicode upper-case letters (category `Lu`). `char-set:upper-case`, `#[A-Z]`.
`:WORD:`	Unicode word characters. `char-set:word`.
`:XDIGIT:`	Hexadecimal digits (same as `:xdigit:`).

Here are some examples:

```
#[aeiou]           ; a character set consists of vowels
#[a-zA-Z]          ; alphabet
#[[:alpha:]]       ; alphabet (using POSIX notation)
#[\\\-]            ; backslash and minus
#[]                ; empty charset
#[\x0d;\x0a;\x3000;] ; carriage return, newline, and ideographic space
```

Literal character sets are immutable, as other literal data. An error is signalled when you attempt to modify an immutable character set.

Note for the compatibility: We used to recognize a syntax `\xNN` (two-digit hexadecimal number, without semicolon terminator) as a character; for example, `#[\x0d\x0a]` as a return and a newline. We still support it when we don't see the terminating semicolon, for the compatibility. There are ambiguous cases: `#[\x0a;]` means only a newline in the current syntax, but a newline and a semicolon in legacy syntax.

Setting the reader mode to `legacy` restores the old behavior. Setting the reader mode to `warn-legacy` makes it work like the default behavior, but prints warning when it finds legacy syntax. See Section 6.22.7.2 [Reader lexical mode], page 228, for the details.

To write code that can work both in new and old syntax, use `\u` escape.

Predefined char-sets

We provide a bunch of predefined character sets, including the ones defined in R7RS charset library (see Section 10.3.6 [R7RS character sets], page 533). Those character sets are immutable.

`char-set:letter` [Variable]

[R7RS charset] Letters (Unicode general category `Lu`, `Ll`, `Lt`, `Lm` and `Lo`).

`char-set:lower-case` [Variable]
`char-set:upper-case` [Variable]
`char-set:title-case` [Variable]

[R7RS charset] Lower case, upper case and title case letters (Unicode general category `Ll`, `Lu` and `Lt`, respectively).

`char-set:digit` [Variable]

[R7RS charset] Digit characters (Unicode general category `Nd`). Note that this contains many more characters than ASCII `0` to `9`. If you need `#[0-9]`, use `char-set:ascii-digit`.

`char-set:hex-digit` [Variable]

[R7RS charset] Digit characters used for hexadecimal, i.e. `#[0-9A-Fa-f]`. This does not contain other Unicode digit characters, for it isn't practical to mix non-ascii digit characters with hexadecimal notation.

`char-set:letter+digit` [Variable]
`char-set:graphic` [Variable]
`char-set:printing` [Variable]

[R7RS charset]

`char-set:whitespace` [Variable]
`char-set:blank` [Variable]

[R7RS charset] Whitespace and blank characters; `char-set:whitespace` includes `#\tab`, `#\newline`, `#\u000B` (vertical tab), `#\page`, `#\return`, and all characters in general category `Zs`, `Zl`, `Zp`, while `char-set:blank` includes `#\tab` and all characters in general category `Zs`. Note that `char-set:whitespace` is the same set of characters that Scheme reader treats as whitespace characters.

`char-set:iso-control` [Variable]

[R7RS charset] Control characters (Unicode general category `Cc`).

`char-set:punctuation` [Variable]

[R7RS charset] Punctuation characters (Unicode general category `Pc`, `Pd`, `Ps`, `Pe`, `Pi`, `Pf` and `Po`).

`char-set:symbol` [Variable]

[R7RS charset] Symbol characters (Unicode general category `Sm`, `Sc`, `Sk` and `So`).

`char-set:ascii` [Variable]

[R7RS charset] Contains all ASCII characters (U+0000 to U+007f).

`char-set:empty` [Variable]

[R7RS charset] An empty character set.

`char-set:full` [Variable]

[R7RS charset] A character set that includes all characters.

`char-set:word` [Variable]

A word constituent characters. In the current version, it is equivalent to `char-set:ascii-word` (`#[0-9A-Za-z_]`) but in future versions we may extend this to other Unicode characters. If you intend to mean ASCII-only words, use `char-set:ascii-word`.

`char-set:ascii-letter` [Variable]
`char-set:ascii-lower-case` [Variable]
`char-set:ascii-upper-case` [Variable]
`char-set:ascii-digit` [Variable]

char-set:ascii-letter+digit [Variable]
char-set:ascii-graphic [Variable]
char-set:ascii-printing [Variable]
char-set:ascii-whitespace [Variable]
char-set:ascii-blank [Variable]
char-set:ascii-control [Variable]
char-set:ascii-punctuation [Variable]
char-set:ascii-symbol [Variable]
char-set:ascii-word [Variable]

These are intersection of `char-set:ascii` and the corresponding char set without `ascii-`. (`char-set:ascii-control` corresponds to `char-set:iso-control`).

The `\d`, `\s` and `\w` notation in the char-set literal and regexp literal corresponds to `char-set:ascii-digit`, `char-set:ascii-whitespace`, and `char-set:ascii-word`, respectively (not the Unicode set).

The POSIX character class notation, such as `[:alpha:]` in char-set literal and regexp literal, refers to these ASCII-only charsets.

Note: We don't have `char-set:ascii-title-case` and `char-set:ascii-hex-digit`. There's no titlecase letter in ASCII range. And `char-set:hex-digit` is limited to ASCII by definition.

char-set:Lu [Variable]
char-set:Ll [Variable]
char-set:Lt [Variable]
char-set:Lm [Variable]
char-set:Lo [Variable]
char-set:Mn [Variable]
char-set:Mc [Variable]
char-set:Me [Variable]
char-set:Nd [Variable]
char-set:Nl [Variable]
char-set:No [Variable]
char-set:Pc [Variable]
char-set:Pd [Variable]
char-set:Ps [Variable]
char-set:Pe [Variable]
char-set:Pi [Variable]
char-set:Pf [Variable]
char-set:Po [Variable]
char-set:Sm [Variable]
char-set:Sc [Variable]
char-set:Sk [Variable]
char-set:So [Variable]
char-set:Zs [Variable]
char-set:Zl [Variable]
char-set:Zp [Variable]
char-set:Cc [Variable]
char-set:Cf [Variable]
char-set:Cs [Variable]
char-set:Co [Variable]
char-set:Cn [Variable]

Each character set contains the corresponding Unicode characters with the given general category; e.g. `char-set:Lu` contains all characters of the general category `Lu`.

`char-set:L` [Variable]
`char-set:M` [Variable]
`char-set:N` [Variable]
`char-set:P` [Variable]
`char-set:S` [Variable]
`char-set:Z` [Variable]
`char-set:C` [Variable]

Each character set contains the Unicode characters with the general category starting with the letter; e.g. `char-set:L` is union of `char-set:Lu`, `char-set:Ll`, `char-set:Lt`, `char-set:Lm` and `char-set:Lo`.

Char-set operations

`char-set?` *obj* [Function]

[SRFI-14] Returns true if and only if *obj* is a character set object.

`char-set-immutable?` *char-set* [Function]

Returns `#t` if *char-set* is an immutable char-set, `#f` if it's a mutable char-set.

`char-set-contains?` *char-set char* [Function]

[SRFI-14] Returns true if and only if a character set object *char-set* contains a character *char*.

```
(char-set-contains? #[a-z] #\y) ⇒ #t
(char-set-contains? #[a-z] #\3) ⇒ #f

(char-set-contains? #[^ABC] #\A) ⇒ #f
(char-set-contains? #[^ABC] #\D) ⇒ #t
```

`char-set char` [Generic application]

A char-set object can be applied to a character, and it works just like `(char-set-contains? char-set char)`.

```
(#[a-z] #\a) ⇒ #t
(#[a-z] #\A) ⇒ #f

(use gauche.collection)
(filter #[a-z] "CharSet") ⇒ (#\h #\a #\r #\e #\t)
```

`char-set` *char ...* [Function]

[SRFI-14] Creates a character set that contains *char* ....

```
(char-set #\a #\b #\c)   ⇒ #[a-c]
```

`char-set-size` *char-set* [Function]

[SRFI-14] Returns a number of characters in the given charset.

```
gosh> (char-set-size #[])
0
gosh> (char-set-size #[[:alnum:]])
62
```

`char-set-copy` *char-set* [Function]

[SRFI-14] Copies a character set *char-set*.

`char-set-complement` *char-set* [Function]
`char-set-complement!` *char-set* [Function]

[SRFI-14] Returns a complement set of *char-set*. The former always returns a new set, while the latter may reuse the given charset.

6.12 Strings

<string> [Builtin Class]

A string class. In Gauche, a string can be viewed in two ways: a sequence of characters, or a sequence of bytes.

It should be emphasized that Gauche's *internal* string object, *string body*, is immutable. To comply R7RS in which strings are mutable, a Scheme-level string object is an indirect pointer to a string body. Mutating a string means that Gauche creates a new immutable string body that reflects the changes, then swap the pointer in the Scheme-level string object.

This may affect some assumptions on the cost of string operations.

- Copying string is O(1), no matter how long the string is, since the same string body is shared.
- Taking substring usually is also O(1), for the resulting string shares the substring of the original string body. Gauche may copy a part of the string for better memory management, but the visible cost should stay pretty close to O(1). (However, note that accessing to a specific point by index within the original string may cost O(N) because of multibyte string; which is a different story).
- On the other hand, mutating a string cost O(N) where N is the length of string, even for replacing a character.

Gauche does not attempt to make string mutation faster; (string-set! s k c) is exactly as slow as to take two substrings, before and after of k-th character, and concatenate them with a single-character string inbetween. So, just avoid string mutations; we believe it's a better practice. See also Section 6.12.3 [String Constructors], page 155.

R7RS string operations are very minimal. Gauche supports some extra built-in operations, and also a rich string library defined in SRFI-13. See Section 11.5 [String library], page 591, for details about SRFI-13.

6.12.1 String syntax

"..." [Reader Syntax]

[R7RS+] Denotes a literal string. Inside the double quotes, the following backslash escape sequences are recognized.

\" [R7RS] Double-quote character

\\ [R7RS] Backslash character

\n [R7RS] Newline character (ASCII 0x0a).

\r [R7RS] Return character (ASCII 0x0d).

\f Form-feed character (ASCII 0x0c).

\t [R7RS] Tab character (ASCII 0x09)

\a [R7RS] Alarm character (ASCII 0x07).

\b [R7RS] Backspace character (ASCII 0x08).

\0 ASCII NUL character (ASCII 0x00).

\<whitespace>*<newline><whitespace>*
[R7RS] Ignored. This can be used to break a long string literal for readability. This escape sequence is introduced in R6RS.

`\xN;` [R7RS] A character whose Unicode codepoint is represented by hexadecimal number *N*, which is any number of hexadecimal digits. (See the compatibility notes below.)

`\uNNNN` A character whose UCS2 code is represented by four-digit hexadecimal number *NNNN*.

`\UNNNNNNNN`
A character whose UCS4 code is represented by eight-digit hexadecimal number *NNNNNNNN*.

The following code is an example of backslash-newline escape sequence:

```
(define *message* "\
  This is a long message \
  in a literal string.")

*message*
  ⇒ "This is a long message in a literal string."
```

Note the whitespace just after 'message'. Since any whitespaces before 'in' is eaten by the reader, you have to put a whitespace between 'message' and the following backslash. If you want to include an actual newline character in a string, and any indentation after it, you can put '\n' in the next line like this:

```
(define *message/newline* "\
  This is a long message, \
  \n   with a line break.")
```

Note for the compatibility: We used to recognize a syntax `\xNN` (two-digit hexadecimal number, without semicolon terminator) as a character in a string; for example, `"\x0d\x0a"` was the same as `"\r\n"`. We still support it when we don't see the terminating semicolon, for the compatibility. There are ambiguous cases: `"\0x0a;"` means `"\n"` in the current syntax, while `"\n;"` in the legacy syntax.

Setting the reader mode to `legacy` restores the old behavior. Setting the reader mode to `warn-legacy` makes it work like the default behavior, but prints warning when it finds legacy syntax. See Section 6.22.7.2 [Reader lexical mode], page 228, for the details.

`#*"..."` [Reader Syntax]

Denotes incomplete string. The same escape sequences as the complete string syntax are recognized.

Rationale of the syntax: '`#*`' is used for bit vector in Common Lisp. Since an incomplete strings is really a byte vector, it has similarity. (Bit vector can be added later, if necessary, and two can coexist).

6.12.2 String Predicates

`string?` *obj* [Function]

[R7RS base] Returns `#t` if *obj* is a string, `#f` otherwise.

`string-immutable?` *obj* [Function]

Returns `#t` if *obj* is an immutable string, `#f` otherwise

`string-incomplete?` *obj* [Function]

Returns `#t` if *obj* is an incomplete string, `#f` otherwise

6.12.3 String Constructors

`make-string` *k :optional char* [Function]

[R7RS base] Returns a string of length *k*. If optional *char* is given, the new string is filled with it. Otherwise, the string is filled with a whitespace. The result string is always complete.

```
(make-string 5 #\x) ⇒ "xxxxx"
```

Note that the algorithm to allocate a string by `make-string` and then fills it one character at a time is *extremely* inefficient in Gauche, and should be avoided.

In Gauche, a string is simply a pointer to an immutable string content. If you mutate a string by, e.g. `string-set!`, Gauche allocates whole new immutable string content, copies the original content with modification, then swap the pointer of the original string. It is no more efficient than making a new copy.

You can use an output string port for a string construction (see Section 6.22.5 [String ports], page 224). Even creating a list of characters and using `list->string` is faster than using `make-string` and `string-set!`.

`make-byte-string` *k :optional byte* [Function]

Creates and returns an incomplete string o size *k*. If *byte* is given, which must be an exact integer, and its lower 8 bits are used to initialize every byte in the created string.

`string` *char ...* [Function]

[R7RS base] Returns a string consisted by *char* ....

`x->string` *obj* [Generic Function]

A generic coercion function. Returns a string representation of *obj*. The default methods are defined as follows: strings are returned as is, numbers are converted by `number->string`, symbols are converted by `symbol->string`, and other objects are converted by `display`.

Other class may provide a method to customize the behavior.

6.12.4 String interpolation

The term "string interpolation" is used in various scripting languages such as Perl and Python to refer to the feature to embed expressions in a string literal, which are evaluated and then their results are inserted into the string literal at run time.

Scheme doesn't define such a feature, but Gauche implements it as a reader macro.

`#string-literal` [Reader Syntax]

Evaluates to a string. If *string-literal* contains the character sequence `~expr`, where *expr* is a valid external representation of a Scheme expression, *expr* is evaluated and its result is inserted in the original place (by using `x->string`, see Section 6.12.3 [String Constructors], page 155).

The tilde and the following expression must be adjacent (without containing any whitespace characters), or it is not recognized as a special sequence.

To include a tilde itself immediately followed by non-delimiting character, use `~~`.

Other characters in the *string-literal* are copied as is.

If you use a variable as *expr* and need to delimit it from the subsequent string, you can use the symbol escape syntax using '`|`' character, as shown in the last two examples below.

```
#"This is Gauche, version ~(gauche-version)."
 ⇒ "This is Gauche, version 0.9.9."

#"Date: ~(sys-strftime \"%Y/%m/%d\" (sys-localtime (sys-time)))"
 ⇒ "Date: 2002/02/18"
```

```
(let ((a "AAA")
      (b "BBB"))
 #"xxx ~a ~b zzz")
 ⇒ "xxx AAA BBB zzz"

#"123~~456~~789"
 ⇒ "123~456~789"

(let ((n 7)) #"R~|n|RS")
 ⇒ "R7RS"

(let ((x "bar")) #"foo~|x|.")
 ⇒ "foobar"
```

In fact, the reader expands this syntax into a macro call, which is then expanded into a call of `string-append` as follows:

```
#"This is Gauche, version ~(gauche-version)."
 ≡
(string-interpolate* ("This is Gauche, version "
                      (gauche-version)
                      "."))

;; then, it expands to...

(string-append "This is Gauche, version "
               (x->string (gauche-version))
               ".")
```

(NB: The exact spec of `string-interpolate*` might change in future, so do not rely on the current behavior.)

Since the `#"..."` syntax is equivalent to a macro call of `string-interpolate*`, which is provided in the Gauche module, it must be visible from where you use the interpolation syntax. When you write Gauche code, typically you implicitly inherit the Gauche module so you don't need to worry; however, if you start from R7RS code, make sure you import `string-interpolate*` (by `(import (gauche base))`, for example) whenever you use string interpolation syntax. Also be careful not to shadow `string-interpolate*` locally.

`#‘`*`string-literal`* [Reader Syntax]

This is the old style of string-interpolation. It is still recognized, but discouraged for the new code.

Inside *string-literal*, you can use `,`*`expr`* (instead of `~`*`expr`*) to evaluate *expr*. If comma isn't immediately followed by a character starting an expression, it loses special meaning.

```
#‘"This is Gauche, version ,(gauche-version)"
```

Rationale of the syntax: There are wide variation of string interpolation syntax among scripting languages. They are usually linked with other syntax of the language (e.g. prefixing `$` to mark evaluating place is in sync with variable reference syntax in some languages).

The old style of string interpolation syntax was taken from quasiquote syntax, because those two are conceptually similar operations (see Section 4.9 [Quasiquotation], page 59). However, since comma character is frequently used in string literals, it was rather awkward.

We decided that tilde is more suitable as the unquote character for the following reasons.

- Traditionally, Lisp's string formatter `format` uses `~` to introduce format directives (see Section 6.22.8.4 [Formatting output], page 235). Lispers are used to scan `~`'s in a string as variable portions.
- Gauche's `~` is a universal accessor, and the operator has a nuance of "taking something out of it" (see Section 6.18.2 [Universal accessor], page 189).
- Clojure, a new Lisp dialect, adopted `~` as the unquote character in the quasiquote syntax, instead of commas.

Note that Scheme allows wider range of characters for valid identifier names than usual scripting languages. Consequently, you will almost always need to use '`|`' delimiters when you interpolate the value of a variable. For example, while you can write `"$year/$month/$day $hour:$minutes:$seconds"` in Perl, you should write `#"~|year|/~|month|/~day ~|hour|:~|minutes|:~seconds"`. It may be better always to delimit direct variable references in this syntax to avoid confusion.

6.12.5 String Accessors & Modifiers

`string-length` *string* [Function]

[R7RS base] Returns a length of (possibly incomplete) string *string*.

`string-size` *string* [Function]

Returns a size of (possibly incomplete) *string*. A size of string is a number of bytes *string* occupies on memory. The same string may have different sizes if the native encoding scheme differs.

For incomplete string, its length and its size always match.

`string-ref` *cstring k :optional fallback* [Function]

[R7RS+] Returns *k*-th character of a complete string *cstring*. It is an error to pass an incomplete string.

By default, an error is signaled if `k` is out of range (negative, or greater than or equal to the length of *cstring*). However, if an optional argument *fallback* is given, it is returned in such case. This is Gauche's extension.

`string-byte-ref` *string k* [Function]

Returns *k-th* byte of a (possibly incomplete) string *string*. Returned value is an integer in the range between 0 and 255. *k* must be greater than or equal to zero, and less than `(string-size string)`.

`string-set!` *string k char* [Function]

[R7RS base] Substitute *string*'s *k*-th character by *char*. *k* must be greater than or equal to zero, and less than `(string-length string)`. Return value is undefined.

If *string* is an incomplete string, integer value of the lower 8 bits of *char* is used to set *string*'s *k*-th byte.

See the notes in `make-string` about performance consideration.

`string-byte-set!` *string k byte* [Function]

Substitute *string*'s *k*-th byte by integer *byte*. *byte* must be in the range between 0 to 255, inclusive. *k* must be greater than or equal to zero, and less than `(string-size string)`. If *string* is a complete string, it is turned to incomplete string by this operation. Return value is undefined.

6.12.6 String Comparison

`string=?` *string1 string2 string3 . . .* [Function]

[R7RS base] Returns `#t` iff all arguments are strings with the same content.

If any of arguments is incomplete string, it returns `#t` iff all arguments are incomplete and have exactly the same content. In other words, a complete string and an incomplete string never equal to each other.

`string<?` *string1 string2 string3 . . .* [Function]
`string<=?` *string1 string2 string3 . . .* [Function]
`string>?` *string1 string2 string3 . . .* [Function]
`string>=?` *string1 string2 string3 . . .* [Function]

[R7RS base] Compares strings in codepoint order. Returns `#t` iff all the arguments are ordered.

Comparison between an incomplete string and a complete string, or between two incomplete strings, are done by octet-to-octet comparison. If a complete string and an incomplete string have exactly the same binary representation of the content, a complete string is smaller.

`string-ci=?` *string1 string2 string3 . . .* [Function]
`string-ci<?` *string1 string2 string3 . . .* [Function]
`string-ci<=?` *string1 string2 string3 . . .* [Function]
`string-ci>?` *string1 string2 string3 . . .* [Function]
`string-ci>=?` *string1 string2 string3 . . .* [Function]

Case-insensitive string comparison.

These procedures fold argument character-wise, according to Unicode-defined character-by-character case mapping. See `char-foldcase` for the details (Section 6.10 [Characters], page 143). Character-wise case folding doesn't handles the case like German eszett:

```
(string-ci=? "\u00df" "SS") ⇒ #f
```

R7RS requires `string-ci*` procedures to use string case folding. Gauche provides R7RS-conformant case insensitive comparison procedures in `gauche.unicode` (see Section 9.35.3 [Full string case conversion], page 475). If you write in R7RS, importing `(scheme char)` library, you'll use `gauche.unicode`'s `string-ci*` procedures.

6.12.7 String utilities

`substring` *string start end* [Function]

[R7RS base] Returns a substring of *string*, starting from *start*-th character (inclusive) and ending at *end*-th character (exclusive). The *start* and *end* arguments must satisfy `0 <= start < N`, `0 <= end <= N`, and `start <= end`, where N is the length of the string.

When *start* is zero and *end* is N, this procedure returns a copy of *string*.

Actually, extended `string-copy` explained below is a superset of `substring`. This procedure is kept mostly for compatibility of R7RS programs. See also `subseq` in Section 9.29 [Sequence framework], page 441, for the generic version.

`string-append` *string . . .* [Function]

[R7RS base] Returns a newly allocated string whose content is concatenation of *string* . . . .

See also `string-concatenate` in Section 11.5.9 [SRFI-13 String reverse & append], page 597.

`string->list` *string :optional start end* [Function]
`list->string` *list* [Function]

[R7RS base] Converts a string to a list of characters or vice versa.

You can give an optional start/end indexes to `string->list`.

For `list->string`, every elements of *list* must be a character, or an error is signaled. If you want to build a string out of a mixed list of strings and characters, you may want to use `tree->string` in Section 12.61 [Lazy text construction], page 793.

`string-copy` *string :optional start end* [Function]

[R7RS base] Returns a copy of *string*. You can give *start* and/or *end* index to extract the part of the original string (it makes `string-copy` a superset of `substring` effectively).

If only *start* argument is given, a substring beginning from *start*-th character (inclusive) to the end of *string* is returned. If both *start* and *end* argument are given, a substring from *start*-th character (inclusive) to *end*-th character (exclusive) is returned. See `substring` above for the condition that *start* and *end* should satisfy.

Node: R7RS's destructive version `string-copy!` is provided by `srfi-13` module (see Section 11.5 [String library], page 591).

`string-fill!` *string char :optional start end* [Function]

[R7RS base] Fills *string* by *char*. Optional *start* and *end* limits the effective area.

```
(string-fill! "orange" #\X)
  ⇒ "XXXXXX"
(string-fill! "orange" #\X 2 4)
  ⇒ "orXXge"
```

See the notes in `make-string` about performance consideration.

`string-join` *strs :optional delim grammer* [Function]

[SRFI-13] Concatenate strings in the list *strs*, with a string *delim* as 'glue'.

The argument *grammer* may be one of the following symbol to specify how the strings are concatenated.

`infix` Use *delim* between each string. This mode is default. Note that this mode introduce ambiguity when *strs* is an empty string or a list with a null string.

```
(string-join '("apple" "mango" "banana") ", ")
  ⇒ "apple, mango, banana"
(string-join '() ":")
  ⇒ ""
(string-join '("") ":")
  ⇒ ""
```

`strict-infix`
Works like `infix`, but empty list is not allowed to *strs*, thus avoiding ambiguity.

`prefix` Use *delim* before each string.

```
(string-join '("usr" "local" "bin") "/" 'prefix)
  ⇒ "/usr/local/bin"
(string-join '() "/" 'prefix)
  ⇒ ""
(string-join '("") "/" 'prefix)
  ⇒ "/"
```

`suffix` Use *delim* after each string.

```
(string-join '("a" "b" "c") "&" 'suffix)
  ⇒ "a&b&c&"
(string-join '() "&" 'suffix)
  ⇒ ""
(string-join '("") "&" 'suffix)
  ⇒ "&"
```

string-scan *string item :optional return* [Function]
string-scan-right *string item :optional return* [Function]

Scan *item* (either a string or a character) in *string*. While `string-scan` finds the leftmost match, `string-scan-right` finds the rightmost match.

The *return* argument specifies what value should be returned when *item* is found in *string*. It must be one of the following symbols.

`index` Returns the index in *string* if *item* is found, or `#f`. This is the default behavior.

```
(string-scan "abracadabra" "ada") ⇒ 5
(string-scan "abracadabra" #\c) ⇒ 4
(string-scan "abracadabra" "aba") ⇒ #f
```

`before` Returns a substring of *string* before *item*, or `#f` if *item* is not found.

```
(string-scan "abracadabra" "ada" 'before) ⇒ "abrac"
(string-scan "abracadabra" #\c 'before) ⇒ "abra"
```

`after` Returns a substring of *string* after *item*, or `#f` if *item* is not found.

```
(string-scan "abracadabra" "ada" 'after) ⇒ "bra"
(string-scan "abracadabra" #\c 'after) ⇒ "adabra"
```

`before*` Returns a substring of *string* before *item*, and the substring after it. If *item* is not found, returns `(values #f #f)`.

```
(string-scan "abracadabra" "ada" 'before*)
  ⇒ "abrac" and "adabra"
(string-scan "abracadabra" #\c 'before*)
  ⇒ "abra" and "cadabra"
```

`after*` Returns a substring of *string* up to the end of *item*, and the rest. If *item* is not found, returns `(values #f #f)`.

```
(string-scan "abracadabra" "ada" 'after*)
  ⇒ "abracada" and "bra"
(string-scan "abracadabra" #\c 'after*)
  ⇒ "abrac" and "adabra"
```

`both` Returns a substring of *string* before *item* and after *item*. If *item* is not found, returns `(values #f #f)`.

```
(string-scan "abracadabra" "ada" 'both)
  ⇒ "abrac" and "bra"
(string-scan "abracadabra" #\c 'both)
  ⇒ "abra" and "adabra"
```

string-split *string splitter :optional grammar limit start end* [Function]
string-split *string splitter :optional limit start end* [Function]

[SRFI-152+] Splits *string* by *splitter* and returns a list of strings. *splitter* can be a character, a character set, a string, a regexp, or a procedure.

If *splitter* is a character or a string, it is used as a delimiter. Note that srfi-152's `string-split` only allows strings for *delimiter* (it also interprets the first optional argument as a grammer; see below for the compatibility note.)

If *splitter* is a character set, any consecutive characters that are member of the character set are used as a delimiter.

If a procedure is given to *splitter*, it is called for each character in *string*, and the consecutive characters that caused *splitter* to return a true value are used as a delimiter.

```
(string-split "/aa/bb//cc" #\/)    ⇒ ("" "aa" "bb" "" "cc")
```

```
(string-split "/aa/bb//cc" "/")    ⇒ ("" "aa" "bb" "" "cc")
(string-split "/aa/bb//cc" "//")   ⇒ ("/aa/bb" "cc")
(string-split "/aa/bb//cc" #[/])   ⇒ ("" "aa" "bb" "cc")
(string-split "/aa/bb//cc" #/\/+/) ⇒ ("" "aa" "bb" "cc")
(string-split "/aa/bb//cc" #[\w])  ⇒ ("/" "/" "//" "")
(string-split "/aa/bb//cc" char-alphabetic?) ⇒ ("/" "/" "//" "")

;; some boundary cases
(string-split "abc" #\/) ⇒ ("abc")
(string-split ""    #\/) ⇒ ("")
```

The *grammar* argument is the same as `string-join` above; it must be one of symbols `infix`, `strict-infix`, `prefix` or `suffix`. When omitted, `infix` is assumed.

```
(string-split "/a/b/c/" "/" 'infix)  ⇒ ("" "a" "b" "c" "")
(string-split "/a/b/c/" "/" 'prefix) ⇒ ("a" "b" "c" "")
(string-split "/a/b/c/" "/" 'suffix) ⇒ ("" "a" "b" "c")
```

In general, the following relationship holds:

```
(string-join XS DELIM GRAMMAR) ⇒ S
(string-split S DELIM GRAMMAR) ⇒ XS
```

If *limit* is given and not `#f`, it must be a nonnegative integer and specifies the maximum number of match to the *splitter*. Once the limit is reached, the rest of string is included in the result as is.

```
(string-split "a.b..c" "." 'infix 0)   ⇒ ("a.b..c")
(string-split "a.b..c" "." 'infix 1)   ⇒ ("a" "b..c")
(string-split "a.b..c" "." 'infix 2)   ⇒ ("a" "b" ".c")
```

Compatibility note: The *grammar* argument is added for the consistency of srfis (srfi-130, srfi-152, see Section 11.35 [String library (reduced)], page 633). However, for the backward compatibility and the convenience, it also accepts *limit* without *grammar* argument; it is distinguishable since *grammar* is a symbol and *limit* is an integer. For the code that's compatible to srfi-152, use the first form that takes *grammar* argument.

```
(string-split "a.b..c" "." 2)    ⇒ ("a" "b" ".c")
```

The *start* and *end* arguments limits input string in the given range before splitting.

See also `string-tokenize` in (see Section 11.5.12 [SRFI-13 other string operations], page 599).

6.12.8 Incomplete strings

A string can be flagged as "incomplete" if it may contain byte sequences that do not consist of a valid multibyte character in the Gauche's native encoding.

Incomplete strings may be generated in several circumstances; reading binary data as a string, reading a string data that has been 'chopped' in middle of a multibyte character, or concatenating a string with other incomplete strings, for example.

Incomplete strings should be regarded as an exceptional case. It used to be a way to handle byte strings, but now we have u8vector (see Section 9.36 [Uniform vectors], page 476) for that purpose. In fact, we're planning to remove it in the future releases.

Just in case, if you happen to get an incomplete string, you can convert it to a complete string by the following procedure:

`string-incomplete->complete` *str :optional handling filler* [Function]

Reinterpret the content of an incomplete string *str* and returns a newly created complete string from it. The *handling* argument specifies how to handle the illegal byte sequences in *str*.

`#f` If *str* contains an illegal byte sequence, give up the conversion and returns `#f`. This is the default behavior.

`:omit` Omit any illegal byte sequences.

`:replace` Replace each byte in illegal byte sequences by a character given in *filler* argument, defaulted to `?`.

`:escape` Replace each byte in illegal byte sequences by a sequence of *filler* `<hexdigit>` `<hexdigit>`. Besides, the *filler* characters in the original string is replaced with *filler filler*.

If *str* is already a complete string, its copy is returned.

The procedure always returns a complete string, except when the *handling* argument is `#f` (default) and the input is an incomplete string, in which case `#f` is returned.

When Gauche's internal encoding is utf-8, the procedure works as follows:

```
(string-incomplete->complete #*"_abc")
  ⇒ "_abc"     ; can be represented as a complete string

(string-incomplete->complete #*"_ab\x80;c")
  ⇒ #f        ; can't be represented as a complete string

(string-incomplete->complete #*"_ab\x80;c" :omit)
  ⇒ "_abc"     ; omit the illegal bytes

(string-incomplete->complete #*"_ab\x80;c" :replace #\_)
  ⇒ "_ab_c"    ; replace the illegal bytes

(string-incomplete->complete #*"_ab\x80;c" :escape #\_)
  ⇒ "__ab_80c" ; escape the illegal bytes and escape char itself
```

6.13 Regular expressions

Gauche has a built-in regular expression engine which is mostly upper-compatible of POSIX extended regular expression, plus some extensions from Perl 5 regexp.

A special syntax is provided for literal regular expressions. Also regular expressions are applicable, that is, it works like procedures that match the given string to itself. Combining with these two features enables writing some string matching idioms compact.

```
(find #/pattern/ list-of-strings)
  ⇒ match object or #f
```

6.13.1 Regular expression syntax

`#/regexp-spec/` [Reader Syntax]
`#/regexp-spec/i` [Reader Syntax]

Denotes literal regular expression object. When read, it becomes an instance of `<regexp>`.

If a letter 'i' is given at the end, the created regexp becomes *case-folding regexp*, i.e. it matches in the case-insensitive way.

The advantage of using this syntax over `string->regexp` is that the regexp is compiled only once. You can use literal regexp inside loop without worrying about regexp compilation overhead. If you want to construct regexp on-the-fly, however, use `string->regexp`.

Gauche's built-in regexp syntax follows POSIX extended regular expression, with a bit of extensions taken from Perl.

Note that the syntax described here is just a surface syntax. Gauche's regexp compiler works on the abstract syntax tree, and alternative syntax such as SRE will be supported in the future versions.

*re** Matches zero or more repetition of *re*.

re+ Matches one or more repetition of *re*.

re? Matches zero or one occurrence of *re*.

re{*n*}
re{*n*,*m*} Bounded repetition. *re*{*n*} matches exactly *n* occurrences of *re*. *re*{*n*,*m*} matches at least *n* and at most *m* occurrences of *re*, where *n* <= *m*. In the latter form, either *n* or *m* can be omitted; omitted *n* is assumed as 0, and omitted *m* is assumed infinity.

*re**?
re+?
re??
re{*n*,*m*}? Same as the above repetition construct, but these syntaxes use "non-greedy" or "lazy" match strategy. That is, they try to match the minimum number of occurrences of *re* first, then retry longer ones only if it fails. In the last form either *n* or *m* can be omitted. Compare the following examples:

```
(rxmatch-substring (#/<.*>/ "<tag1><tag2><tag3>") 0)
  ⇒ "<tag1><tag2><tag3>"

(rxmatch-substring (#/<.*?>/ "<tag1><tag2><tag3>") 0)
  ⇒ "<tag1>"
```

(*re*...) Clustering with capturing. The regular expression enclosed by parenthesis works as a single *re*. Besides, the string that matches *re* ... is saved as a *submatch*.

(?:*re*...)
Clustering without capturing. re ... works as a single *re*, but the matched string isn't saved.

(?<*name*>*re*...)
Named capture and clustering. Like (*re*...), but adds the name *name* to the matched substring. You can refer to the matched substring by both index number and the name.

When the same name appears more than once in a regular expression, it is undefined which matched substring is returned as the submatch of the named capture.

(?i:*re*...)
(?-i:*re*...)
Lexical case sensitivity control. (?i:*re*...) makes *re*... matches case-insensitively, while (?-i:*re*...) makes *re*... matches case-sensitively.

Perl's regexp allows several more flags to appear between '?' and ':'. Gauche only supports above two, for now.

pattern1|*pattern2*|...
Alternation. Matches either one of patterns, where each pattern is *re* ....

n Backreference. *n* is an integer. Matches the substring captured by the *n*-th capturing group. (counting from 1). When capturing groups are nested, groups are counted by their beginnings. If the *n*-th capturing group is in a repetition and has matched more than once, the last matched substring is used.

`\k<name>` Named backreference. Matches the substring captured by the capturing group with the name *name*. If the named capturing group is in a repetition and has matched more than once, the last matched substring is used. If there are more than one capturing group with *name*, matching will succeed if the input matches either one of the substrings captured by those groups.

`.` Matches any character (including newline).

`[char-set-spec]`
: Matches any of the character set specified by *char-set-spec*. See Section 6.11 [Character set], page 147, for the details of *char-set-spec*.

`\s, \d, \w`
: Matches a whitespace character (`char-set:ascii-whitespace`, `#[\u0009-\u000d ]`), a digit character(`char-set:ascii-digit`, `#[0-9]`), or a word-constituent character (`char-set:ascii-word`, `#[A-Za-z0-9_]`), respectively. Note that they don't include characters outside ASCII range.

  Can be used both inside and outside of character set.

`\S, \D, \W`
: Matches the complement character set of `\s`, `\d` and `\w`, respectively.

`^, $` Beginning and end of string assertion, when appears at the beginning or end of the pattern, or optionally, beginning and end of line in multi-line mode.

  These characters loses special meanings and matches the characters themselves if they appear in the position other than the beginning of the pattern (for `^`) or the end (for `$`). For the sake of recognizing those characters, lookahead/lookbehind assertions (`(?=...)`, `(?!...)`, `(?<=...)`, `(?<!...)`) and atomic clustering (`(?>...)`) are treated as if they are a whole pattern. That is, `^` at the beginning of those groupings are beginning-of-string assertion no matter where these group appear in the containing regexp. So as `$` at the end of these groupings.

`\b, \B` Word boundary and non word boundary assertion, respectively. That is, `\b` matches an empty string between word-constituent character and non-word-constituent character, and `\B` matches an empty string elsewhere.

`\;`
`\"`
`\#` These are the same as `;`, `"`, and `#`, respectively, and can be used to avoid confusing Emacs or other syntax-aware editors that are not familiar with Gauche's extension.

`(?=pattern)`
`(?!pattern)`
: Positive/negative lookahead assertion. Match succeeds if *pattern* matches (or does not match) the input string from the current position, but this doesn't move the current position itself, so that the following regular expression is applied again from the current position.

  For example, the following expression matches strings that might be a phone number, except the numbers in Japan (i.e. ones that begin with "81").

```
\+(?!81)\d{9,}
```

`(?<=pattern)`
`(?<!pattern)`
: Positive/negative lookbehind assertion. If the input string immediately before the current input position matches *pattern*, this pattern succeeds or fails, respectively. Like lookahead assertion, the input position isn't changed.

Internally, this match is tried by reversing *pattern* and applies it to the backward of input character sequence. So you can write any regexp in *pattern*, but if the submatches depend on the matching order, you may get different submatches from when you match *pattern* from left to right.

`(?>pattern)`
: Atomic clustering. Once *pattern* matches, the match is fixed; even if the following pattern fails, the engine won't backtrack to try the alternative match in *pattern*.

`re*+`
`re++`
`re?+`
: They are the same as (?>*re**), (?>*re*+), (?>*re*?), respectively.

`(?test-pattern then-pattern)`
`(?test-pattern then-pattern|else-pattern)`
: Conditional matching. If *test-pattern* counts true, *then-pattern* is tried; otherwise *else-pattern* is tried when provided.

test-pattern can be either one of the following:

`(integer)`
: Backreference. If *integer*-th capturing group has a match, this test counts true.

`(?=pattern)`
`(?!pattern)`
: Positive/negative lookahead assertion. It tries *pattern* from the current input position without consuming input, and if the match succeeds or fails, respectively, this test counts true.

`(?<=pattern)`
`(?<!pattern)`
: Positive/negative lookbehind assertion. It tries *pattern* backward from the left size of the current input position, and if the match succeeds or fails, respectively, this test counts true.

6.13.2 Using regular expressions

Regexp object and rxmatch object

`<regexp>` [Builtin Class]

Regular expression object. You can construct a regexp object from a string by `string->regexp` at run time. Gauche also has a special syntax to denote regexp literals, which construct regexp object at loading time.

Gauche's regexp engine is fully aware of multibyte characters.

`<regmatch>` [Builtin Class]

Regexp match object. A regexp matcher `rxmatch` returns this object if match. This object contains all the information about the match, including submatches.

The advantage of using match object, rather than substrings or list of indices, is efficiency. The regmatch object keeps internal state of match, and computes indices and/or substrings only when requested. This is particularly effective for multibyte strings, for index access is slow on them.

`string->regexp` *string :key case-fold multi-line* [Function]

Takes *string* as a regexp specification, and constructs an instance of `<regexp>` object.

If a true value is given to the keyword argument *case-fold*, the created regexp object becomes case-folding regexp. (See the above explanation about case-folding regexp).

If a true value is given to the keyword argument *multi-line*, `^` and `$` will assert the beginning and end of line in addition to beginning and end of string. Popular line terminators (LF only, CRLF and CR only) are recognized.

`regexp?` *obj* [Function]

Returns true iff *obj* is a regexp object.

`regexp->string` *regexp* [Function]

Returns a source string describing the regexp *regexp*. The returned string is immutable.

`regexp-num-groups` *regexp* [Function]
`regexp-named-groups` *regexp* [Function]

Queries the number of capturing groups, and an alist of named capturing groups, in the given *regexp*, respectively.

The number of capturing groups corresponds to the number of matches returned by `rxmatch-num-matches`. Note that the entire regexp forms a group, so the number is always positive.

The alist returned from `regexp-named-groups` has the group name (symbol) in `car`, and its subgroup number in `cdr`. Note that the order of groups in the alist isn't fixed.

```
(regexp-num-groups #/abc(?<foo>def)(ghi(?<bar>jkl)(mno))/)
  ⇒ 5
(regexp-named-groups #/abc(?<foo>def)(ghi(?<bar>jkl)(mno))/)
  ⇒ ((bar . 3) (foo . 1))
```

Trying a match

`rxmatch` *regexp string :optional start end* [Function]

Regexp is a regular expression object. A string *string* is matched by *regexp*. If it matches, the function returns a `<regmatch>` object. Otherwise it returns `#f`.

If *start* and/or *end* are given, only the substring between *start* (inclusive) and *end* (exclusive) is searched.

This is called `match`, `regexp-search` or `string-match` in some other Scheme implementations.

Internally, Gauche uses backtracking for regexp match. When regexp has multiple match possibilities, Gauche saves an intermediate result in a stack and try one choice, and if it fails try another. Depending on regexp, the saved results may grow linear to the input. Gauche allocates a fixed amount of memory for that, and if there are too many saved results, you'll get the following error:

```
ERROR: Ran out of stack during matching regexp #/.../. Too many retries?
```

If you get this error, consider using hybrid parsing approach. Our regexp engine isn't made to do everything-in-one-shot parsing; in most cases, the effect of complex regexp can be achieved better with more powerful grammar than regular grammar.

To apply the match repeatedly on the input string, or to match from the input stream (such as the data from the port), you may want to check `grxmatch` in `gauche.generator` (see Section 9.11.2 [Generator operations], page 376).

`regexp string` [Generic application]

A regular expression object can be applied directly to the string. This works the same as `(rxmatch` *`regexp string`*`)`, but allows shorter notation. See Section 6.18.6 [Applicable objects], page 195, for generic mechanism used to implement this.

Accessing the match result

`rxmatch-start` *match :optional* (*i 0*) [Function]
`rxmatch-end` *match :optional* (*i 0*) [Function]
`rxmatch-substring` *match :optional* (*i 0*) [Function]

Match is a match object returned by `rxmatch`. If *i* equals to zero, the functions return start, end or the substring of entire match, respectively. With positive integer *I*, it returns those of *I*-th submatches. It is an error to pass other values to *I*.

It is allowed to pass `#f` to *match* for convenience. The functions return `#f` in such case.

These functions correspond to scsh's `match:start`, `match:end` and `match:substring`.

`rxmatch-after` *match :optional* (*i 0*) [Function]
`rxmatch-before` *match :optional* (*i 0*) [Function]

Returns substring of the input string after or before *match*. If optional argument is given, the *i*-th submatch is used (0-th submatch is the entire match).

```
(define match (rxmatch #/(\d+)\.(\d+)/ "pi=3.14..."))

(rxmatch-after match) ⇒ "..."
(rxmatch-after match 1) ⇒ ".14..."

(rxmatch-before match) ⇒ "pi="
(rxmatch-before match 2) ⇒ "pi=3."
```

`rxmatch-substrings` *match :optional start end* [Function]
`rxmatch-positions` *match :optional start end* [Function]

Retrieves multiple submatches (again, 0-th match is the entire match), in substrings and in a cons of start and end position, respectively.

```
(rxmatch-substrings (#/(\d+):(\d+):(\d+)/ "12:34:56"))
  ⇒ ("12:34:56" "12" "34" "56")

(rxmatch-positions (#/(\d+):(\d+):(\d+)/ "12:34:56"))
  ⇒ ((0 . 8) (0 . 2) (3 . 5) (6 . 8))
```

For the convenience, you can pass `#f` to *match*; those procedures returns `()` in that case.

The optional *start* and *end* arguments specify the range of submatch index. If omitted, *start* defaults to 0 and *end* defaults to `(rxmatch-num-matches match)`. For example, if you don't need the whole match, you can give `1` to *start* as follows:

```
(rxmatch-substrings (#/(\d+):(\d+):(\d+)/ "12:34:56") 1)
  ⇒ ("12" "34" "56")
```

`rxmatch->string` *regexp string :optional selector . . .* [Function]

A convenience procedure to match a string to the given regexp, then returns the matched substring, or `#f` if it doesn't match.

If no *selector* is given, it is the same as this:

```
(rxmatch-substring (rxmatch regexp string))
```

If an integer is given as a selector, it returns the substring of the numbered submatch.

If a symbol `after` or `before` is given, it returns the substring after or before the match. You can give these symbols and an integer to extract a substring before or after the numbered submatch.

```
gosh> (rxmatch->string #/\d+/ "foo314bar")
"314"
```

```
gosh> (rxmatch->string #/(\w+)@([\w.]+)/ "foo@example.com" 2)
"example.com"
gosh> (rxmatch->string #/(\w+)@([\w.]+)/ "foo@example.com" 'before 2)
"foo@"
```

regmatch :optional index [Generic application]
regmatch 'before *:optional index* [Generic application]
regmatch 'after *:optional index* [Generic application]

A regmatch object can be applied directly to the integer index, or a symbol `before` or `after`. They works the same as `(rxmatch-substring` *regmatch index*`)`, `(rxmatch-before` *regmatch*`)`, and `(rxmatch-after` *regmatch*`)`, respectively. This allows shorter notation. See Section 6.18.6 [Applicable objects], page 195, for generic mechanism used to implement this.

```
(define match (#/(\d+)\.(\d+)/ "pi=3.14..."))

  (match)           ⇒ "3.14"
  (match 1)         ⇒ "3"
  (match 2)         ⇒ "14"

  (match 'after)    ⇒ "..."
  (match 'after 1)  ⇒ ".14..."

  (match 'before)   ⇒ "pi="
  (match 'before 2) ⇒ "pi=3."

(define match (#/(?<integer>\d+)\.(?<fraction>\d+)/ "pi=3.14..."))

  (match 1)         ⇒ "3"
  (match 2)         ⇒ "14"

  (match 'integer)  ⇒ "3"
  (match 'fraction) ⇒ "14"

  (match 'after 'integer)   ⇒ ".14..."
  (match 'before 'fraction) ⇒ "pi=3."
```

`rxmatch-num-matches` *match* [Function]
`rxmatch-named-groups` *match* [Function]

Returns the number of matches, and an alist of named groups and whose indices, in *match*. This corresponds `regexp-num-groups` and `regexp-named-groups` on a regular expression that has been used to generate *match*. These procedures are useful to inspect *match* object without having the original regexp object.

The number of matches includes the "whole match", so it is always a positive integer for a `<regmatch>` object. The number also includes the submatches that don't have value (see the examples below). The result of `rxmatch-named-matches` also includes all the named groups in the original regexp, not only the matched ones.

For the convenience, `rxmatch-num-matches` returns 0 and `rxmatch-named-groups` returns `()` if *match* is `#f`.

```
(rxmatch-num-matches (rxmatch #/abc/ "abc")) ⇒ 1
(rxmatch-num-matches (rxmatch #/(a(.))|(b(.))/ "ba")) ⇒ 5
(rxmatch-num-matches #f) ⇒ 0

(rxmatch-named-groups
```

```
(rxmatch #/(?<h>\d\d):(?<m>\d\d)(:(?<s>\d\d))?/ "12:34"))
⇒ ((s . 4) (m . 2) (h . 1))
```

Convenience utilities

regexp-replace *regexp string substitution* [Function]
regexp-replace-all *regexp string substitution* [Function]

Replaces the part of *string* that matched to *regexp* for *substitution*. `regexp-replace` just replaces the first match of *regexp*, while `regexp-replace-all` repeats the replacing throughout entire *string*.

substitution may be a string or a procedure. If it is a string, it can contain references to the submatches by digits preceded by a backslash (e.g. `\2`) or the named submatch reference (e.g. `\k<name>`. `\0` refers to the entire match. Note that you need two backslashes to include backslash character in the literal string; if you want to include a backslash character itself in the *substitution*, you need four backslashes.

```
(regexp-replace #/def|DEF/ "abcdefghi" "...")
  ⇒ "abc...ghi"
(regexp-replace #/def|DEF/ "abcdefghi" "|\\0|")
  ⇒ "abc|def|ghi"
(regexp-replace #/def|DEF/ "abcdefghi" "|\\\\0|")
  ⇒ "abc|\\0|ghi"
(regexp-replace #/c(.*)g/ "abcdefghi" "|\\1|")
  ⇒ "ab|def|hi"
(regexp-replace #/c(?<match>.*)g/ "abcdefghi" "|\\k<match>|")
  ⇒ "ab|def|hi"
```

If *substitution* is a procedure, for every match in *string* it is called with one argument, regexp-match object. The returned value from the procedure is inserted to the output string using `display`.

```
(regexp-replace #/c(.*)g/ "abcdefghi"
                (lambda (m)
                  (list->string
                   (reverse
                    (string->list (rxmatch-substring m 1))))))
 ⇒ "abfedhi"
```

Note: `regexp-replace-all` applies itself recursively to the remaining of the string after match. So the beginning of string assertion in *regexp* doesn't only mean the beginning of input string.

Note: If you want to operate on multiple matches in the string instead of replacing it, you can use `lrxmatch` in `gauche.lazy` module or `grxmatch` in `gauche.generator` module. Both can match a regexp *repeatedly* and *lazily* to the given string, and `lrxmatch` returns a lazy sequence of `regmatch`es, while `grxmatch` returns a generator that yields `regmatch`es.

```
(map rxmatch-substring (lrxmatch #/\w+/ "a quick brown fox!?"))
 ⇒ ("a" "quick" "brown" "fox")
```

regexp-replace* *string rx1 sub1 rx2 sub2 . . .* [Function]
regexp-replace-all* *string rx1 sub1 rx2 sub2 . . .* [Function]

First applies `regexp-replace` or `regexp-replace-all` to *string* with a regular expression *rx1* substituting for *sub1*, then applies the function on the result string with a regular expression *rx2* substituting for *sub2*, and so on. These functions are handy when you want to apply multiple substitutions sequentially on a string.

`regexp-quote` *string* [Function]

Returns a string with the characters that are special to regexp escaped.

```
(regexp-quote "[2002/10/12] touched foo.h and *.c")
 ⇒ "\\[2002/10/12\\] touched foo\\.h and \\*\\.c"
```

In the following macros, *match-expr* is an expression which produces a match object or `#f`. Typically it is a call of `rxmatch`, but it can be any expression.

`rxmatch-let` *match-expr* (*var* ...) *form* ... [Macro]

Evaluates *match-expr*, and if matched, binds *var* ... to the matched strings, then evaluates *forms*. The first *var* receives the entire match, and subsequent variables receive submatches. If the number of submatches are smaller than the number of variables to receive them, the rest of variables will get `#f`.

It is possible to put `#f` in variable position, which says you don't care that match.

```
(rxmatch-let (rxmatch #/(\d+):(\d+):(\d+)/
                      "Jan  1 23:59:58, 2001")
   (time hh mm ss)
  (list time hh mm ss))
 ⇒ ("23:59:58" "23" "59" "58")

(rxmatch-let (rxmatch #/(\d+):(\d+):(\d+)/
                      "Jan  1 23:59:58, 2001")
   (#f hh mm)
  (list hh mm))
 ⇒ ("23" "59")
```

This macro corresponds to scsh's `let-match`.

`rxmatch-if` *match-expr* (*var* ...) *then-form else-form* [Macro]

Evaluates *match-expr*, and if matched, binds *var* ... to the matched strings and evaluate *then-form*. Otherwise evaluates *else-form*. The rule of binding *vars* is the same as `rxmatch-let`.

```
(rxmatch-if (rxmatch #/(\d+:\d+)/ "Jan 1 11:22:33")
    (time)
  (format #f "time is ~a" time)
  "unknown time")
 ⇒ "time is 11:22"

(rxmatch-if (rxmatch #/(\d+:\d+)/ "Jan 1 11-22-33")
    (time)
  (format #f "time is ~a" time)
  "unknown time")
 ⇒ "unknown time"
```

This macro corresponds to scsh's `if-match`.

`rxmatch-cond` *clause* ... [Macro]

Evaluate condition in *clauses* one by one. If a condition of a clause satisfies, rest portion of the clause is evaluated and becomes the result of `rxmatch-cond`. *Clause* may be one of the following pattern.

`(`*match-expr* `(`*var* `...)` *form* `...)`

Evaluate *match-expr*, which may return a regexp match object or `#f`. If it returns a match object, the matches are bound to *vars*, like *rxmatch-let*, and *forms* are evaluated.

(test *expr* *form* ...)
: Evaluates *expr*. If it yields true, evaluates *form*s.

(test *expr* => *proc*)
: Evaluates *expr* and if it is true, calls *proc* with the result of *expr* as the only argument.

(else *form* ...)
: If this clause exists, it must be the last clause. If other clauses fail, *form*s are evaluated.

If no `else` clause exists, and all the other clause fail, an undefined value is returned.

```
;; parses several possible date format
(define (parse-date str)
  (rxmatch-cond
    ((rxmatch #/^(\d\d?)\/(\d\d?)\/(\d\d\d\d)$/ str)
        (#f mm dd yyyy)
      (map string->number (list yyyy mm dd)))
    ((rxmatch #/^(\d\d\d\d)\/(\d\d?)\/(\d\d?)$/ str)
        (#f yyyy mm dd)
      (map string->number (list yyyy mm dd)))
    ((rxmatch #/^\d+\/\d+\/\d+$/ str)
        (#f)
     (errorf "ambiguous: ~s" str))
    (else (errorf "bogus: ~s" str))))

(parse-date "2001/2/3") ⇒ (2001 2 3)
(parse-date "12/25/1999") ⇒ (1999 12 25)
```

This macro corresponds to scsh's `match-cond`.

`rxmatch-case` *string-expr clause* ... [Macro]

String-expr is evaluated, and *clause*s are interpreted one by one. A *clause* may be one of the following pattern.

(*re* (*var* ...) *form* ...)
: *Re* must be a literal regexp object (see Section 6.13 [Regular expressions], page 162). If the result of *string-expr* matches *re*, the match result is bound to *var*s and *form*s are evaluated, and `rxmatch-case` returns the result of the last *form*.

: If *re* doesn't match the result of *string-expr*, *string-expr* yields non-string value, the interpretation proceeds to the next clause.

(test *proc* *form* ...)
: A procedure *proc* is applied on the result of *string-expr*. If it yields true value, *form*s are evaluated, and `rxmatch-case` returns the result of the last *form*.

: If *proc* yields `#f`, the interpretation proceeds to the next clause.

(test *proc* => *proc2*)
: A procedure *proc* is applied on the result of *string-expr*. If it yields true value, *proc2* is applied on the result, and its result is returned as the result of `rxmatch-case`.

: If *proc* yields `#f`, the interpretation proceeds to the next clause.

(else *form* ...)
: This form must appear at the end of *clause*s, if any. If other clauses fail, *form*s are evaluated, and the result of the last *form* becomes the result of `rxmatch-case`.

`(else => proc)`
: This form must appear at the end of *clauses*, if any. If other clauses fail, *proc* is evaluated, which should yield a procedure taking one argument. The value of *string-expr* is passed to *proc*, and its return values become the return values of `rxmatch-case`. `rx`

If no `else` clause exists, and all other clause fail, an undefined value is returned.

The `parse-date` example above becomes simpler if you use `rxmatch-case`

```
(define (parse-date2 str)
  (rxmatch-case str
    (test (lambda (s) (not (string? s))) #f)
    (#/^(\d\d?)\/(\d\d?)\/(\d\d\d\d)$/ (#f mm dd yyyy)
     (map string->number (list yyyy mm dd)))
    (#/^(\d\d\d\d)\/(\d\d?)\/(\d\d?)$/ (#f yyyy mm dd)
     (map string->number (list yyyy mm dd)))
    (#/^\d+\/\d+\/\d+$/               (#f)
     (errorf "ambiguous: ~s" str))
    (else (errorf "bogus: ~s" str))))
```

6.13.3 Inspecting and assembling regular expressions

When Gauche reads a string representation of regexp, first it parses the string and construct an abstract syntax tree (AST), performs some optimizations on it, then compiles it into an instruction sequence to be executed by the regexp engine.

The following procedures expose this process to user programs. It may be easier for programs to manipulate an AST than a string representation.

`regexp-parse` *string :key case-fold multi-line* [Function]

Parses a string specification of regexp in *string* and returns its AST, represented in S-expression. See below for the spec of AST.

When a true value is given to the keyword argument *case-fold*, returned AST will match case-insensitively. (Case insensitive regexp is handled in parser level, not by the engine).

`regexp-parse-sre` *sre* [Function]

Parses *sre* as a Scheme Regular Expression as described in SRFI-115 and returns its AST. Only a subset of SRE is supported at the moment. The following syntax is not supported: `bog`, `eog` and `grapheme`. see Section 10.3.16 [R7RS regular expressions], page 552.

`regexp-optimize` *ast* [Function]

Performs some rudimental optimization on the regexp AST, returning regexp AST.

Currently it only optimizes some trivial cases. The plan is to make it cleverer in future.

`regexp-compile` *ast :key multi-line* [Function]

Takes a regexp *ast* and returns a regexp object. Currently the outermost form of *ast* must be the zero-th capturing group. (That is, *ast* should have the form `(0 #f x ...)`.) The outer grouping is always added by `regexp-parse` to capture the entire regexp.

Note: The function does some basic check to see the given AST is valid, but it may not reject invalid ASTs. In such case, the returned regexp object doesn't work properly. It is caller's responsibility to provide a properly constructed AST. (Even if it rejects an AST, error messages are often incomprehensible. So, don't use this procedure as a AST validness checker.)

regexp-compile-sre *sre :key multi-line* [Function]

Takes a scheme regexp *sre* and returns a regexp object. The zero-th group is always captured.

If a false value is given to the keyword argument *multi-line*, which is the default, `bol` and `eol` behave like `bos` and `eos` (i.e. only match at the beginning or end of string).

regexp-ast *regexp* [Function]

Returns AST used for the regexp object *regexp*.

regexp-unparse *ast :key (on-error :error)* [Function]

From the regexp's *ast*, reconstruct the string representation of the regexp. The keyword argument *on-error* can be a keyword `:error` (default) or `#f`. If it's the former, an error is signaled when *ast* isn't valid regexp AST. If it's the latter, `regexp-unparse` just returns `#f`.

This is the structure of AST. Note that this is originally developed only for internal use, and not very convenient to manipulate from the code (e.g. if you insert or delete a subtree, you have to renumber capturing groups to make them consistent.) There's a plan to provide a better representation, such as SRE, and a tool to convert it to this AST back and forth. Contributions are welcome.

```
<ast> : <clause>   ; special clause
      | <item>     ; matches <item>

<item> : <char>       ; matches char
       | <char-set>   ; matches char set
       | (comp . <char-set>) ; matches complement of char set
       | any          ; matches any char
       | bos | eos    ; beginning/end of string assertion
       | bol | eol    ; beginning/end of line assertion
       | bow | eow | wb | nwb ; word-boundary/negative word boundary assertion

<clause> : (seq <ast> ...)       ; sequence
       | (seq-uncase <ast> ...)  ; sequence (case insensitive match)
       | (seq-case <ast> ...)    ; sequence (case sensitive match)
       | (alt <ast> ...)         ; alternative
       | (rep <m> <n> <ast> ...) ; repetition at least <m> up to <n> (greedy)
                                 ; <n> may be '#f'
       | (rep-min <m> <n> <ast> ...)
                                 ; repetition at least <m> up to <n> (lazy)
                                 ; <n> may be '#f'
       | (rep-while <m> <n> <ast> ...)
                                 ; like rep, but no backtrack
       | (<integer> <symbol> <ast> ...)
                                 ; capturing group.  <symbol> may be #f.
       | (cpat <condition> (<ast> ...) (<ast> ...))
                                 ; conditional expression
       | (backref . <integer>) ; backreference by group number
       | (backref . <symbol>)  ; backreference by name
       | (once <ast> ...)      ; standalone pattern.  no backtrack
       | (assert . <asst>)     ; positive lookahead assertion
       | (nassert . <asst>)    ; negative lookahead assertion

<condition> : <integer>     ; (?(1)yes|no) style conditional expression
       | (assert . <asst>)  ; (?(?=condition)...) or (?(?<=condition)...)
       | (nassert . <asst>) ; (?(?!condition)...) or (?(?<!condition)...)

<asst> : <ast> ...
       | ((lookbehind <ast> ...))
```

6.14 Vectors

<vector> [Builtin Class]

A vector is a simple 1-dimensional array of Scheme objects. You can access its element by index in constant time. Once created, a vector can't be resized.

Class <vector> inherits <sequence> and you can use various generic functions such as map and fold on it. See Section 9.5 [Collection framework], page 344, and See Section 9.29 [Sequence framework], page 441.

If you keep only a homogeneous numeric type, you may be able to use SRFI-4 homogeneous vectors (see Section 11.2 [Homogeneous vectors], page 590).

R7RS defines bytevectors; in Gauche, they're just u8vectors in gauche.uvector module (r7rs modules defines aliases. see Section 10.2.2 [R7RS base library], page 504).

See Section 11.30 [Vector library], page 632, for additional operations on vectors.

vector? *obj* [Function]

[R7RS base] Returns #t if *obj* is a vector, #f otherwise.

make-vector *k :optional fill* [Function]

[R7RS base] Creates and returns a vector with length *k*. If optional argument *fill* is given, each element of the vector is initialized by it. Otherwise, the initial value of each element is undefined.

vector *obj ...* [Function]

[R7RS base] Creates a vector whose elements are *obj ....*

vector-tabulate *len proc* [Function]

Creates a vector of length *len*, initializing *i*-th element of which by (*proc i*) for all *i* between 0 and *len*

```
(vector-tabulate 5 (^x (* x x)))
  ⇒ #(0 1 4 9 16)
```

vector-length *vector* [Function]

[R7RS base] Returns the length of a vector *vector*.

With gauche.collection module, you can also use a method size-of.

vector-ref *vector k :optional fallback* [Function]

[R7RS+] Returns *k*-th element of vector *vector*.

By default, vector-ref signals an error if *k* is negative, or greater than or equal to the length of *vector*. However, if an optional argument *fallback* is given, it is returned for such case. This is an extension of Gauche.

With gauche.sequence module, you can also use a method ref.

vector-set! *vector k obj* [Function]

[R7RS base] Sets *k*-th element of the vector *vector* to *obj*. It is an error if *k* is negative or greater than or equal to the length of *vector*.

With gauche.sequence module, you can also use a setter method of ref.

vector->list *vector :optional start end* [Function]
list->vector *list :optional start end* [Function]

[R7RS+] Converts a vector to a list, or vice versa.

The optional *start* and *end* arguments limit the range of the source. (R7RS don't define *start* and *end* arguments for list->vector.)

```
(vector->list '#(1 2 3 4 5))      ⇒ (1 2 3 4 5)
```

```
(list->vector '(1 2 3 4 5))       ⇒ #(1 2 3 4 5)
(vector->list '#(1 2 3 4 5) 2 4) ⇒ (3 4)
(list->vector (circular-list 'a 'b 'c) 1 6)
  ⇒ #(b c a b c)
```

With *gauche.collection* module, you can use `(coerce-to <list> vector)` and `(coerce-to <vector> list)` as well.

`reverse-list->vector` *list :optional start end* [Function]

[R7RS vector] Without optional arguments, it returns the same thing as `(list->vector (reverse list))`, but does not allocate the intermediate list. The optional *start* and *end* argument limits the range of the input list.

```
(reverse-list->vector '(a b c d e f g) 1 5)
  ⇒ #(e d c b)
```

`vector->string` *vector :optional start end* [Function]
`string->vector` *string :optional start end* [Function]

[R7RS base] Converts a vector of characters to a string, or vice versa. It is an error to pass a vector that contains other than characters to `vector->string`.

The optional *start* and *end* arguments limit the range of the source.

```
(vector->string '#(#\a #\b #\c #\d #\e))     ⇒ "abcde"
(string->vector "abcde")                     ⇒ #(#\a #\b #\c #\d #\e)
(vector->string '#(#\a #\b #\c #\d #\e) 2 4) ⇒ ("cd")
```

With *gauche.collection* module, you can use `(coerce-to <string> vector)` and `(coerce-to <vector> string)` as well.

`vector-fill!` *vector fill :optional start end* [Function]

[R7RS base] Sets all elements in a vector *vector* to *fill*.

Optional *start* and *end* limits the range of effect between *start*-th index (inclusive) to *end*-th index (exclusive). *Start* defaults to zero, and *end* defaults to the length of *vector*.

`vector-copy` *vector :optional start end fill* [Function]

[R7RS base] Copies a vector *vector*. Optional *start* and *end* arguments can be used to limit the range of *vector* to be copied. If the range specified by *start* and *end* falls outside of the original *vector*, the *fill* value is used to fill the result vector.

```
(vector-copy '#(1 2 3 4 5))      ⇒ #(1 2 3 4 5)
(vector-copy '#(1 2 3 4 5) 2 4) ⇒ #(3 4)
(vector-copy '#(1 2 3 4 5) 3 7 #f) ⇒ #(4 5 #f #f)
```

`vector-copy!` *target tstart source :optional sstart send* [Function]

[R7RS base] Copies the content of *source* vector into the *target* vector starting from *tstart* in the target. The *target* vector must be mutable. Optional *sstart* and *send* limits the range of source vector.

```
(rlet1 v (vector 'a 'b 'c 'd 'e)
  (vector-copy! v 2 '#(1 2)))
  ⇒ #(a b 1 2 e)
(rlet1 v (vector 'a 'b 'c 'd 'e)
  (vector-copy! v 2 '#(1 2 3 4) 1 3))
  ⇒ #(a b 2 3 e)
```

An error is raised if the portion to be copied is greater than the room in the target (that is, between *tstart* to the end).

It is ok to pass the same vector to *target* and *source*; it always works even if the regions of source and destination are overlapping.

`vector-append` *vec* ... [Function]

[R7RS base] Returns a newly allocated vector whose contents are concatenation of elements of *vec* in order.

```
(vector-append '#(1 2 3) '#(a b)) ⇒ #(1 2 3 a b)
(vector-append) ⇒ #()
```

`vector-map` *proc vec1 vec2* ... [Function]

[R7RS base] Returns a new vector, *i*-th of which is calculated by applying *proc* on the list of each *i*-th element of *vec1 vec2* .... The length of the result vector is the same as the shortest vector of the arguments.

```
(vector-map + '#(1 2 3) '#(4 5 6 7))
 ⇒ #(5 7 9)
```

The actual order *proc* is called is undefined, and may change in the future versions, so *proc* shouldn't use side effects affected by the order.

Note: If you use `gauche.collection`, you can get the same function by `(map-to <vector> proc vec1 vec2 ...)`.

`vector-map-with-index` *proc vec1 vec2* ... [Function]

Like `vector-map`, but *proc* receives the current index as the first argument.

```
(vector-map-with-index list '#(a b c d e) '#(A B C))
 ⇒ #((0 a A) (1 b B) (2 c C))
```

This is what SRFI-43 calls `vector-map`. See Section 11.13 [Vector library (Legacy)], page 615.

Note: If you use `gauche.collection`, you can get the same function by `(map-to-with-index <vector> proc vec1 vec2 ...)`.

`vector-map!` *proc vec1 vec2* ... [Function]

[R7RS vector] For each index *i*, calls *proc* with *i*-th index of *vec1 vec2* ..., and set the result back to *vec1*. The value is calculated up to the minimum length of input vectors.

```
(rlet1 v (vector 1 2 3)
  (vector-map! ($ + 1 $) v))
  ⇒ #(2 3 4)

(rlet1 v (vector 1 2 3 4)
  (vector-map! + v '#(10 20)))
  ⇒ #(11 22 3 4)
```

`vector-map-with-index!` *proc vec1 vec2* ... [Function]

Like `vector-map!`, but *proc* receives the current index as the first argument. This is equivalent to SRFI-43's `vector-map!` (see Section 11.13 [Vector library (Legacy)], page 615).

```
(rlet1 v (vector 'a 'b 'c)
  (vector-map-with-index! list v))
  ⇒ #((0 a) (1 b) (2 c))
```

`vector-for-each` *proc vec1 vec2* ... [Function]

[R7RS base] For all *i* below the minimum length of input vectors, calls *proc* with *i*-th elements of *vec1 vec2* ..., in increasing order of *i*.

```
(vector-for-each print '#(a b c))
 ⇒ prints a, b and c.
```

`vector-for-each-with-index` *proc vec1 vec2* ... [Function]

Like `vector-for-each`, but *proc* receives the current index in the first argument.

This is equivalent to SRFI-43's `vector-for-each`. See Section 11.13 [Vector library (Legacy)], page 615.

6.15 Hashtables

R7RS-large defines hashtable (`scheme.hash-table` module, see Section 10.3.7 [R7RS hash tables], page 536) but its API is not completely consistent with Gauche's original hashtables and other native APIs.

Rather than mixing different flavor of APIs, we keep Gauche's native API consistent, and provide R7RS procedures that are inconsistent with aliases—specifically, those procedures are suffixed with `-r7` in `gauche` module. For portable programs, you can import `scheme.hash-table` to get R7RS names.

`<hash-table>` [Builtin Class]

Hash table class. Inherits `<collection>` and `<dictionary>`.

Gauche doesn't provide immutable hash tables for now. (If you need immutable maps, see Section 12.11 [Immutable map], page 654).

Hash table properties

`hash-table?` *obj* [Function]

[R7RS hash-table] Returns `#t` iff *obj* is a hash table.

`hash-table-mutable?` *ht* [Function]

[R7RS hash-table] Returns `#t` iff a hash table *ht* is mutable. Gauche doesn't have immutable hash tables, so this procedure always returns `#t` for any hash tables.

`hash-table-comparator` *ht* [Function]

Returns a comparator used in the hashtable *ht*.

`hash-table-type` *ht* [Function]

This is an old API, superseded by `hash-table-comparator`.

Returns one of symbols `eq?`, `eqv?`, `equal?`, `string=?`, `general`, indicating the type of the hash table *ht*.

`hash-table-num-entries` *ht* [Function]
`hash-table-size` *ht* [Function]

[R7RS hash-table] Return the number of entries in the hash table *ht*. R7RS name is `hash-table-size`.

Hash table constructors and converters

`make-hash-table` *:optional comparator* [Function]

[R7RS+ hash-table] Creates a hash table. The *comparator* argument specifies key equality and hash function using a comparator (see Section 6.2.4 [Basic comparators], page 103). If omitted, `eq-comparator` is used. Note that in R7RS, *comparator* argument can't be omitted.

As Gauche's extension, the *comparator* argument can also be one of the symbols `eq?`, `eqv?`, `equal?` or `string=?`. If it is one of those symbols, `eq-comparator`, `eqv-comparator`, `equal-comparator` and `string-comparator` will be used, respectively.

The comparator must have hash function, of course. See Section 6.2.3 [Hashing], page 100, for the built-in hash functions. In general, comparators derived from other comparators having hash functions also have appropriate hash functions.

`hash-table-from-pairs` *comparator key***&value** . . . [Function]

Constructs and returns a hash table from given list of arguments. The *comparator* argument is the same as of `make-hash-table`. Each *key&value* must be a pair, and its car is used as a key and its cdr is used as a value.

Note: This is called `hash-table` by 0.9.5. R7RS introduced a procedure with the same name, but different interface. We see R7RS version makes more sense, so we'll eventually switch to it, but the transition will take long time. The R7RS interface is available as `hash-table-r7`, and we urge you to use it in the new code, and replace existing `hash-table` with `hash-table-from-pairs`.

```
(hash-table-from-pairs 'eq? '(a . 1) '(b . 2))
  ≡
  (rlet1 h (make-hash-table 'eq?)
     (hash-table-put! h 'a 1)
     (hash-table-put! h 'b 2))
```

`hash-table` *comparator key*&**value** . . . [Function]

An alias of `hash-table-from-pairs` above. R7RS introduced the same name procedure with different interface (see `hash-table-r7` below), and we'd like to switch to it in future. For now, use either `hash-table-from-pairs` or `hash-table-r7`, or import `scheme.hash-table` and write in R7RS.

`hash-table-r7` *comparator args* . . . [Function]

Create and returns a hash table using *comparator*. The *args* . . . are the contents, alternating keys and values.

This is defined as `hash-table` in R7RS `scheme.hash-table` (see Section 10.3.7 [R7RS hash tables], page 536).

```
(hash-table-r7 'eq? 'a 1 'b 2)
  ≡
  (rlet1 h (make-hash-table 'eq?)
     (hash-table-put! h 'a 1)
     (hash-table-put! h 'b 2))
```

Note: An R7RS compliant implementation of `hash-table` may return an immutable hash table. Since Gauche doesn't have immutable hash tables (we have immutable maps instead; see Section 12.11 [Immutable map], page 654), we return mutable hash tables. However, the portable program should refrain from mutating the returned hash tables.

`hash-table-unfold` *p f g seed comparator :rest args* [Function]

[R7RS hash-table] Constructs and returns a new hash table with those repetitive steps. Each iteration keeps the current seed value, whose initial value is *seed*.

1. Apply a stop predicate *p* to the current seed value. If it returns a true value, stop.
2. Apply a value producer *f* to the current seed value. It must return two values, which are used as a key and the corresponding value, of the hash table.
3. Apply a next procedure *g* to the current seed value. The value it returns becomes the next seed value.

`hash-table-copy` *ht :optional mutable?* [Function]

[R7RS hash-table] Returns a new copy of a hash table *ht*.

R7RS defines this procedure to return an immutable hash table if the implementation supports one, unless the optional *mutable?* argument is provided and not false. Gauche doesn't have immutable hash tables so it ignores the optional argument and always returns a mutable hash table. But when you write a portable programs, keep it in mind.

`hash-table-empty-copy` *ht* [Function]

[R7RS hash-table] Returns a new mutable empty hash table that has the same properties as the given hash table *ht*.

alist->hash-table *alist :optional comparator* [Function]

[R7RS+ hash-table] Creates and returns a hash table that has entries of each element in alist, using its car as the key and its cdr as the value. The *comparator* argument is the same as in `make-hash-table`. The default value of *comparator* is `eq-comparator`.

R7RS doesn't allow to omit *comparator*.

hash-table->alist *hash-table* [Function]

[R7RS hash-table]

```
(hash-table-map h cons)
```

Hash table lookup and mutation

hash-table-get *ht key :optional default* [Function]

Search *key* from a hash table *ht*, and returns its value if found. If the key is not found in the table and *default* is given, it is returned. Otherwise an error is signaled.

hash-table-put! *ht key value* [Function]

Puts a key *key* with a value *value* to the hash table *ht*.

ref (*ht* <*hash-table*>) *key :optional default* [Method]

(setter ref) (*ht* <*hash-table*>) *key value* [Method]

Method versions of `hash-table-get` and `hash-table-put!`.

hash-table-ref *ht key :optional failure success* [Function]

[R7RS hash-table] This is R7RS way to look up a hash table.

Look up a value associated to the *key* in the table *ht*, then pass it to a procedure *success*, and returns its value. If *success* is omitted, an identity function is used.

If there's no association for *key* in *ht*, a thunk *failure* is called and its result is returned. The default value of *failure* throws an error.

It is more general than Gauche's `hash-table-get`, but if you need to simply return a fallback value in case of failure, you need to wrap it with a clojure, which is annoying. In R7RS, you can use `hash-table-ref/default` below.

hash-table-ref/default *ht key default* [Function]

[R7RS hash-table] Looks up *key* in a hash table *ht* and returns the associated value. If there's no *key* in the table, returns *default*.

This is same as Gauche's `hash-table-get`, except that *default* is not optional. We provide both, for `hash-table-get` is short and handy.

hash-table-set! *ht args ...* [Function]

[R7RS hash-table] This is R7RS version to put associations into a hash table. The *args ...* is a list of alternating keys and values; so, unlike Gauche's `hash-table-put!`, you can insert more than one associations at once. It is an error if *args ...* have odd number of arguments.

```
(hash-table-set! ht 'a 1 'b 2)
  ≡
  (begin (hash-table-put! ht 'a 1)
         (hash-table-put! ht 'b 2))
```

hash-table-intern!-r7 *ht key failure* [Function]

This is defined in R7RS as `hash-table-intern!`. We add `-r7` suffix to remind that it takes a failure thunk, which is consistent with R7RS hash-table interface but not Gauche's way.

Lookup *key* in *ht*. If there's already an entry, it just returns the value. Otherwise, it calls a thunk *failure*, and insert the association of *key* and the return value of *failure* into *ht*, and returns the value.

`hash-table-exists?` *ht key* [Function]
`hash-table-contains?` *ht key* [Function]

[R7RS hash-table] Returns `#t` if a hash table *ht* has a key *key*.

R7RS name is `hash-table-contains?`.

`hash-table-delete!` *ht key* [Function]

Deletes an entry that has a key *key* from the hash table *ht*. Returns `#t` if the entry has exist, or `#f` if the entry hasn't exist. The same function is called `hash-table-remove!` in STk (except that it returns an undefined value); I use 'delete' for consistency to SRFI-1, SRFI-13 and other parts of the libraries.

Note: This is different from R7RS `hash-table-delete!`, so we provide R7RS interface with an alias `hash-table-delete!-r7`.

`hash-table-delete!-r7` *ht key . . .* [Function]

Delets entries that have *key . . .* from the hash table *ht*. The *key* which isn't in *ht* has no effect. Returns the number of entries actually deleted.

This is called `hash-table-delete!` in R7RS, and so as in `scheme.hash-table`. We provide this under different name, for Gauche's `hash-table-delete!` returns a boolean value.

`hash-table-clear!` *ht* [Function]

[R7RS hash-table] Removes all entries in the hash table *ht*.

`hash-table-push!` *ht key value* [Function]

Conses *value* to the existing value for the key *key* in the hash table *ht* and makes it the new value for *key*. If there's no entry for *key*, an entry is created with the value `(list value)`.

Works the same as the following code, except that this function only looks up the *key* once, thus it's more efficient.

```
(hash-table-put! ht key
    (cons value (hash-table-get ht key '())))
```

`hash-table-pop!` *ht key :optional default* [Function]

Looks for the value for the key *key* in the hash table *ht*. If found and it is a pair, replaces the value for its cdr and returns car of the original value. If no entry for *key* is in the table, or the value is not a pair, the table is not modified and the procedure returns *default* if given, or signals an error otherwise.

During the operation the key is looked for only once, thus runs efficiently.

Note: R7RS has `hash-table-pop!` but its totally different. We provide R7RS version as an alias `hash-table-pop!-r7`

`hash-table-pop!-r7` *ht* [Function]

Removes one arbitrary entry from *ht*, and returns the removed entry's key and value as two values. If *ht* is empty, an error is thrown.

This is called `hash-table-pop!` in R7RS, and so as in `scheme.hash-table`.

`hash-table-update!` *ht key proc :optional default* [Function]

A more general version of `hash-table-push!` etc. It works basically as the following code piece, except that the lookup of *key* is only done once.

```
(let ((tmp (proc (hash-table-get ht key default))))
  (hash-table-put! ht key tmp)
  tmp)
```

For example, when you use a hash table to count the occurrences of items, the following line is suffice to increment the counter of the item, regardless of whether *item* has already appeared or not.

```
(hash-table-update! ht item (cut + 1 <>) 0))
```

R7RS provides `hash-table-update!` with different interface, so we provide R7RS version as an alias `hash-table-update!-r7`.

`hash-table-update!-r7` *ht key updater :optional failure success* [Function]

This is R7RS version of `hash-table-update!`. With no optional arguments, it works like Gauche's `hash-table-update!`. But in practice you often needs to specify the behavior when *key* hasn't been in *ht*, in which case R7RS differs from Gauche.

The R7RS version works like this but potentially more efficiently:

```
(hash-table-put! ht key (updater (hash-table-ref-r7 ht key failure success)))
```

`hash-table-update!/default` *ht key updater default* [Function]

[R7RS hash-table] This is the same as Gauche's `hash-table-default!`, except that the default value can't be omitted.

Hash table scanners

`hash-table-for-each` *ht proc* [Function]
`hash-table-map` *ht proc* [Function]

A procedure *proc* is called with two arguments, a key and its associated value, over all the entries in the hash table *ht*.

`hash-table-fold` *ht kons knil* [Function]

For all entries in the hash table *ht*, a procedure *kons* is called with three arguments; a key, its associated value, and the previous return value of *kons*. The first call of *kons* receives *knil* as the third argument. The return value of the last call of *kons* is returned from `hash-table-fold`.

`hash-table-find` *ht pred :optional failure* [Function]

Apply *pred* with each key and value in the hash table *ht*. Once *pred* returns a true value, that return value is immediately returned from `hash-table-find`. If no key-value satisfies *pred*, a thunk *failure* is invoked and its result is returned. If *failure* is omitted, `(lambda () #f)` is assumed.

Note: The convention starting from srfi-1 is that `*-find` returns an item in the collection that satisfy the predicate, while `*-any` returns a non-false value the predicate returns. SRFI-125 broke the convention. The justification given in SRFI-125 discussion was that the "any" semantics is strictly upper-compatible to the "find" semantics so we can combine two. So far, though, SRFI-125 is the only exception of this convention.

```
;; Find if hash tables ha and hb has a common key.
(hash-table-find ha (^[k v] (hash-table-exists? hb k)))
```

`hash-table-keys` *ht* [Function]
`hash-table-values` *ht* [Function]

Returns all the keys or values of hash table *ht* in a list, respectively.

Hash table as sets

`hash-table-compare-as-sets` *ht1 ht2 :optional value=? fallback* [Function]

A hash table can be viewed as a set of pairs of key and value. This procedure compares two hash tables *ht1* and *ht2* as such sets.

The key comparators of two tables must match (in terms of `equal?` of the comparators). Otherwise, an error is signaled.

Two elements of the set are equal to each other iff their keys match with the equality predicate of the key comparator, and their values match with *value=?* procedure. If omitted, `equal?` is used for *value=?*

There can be four cases.

- If *ht1* is a pure subset of *ht2*, returns -1 (*ht1* is smaller than *ht2*).
- If *ht2* is a pure subset of *ht1*, returns 1 (*ht1* is greater than *ht2*).
- If *ht1* and *ht2* contains exactly the same elements, returns 0 (*ht1* equals to *ht2*).
- Neither *ht1* nor *ht2* is a subset of another. In this case, *fallback* is returned if it is given, or an error is thrown.

`hash-table=?` *value-cmpr ht1 ht2* [Function]

[R7RS hash-table] This also compares two hash tables *ht1* and *ht2* as sets, and returns true iff two are the same. That is, every element in *ht1* is also in *ht2* and vice versa.

Two element are the same iff their keys are the same in terms of the equality predicate of the tables' key comparator, and their values are the same in terms of the equality predicate of a comparator *value-cmpr*.

It is an error if *ht1* and *ht2* has different key comparators. See also `hash-table-compare-as-sets` above.

`hash-table-union!` *ht1 ht2* [Function]
`hash-table-intersection!` *ht1 ht2* [Function]
`hash-table-difference!` *ht1 ht2* [Function]
`hash-table-xor!` *ht1 ht2* [Function]

[R7RS hash-table] Perform set operations on two hashtables *ht1* and *ht2*, and modify *ht1* to store the result. Note that these procedures only look at the keys for operation; if the values of the same key differ between *ht1* and *ht2*, the value in *ht1* is taken.

- The union operation picks each entry that is in at least one of *ht1* or *ht2*.
- The intersection operation picks each entry that is both in *ht1* and *ht2*.
- The difference operation picks each entry that is in *ht1* but not in *ht2*.
- The xor operation picks each entry that is in only one of *ht1* or *ht2*, but not in both.

6.16 Treemaps

`<tree-map>` [Builtin Class]

Tree map class. Tree maps are a data structure that maps key objects to value objects. It's like hash tables except tree maps uses balanced tree internally. Insertion and lookup is O(log n).

Unlike hashtables, a tree map keeps the order of the keys, so it is easy to traverse entries in the order of keys, to find minimum/maximum keys, or to find a key closest to the given value.

The `<tree-map>` class inherits `<sequence>` and `<ordered-dictionary>`.

`make-tree-map` *:optional comparator* [Function]
`make-tree-map` *key=? key<?* [Function]

Creates and returns an instance of `<tree-map>`. The keys are compared by *comparator*, whose default is `default-comparator`. The comparator must have a comparison procedure, for we need a total order in the keys. See Section 6.2.4 [Basic comparators], page 103, for the details.

For the backward compatibility, `make-tree-map` also accepts a procedure as a *comparator*; the procedure must take two keys and returns either `-1`, `0`, or `1`, depending on whether the first key is less than, equal to, or greater than the second key, respectively. In other words, it is a comparison procedure of a comparator.

The second form of `make-tree-map` is also for the backward compatibility; it takes two procedures, each must be a procedure that takes two keys; the first one returns `#t` iff two keys are equal, and the second one returns `#t` iff the first key is strictly smaller than the second.

`tree-map-comparator` *tree-map* [Function]

Returns the comparator used in the tree map.

`tree-map-copy` *tree-map* [Function]

Copies and returns *tree-map*. Modification on the returned tree doesn't affect the original tree.

`tree-map-empty?` *tree-map* [Function]

Returns `#t` if *tree-map* doesn't have any elements, or `#f` otherwise.

`tree-map-num-entries` *tree-map* [Function]

Returns the number of elements in *tree-map*.

`tree-map-exists?` *tree-map key* [Function]

Returns `#t` if *tree-map* has an entry with *key*, or `#f` otherwise.

`tree-map-get` *tree-map key :optional fallback* [Function]

Looks for *key* in *tree-map*. If the entry is found, returns a value corresponding to the key. Otherwise, returns *fallback* if it is provided, or signals an error.

`tree-map-put!` *tree-map key value* [Function]

Inserts an entry with a *key* and corresponding *value* into *tree-map*. If there already exists an entry with a key which is equivalent (under *key=?*), the entry is modified to have *value*.

`tree-map-delete!` *tree-map key* [Function]

Deletes an entry with *key* from *tree-map* if such an entry exists, and returns `#t`. If *tree-map* doesn't have such an entry, `#f` is returned.

`tree-map-clear!` *tree-map* [Function]

Removes all entries in *tree-map*.

`tree-map-update!` *tree-map key proc :optional fallback* [Function]

A generalized version of `tree-map-push!` etc. It works like the following code, except that searching for the key is done only once.

```
(let ((tmp (proc (tree-map-get tree-map key fallback))))
  (tree-map-put! tree-map key tmp)
  tmp)
```

`tree-map-push!` *tree-map key value* [Function]

Looks for an entry with *key* in *tree-map*. If it exists, the procedure conses *value* to the original value and makes it as a new value. Otherwise, the procedure creates a new entry for the *key* and makes `(list value)` its value.

`tree-map-pop!` *tree-map key :optional fallback* [Function]

Looks for an entry with *key* in *tree-map*. If it exists and its value is a pair, then the procedure updates its value with `cdr` of the original value, and returns `car` of the original entry. If such an entry does not exist, or has a non-pair value, the procedure doesn't modify *tree-map* and returns *fallback* if it is given, otherwise reports an error.

`tree-map-min` *tree-map* [Function]
`tree-map-max` *tree-map* [Function]

Returns a pair of a key and its value with the minimum or maximum key, respectively. If *tree-map* is empty, `#f` is returned.

`tree-map-pop-min!` *tree-map* [Function]
`tree-map-pop-max!` *tree-map* [Function]

Looks for an entry with minimum or maximum key, respectively, then deletes the entry from *tree-map* and returns a pair of the key and its value of the original entry. If *tree-map* is empty, `#f` is returned.

`tree-map-fold` *tree-map proc seed* [Function]
`tree-map-fold-right` *tree-map proc seed* [Function]

Iterate over elements in *tree-map*, applying *proc* which has a type `(key, value, seed) -> seed`. The difference of `tree-map-fold` and `tree-map-fold-right` is the associative order of applying *proc*, just like the difference between `fold` and `fold-right`.

```
tree-map-fold:
  (proc Kn Vn (proc Kn-1 Vn-1 ... (proc K0 V0 seed)))

tree-map-fold-right
  (proc K0 V0 (proc K1 V1 ... (proc Kn Vn seed)))
```

Some examples:

```
(define tree (alist->tree-map '((3 . a) (7 . b) (5 . c)) = <))

(tree-map-fold tree list* '())
   ⇒ (7 b 5 c 3 a)
(tree-map-fold-right tree list* '())
   ⇒ (3 a 5 c 7 b)
```

`tree-map-map` *tree-map proc* [Function]

Calls *proc*, which must take two arguments, with each key/value pair in *tree-map*, and collect the results into a list and returns it. The order of results corresponds to the order of keys—that is, the first element of the result list is what *proc* returns with minimum key and its value, and the last element of the result list is what *proc* returns with the maximum key and its value. (Note: Like `map`, the order that *proc* is actually called is unspecified; *proc* is better to be side-effect free.)

`tree-map-for-each` *tree-map proc* [Function]

Calls *proc*, which must take two arguments, with each key/value pair in *tree-map*, in the increasing order of the keys. *proc* is called purely for side effects; the returned values are discarded.

`tree-map-floor` *tree-map probe :optional fallback-key fallback-value* [Function]
`tree-map-ceiling` *tree-map probe :optional fallback-key fallback-value* [Function]
`tree-map-predecessor` *tree-map probe :optional fallback-key fallback-value* [Function]
`tree-map-successor` *tree-map probe :optional fallback-key fallback-value* [Function]

These procedures search the entry which has the closest key to the given *probe*. If such an entry is found, returns two values, its key and its value. Otherwise, returns two values, *fallback-key* and *fallback-value*, both defaulted to `#f`.

The criteria of "closest" differ slightly among these procedures; `tree-map-floor` finds the maximum key which is no greater than *probe*; `tree-map-ceiling` finds the minimum key which is no less than *probe*; `tree-map-predecessor` finds the maximum key which is strictly less than *probe*; and `tree-map-successor` finds the minimum key which is strictly greater than *probe*.

`tree-map-floor-key` *tree-map probe* `optional` *fallback-key* [Function]
`tree-map-ceiling-key` *tree-map probe* `optional` *fallback-key* [Function]
`tree-map-predecessor-key` *tree-map probe* `optional` *fallback-key* [Function]
`tree-map-successor-key` *tree-map probe* `optional` *fallback-key* [Function]

Like `tree-map-floor` etc., but only returns the key of the found entry (or *fallback-key* if there's no entry which satisfies the criteria).

`tree-map-floor-value` *tree-map probe* `optional` *fallback-value* [Function]
`tree-map-ceiling-value` *tree-map probe* `optional` *fallback-value* [Function]
`tree-map-predecessor-value` *tree-map probe* `optional` *fallback-value* [Function]
`tree-map-successor-value` *tree-map probe* `optional` *fallback-value* [Function]

Like `tree-map-floor` etc., but only returns the value of the found entry (or *fallback-value* if there's no entry which satisfies the criteria).

`tree-map-keys` *tree-map* [Function]
`tree-map-values` *tree-map* [Function]

Returns a list of all keys and all values, respectively. The keys and values are in ascending order of the keys.

`tree-map->alist` *tree-map* [Function]

Returns a list of pairs of keys and values for all entries. The pairs are in ascending order of the keys.

`alist->tree-map` *alist :optional comparator* [Function]
`alist->tree-map` *alist key=? key<?* [Function]

Creates a new tree map with the *comparator* or *key=?/key<?* procedures, then populates it with *alist*, each pair in which are interpreted as a cons of a key and its value. The meaning of *comparator*, *key=?* and *key<?* are the same as `make-tree-map`.

The following two procedures compares two tree maps with slightly different views.

`tree-map-compare-as-sets` *tree-map1 tree-map2 :optional value=? fallback* [Function]

Compares two tree maps as sets of entries. If we look at tree maps as sets of entries, we can define a partial order between two maps; they are equal to each other if they have exactly the same entries, and tree-map A is smaller than tree-map B if A us a strict subset of B.

If *tree-map1* and *tree-map2* are the same, 0 is returned. If *tree-map1* is smaller than *tree-map2*, -1 is returned. If *tree-map1* is greater than *tree-map2*, 1 is returned.

If one argument isn't subset of the other, we can't determine the order. In such a case, if *fallback* is given, it is returned. Otherwise, an error is signalled.

The comparators of *tree-map1* and *tree-map2* must be the same (`equal?`), otherwise an error is signalled. See Section 6.2.4 [Basic comparators], page 103, about the comparators.

An entry is equal to another entry if their keys match in terms of the comparator of the tree-map, and also their values match with the provided *value=?* predicate, which is defaulted to `equal?`.

```
(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (2 . b) (3 . c)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator))
 ⇒ 0
```

```
(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (3 . c)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator))
 ⇒ -1

(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (3 . c) (4 . d) (2 . b)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator))
 ⇒ 1

(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (3 . c) (4 . d)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator))
 ⇒ ERROR: tree-maps can't be ordered

(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (3 . c) (4 . d)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator)
 eq?
 #f)
 ⇒ #f
```

`tree-map-compare-as-sequences` *tree-map1 tree-map2 :optional value-cmp* [Function]

Compares two tree maps as sequence of entries, ordered by keys. If both maps have entries with the same key, we use a comparator *value-cmp* to break the tie (naturally, *value-cmp* must have ordering predicate.) If *value-cmp* is omitted, `default-comparator` is used.

The comparators of *tree-map1* and *tree-map2* must be the same (`equal?`), otherwise an error is signalled. See Section 6.2.4 [Basic comparators], page 103, about the comparators.

If *tree-map1* and *tree-map2* are the same, 0 is returned. If *tree-map1* is smaller than *tree-map2*, -1 is returned. If *tree-map1* is greater than *tree-map2*, 1 is returned.

Unlike `tree-map-compare-as-sets`, this procedure defines total order of tree maps which share the same comparator.

```
(tree-map-compare-as-sequences
 (alist->tree-map '((1 . a) (3 . c)) default-comparator)
 (alist->tree-map '((3 . c) (2 . b)) default-comparator))
 ⇒ -1

(tree-map-compare-as-sequences
 (alist->tree-map '((2 . b) (3 . d)) default-comparator)
 (alist->tree-map '((3 . c) (2 . b)) default-comparator))
 ⇒ 1
```

6.17 Weak pointers

A weak pointer is a reference to an object that doesn't prevent the object from being garbage-collected. Gauche provides weak pointers as a *weak vector* object. A weak vector is like a vector of objects, except each object can be garbage collected if it is not referenced from objects other than weak vectors. If the object is collected, the entry of the weak vector is replaced for `#f`.

```
gosh> (define v (make-weak-vector 1))
```

```
v
gosh> (weak-vector-ref v 0)
#f
gosh> (weak-vector-set! v 0 (cons 1 1))
#<undef>
gosh> (weak-vector-ref v 0)
(1 . 1)
gosh> (gc)
#<undef>
gosh> (gc)
#<undef>
gosh> (weak-vector-ref v 0)
#f
```

`<weak-vector>` [Builtin Class]

The weak vector class. Inherits `<sequence>` and `<collection>`, so you can use `gauche.collection` (see Section 9.5 [Collection framework], page 344) and `gauche.sequence` (see Section 9.29 [Sequence framework], page 441).

```
(coerce-to <weak-vector> '(1 2 3 4))
  ⇒ a weak vector with four elements
```

`make-weak-vector` *size* [Function]

Creates and returns a weak vector of size *size*.

`weak-vector-length` *wvec* [Function]

Returns the length of a weak vector *wvec*.

`weak-vector-ref` *wvec k :optional fallback* [Function]

Returns *k*-th element of a weak vector *wvec*.

By default, `weak-vector-ref` signals an error if *k* is negative, or greater than or equal to the size of *wvec*. However, if an optional argument *fallback* is given, it is returned for such case.

If the element has been garbage collected, this procedure returns *fallback* if it is provided, `#f` otherwise.

With `gauche.sequence` module, you can also use a method `ref`.

`weak-vector-set!` *wvec k obj* [Function]

Sets *k*-th element of the weak vector *wvec* to *obj*. It is an error if *k* is negative or greater than or equal to the size of *wec*.

6.18 Procedures and continuations

In Scheme, *procedures* are fundamental blocks to build a program (see Section 4.3 [Making procedures], page 42). A procedure represents a certain computation, possibly parameterized, and can be *applied* to the actual arguments to execute the computation. Scheme also provides the means to extract the continuation of the current computation and wraps it in a procedure (see Section 6.18.7 [Continuations], page 195).

Gauche extends the concept of procedure application, allowing you to apply any object as if it's a procedure; for example, you can set up Gauche to accept `("abc" 2)` can be a valid application syntax. See Section 6.18.6 [Applicable objects], page 195, for the details.

6.18.1 Procedure class and applicability

`<procedure>` [Builtin Class]

Represents a procedure. Ordinary Scheme procedures created by `lambda` is an instance of this class, as well as built-in primitive procedures written in C. Note that, in Gauche, other type of objects can behave as a procedure; so checking whether an object is a procedure or not doesn't mean much unless you want to mess around with Gauche internals.

`procedure?` *obj* [Function]

[R7RS base] Returns `#t` if *obj* is *inherently* applicable objects, `#f` otherwise. By *inherently* applicable we mean Gauche unconditionally understands that *obj* can be called as a procedure; an instance of `<procedure>` is so, as well as generic functions (`<generic>`) and methods (`<method>`). See Section 7.4 [Generic function and method], page 299, for the details.

Since you can make any type of objects applicable at any time (see Section 6.18.6 [Applicable objects], page 195), the fact that `procedure?` returned `#f` doesn't mean that the object cannot be applied. To check if an object can be applied or not, use `applicable?` below.

`apply` *proc arg1 . . . args* [Function]

[R7RS base] Calls a procedure *proc* with a list of arguments, `(`*arg1* `. . . .` *args*`)`. The last argument *args* must be a proper list. Returns (a) value(s) *proc* returns.

```
(apply list 'a 'b '(c d e)) ⇒ (a b c d e)

(apply + 1 2 '(3 4 5))      ⇒ 15
```

`applicable?` *obj class . . .* [Function]

Checks if *obj* can be called with the types of arguments listed in *class* . . . . That is, when `(applicable? foo <string> <integer>)` returns `#t`, then you can call `foo` as `(foo "x" -2)`, for example. (It doesn't mean you won't get an error; `foo` may be accept only nonnegative integers, which you cannot tell from the result of `applicable?`. But if `applicable?` returns `#t`, Gauche won't complain "foo is not applicable" when you call `foo`.

This procedure takes applicable objects into account. So, for example, `(applicable? #/a/ <string>)` returns `#t`, for the regular expressions are applicable to a string (see Section 6.13 [Regular expressions], page 162).

For generic functions, `applicable?` returns `#t` if it has at least one method such that each of its specifiers is a superclass of the corresponding *class* argument given to `applicable?`.

```
(define-method foo ((x <sequence>) (y <integer>)) #f)

(applicable? foo <sequence> <integer>) ⇒ #t
(applicable? foo <string> <integer>) ⇒ #t
(applicable? foo <hash-table> <integer>) ⇒ #f
(applicable? foo <string> <real>) ⇒ #f
```

The second example returns `#t` since `<string>` is a subclass of `<sequence>`, while the third example returns `#f` since `<hash-table>` isn't a subclass of `<sequence>`. The fourth example returns `#f` since `<real>` isn't a subclass of `<integer>`.

Traditional Scheme procedures (such as ones created by `lambda`) only cares the number of arguments but not their types; it accepts any type as far as the number of arguments matches. To check such a condition, pass `<top>` as the argument class. (`<top>` is a superclass of all classes.)

```
(applicable? cons <top> <top>) ⇒ #t
```

If you want to check an object is applicable to a certain number of *some* class of arguments, you can pass `<bottom>` as the argument class instead. (`<bottom>` is a subclass of all classes.)

```
(define-method foo ((x <sequence>) (y <integer>)) #f)
```

```
(applicable? foo <top> <top>) ⇒ #f
(applicable? foo <bottom> <bottom>) ⇒ #t
```

See Section 6.1 [Types and classes], page 96, for the details of `<top>`, `<bottom>` and Gauche's type handling.

6.18.2 Universal accessor

~ *obj key keys ...* [Function]
`(setter ~)` *obj key keys ...* [Function]

The procedure `~` can be used to access a part of various aggregate types.

```
;; Access to an element of a sequence by index
(~ '(a b c) 0)          ⇒ a
(~ '#(a b c) 2)         ⇒ c
(~ "abc" 1)             ⇒ #\b
(~ '#u8(10 20 30) 1)  ⇒ 20

;; Access to an element of a collection by key
(~ (hash-table 'eq? '(a . 1) '(b . 2)) 'a)
  ⇒ 1

;; Access to a slot of an object by slot name
(~ (sys-localtime (sys-time)) 'hour)
  ⇒ 20
```

The access can be chained:

```
(~ '#((a b c) (d e f) (g h i)) 1 2) ⇒ f

(~ (hash-table 'eq? '(a . "abc") '(d . "def")) 'a 2)
  ⇒ #\c
```

You can think `~` as left-associative, that is,

```
(~ x k j) ≡ (~ (~ x k) j)
```

and so on.

The generalized setter `set!` can be used with `~` to replace the specified element.

```
(define z (vector 'a 'b 'c))
(set! (~ z 1) 'Z)

z ⇒ #(a Z c)

(define z (vector (list (vector 'a 'b 'c)
                        (vector 'd 'e 'f)
                        (vector 'g 'h 'i))
                  (list (vector 'a 'b 'c)
                        (vector 'd 'e 'f)
                        (vector 'g 'h 'i))))

z ⇒ #((#(a b c) #(d e f) #(g h i))
     (#(a b c) #(d e f) #(g h i)))

(set! (~ z 1 2 0) 'Z)
z ⇒  #((#(a b c) #(d e f) #(g h i))
     (#(a b c) #(d e f) #(Z h i)))
```

Internally, a call to `~` is implemented by a generic function `ref`. See Chapter 7 [Object system], page 279, for more about generic functions.

`ref` *object key :optional args . . .* [Generic function]
`(setter ref)` *object key value* [Generic function]

Many aggregate types defines a specialized method of these to provide uniform access and mutation. Meaning of optional arguments *args* of `ref` depends on each specialized method, but it is common that the first optional argument of `ref` is a *fallback* value, which is to be returned when *object* doesn't have a meaningful association with *key*.

The manual entry of each aggregate type shows the specialized method and its semantics in detail.

Conceptually, `~` can be understood as follows:

```
(define ~
  (getter-with-setter
   (case-lambda
     [(obj selector) (ref obj selector)]
     [(obj selector . more) (apply ~ (ref obj selector) more)])
   (case-lambda
     [(obj selector val) ((setter ref) obj selector val)]
     [(obj selector selector2 . rest)
      (apply (setter ~) (ref obj selector) selector2 rest)])))
```

(Gauche may use some short-cut for optimization, though, so this code may not reflect the actual implementation.)

6.18.3 Combinators

Gauche has some primitive procedures that allows combinatory programming.

`pa$` *proc arg . . .* [Function]

Partial application. Returns a procedure, and when it is called with arguments *m* . . ., it is equivalent to call `(proc arg ... m ...)`.

```
(define add3 (pa$ + 3))
(add3 4) ⇒ 7

(map (pa$ * 2) '(1 2 3)) ⇒ (2 4 6)
```

Macros `cut` and `cute` defined in SRFI-26 provide a similar abstraction, with a bit more flexible but less compact notation. See Section 4.3 [Making procedures], page 42.

`apply$` *proc* [Function]
`map$` *proc* [Function]
`for-each$` *proc* [Function]

Partial application versions of `apply`, `map` and `for-each`.

```
(define map2* (map$ (pa$ * 2)))
(map2* '(1 2 3)) ⇒ (2 4 6)
```

`count$` *pred* [Function]
`fold$` *kons :optional knil* [Function]
`fold-right$` *kons :optional knil* [Function]
`reduce$` *f :optional ridentity* [Function]
`reduce-right$` *f :optional ridentity* [Function]
`filter$` *pred* [Function]
`remove$` *pred* [Function]

`partition$` *pred* [Function]
`member$` *item* [Function]
`find$` *pred* [Function]
`find-tail$` *pred* [Function]
`any$` *pred* [Function]
`every$` *pred* [Function]
`delete$` *pred* [Function]
`assoc$` *item* [Function]

Partial application versions of some srfi-1 (R7RS `(scheme list)`) procedures (see Section 10.3.1 [R7RS lists], page 512).

`.$` *f . . .* [Function]
`compose` *f . . .* [Function]

Combine procedures. All arguments must be procedures. When two procedures are given, `(.$ f g)` is equivalent to the following code:

```
(lambda args (call-with-values (lambda () (apply g args)) f))
```

When more than two arguments are passed, they are composed as follows:

```
(.$ f g h ...) ≡ (.$ (.$ f g) h ...)
```

Some examples:

```
(define not-zero? (.$ not zero?))
(not-zero? 3) ⇒ #t
(not-zero? 0) ⇒ #f

(define dot-product (.$ (apply$ +) (map$ *)))
(dot-product '(1 2 3) '(4 5 6)) ⇒ 32
```

A couple of edge cases: if only one argument is given, the argument itself is returned. If no arguments are given, the procedure `values` is returned.

Note: The name `.$` comes from the fact that `.` is commonly used for function composition in literatures and some programming languages, and that Gauche uses suffix `$` to indicate combinators. However, since it is not a valid R7RS identifier, portable programs may want to use the alias `compose`, with which you can easily add a portable definition using `srfi-0`, for example.

`complement` *pred* [Function]

Returns a procedure that reverses the meaning of the predicate *pred*. That is, for the arguments for which *pred* returns true return false, and vice versa.

```
(map (complement even?) '(1 2 3)) ⇒ '(#t #f #t)
(map (complement =) '(1 2 3) '(1 1 3)) ⇒ '(#f #t #f)
((complement (lambda () #f))) ⇒ #t
```

`any-pred` *pred . . .* [Function]

Returns a procedure which applies given argument(s) to each predicate *pred*. If any *pred* returns a non-`#f` value, the value is returned. If all the *pred*s return `#f`, `#f` is returned.

```
(define string-or-symbol? (any-pred string? symbol?))
(string-or-symbol? "abc") ⇒ #t
(string-or-symbol? 'abc)  ⇒ #t
(string-or-symbol? 3)     ⇒ #f

(define <> (any-pred < >))
(<> 3 4) ⇒ #t
(<> 3 3) ⇒ #f
```

```
((any-pred (cut memq <> '(a b c))
           (cut memq <> '(1 2 3)))
 'b)  ⇒ '(b c)
```

`every-pred` *pred ...* [Function]

Returns a procedure which applies given argument(s) to each predicate *pred*. If every *pred* returns a non-`#f` value, the value returned by the last *pred* is returned. If any *pred* returns `#f`, `every-pred` returns `#f` without calling further *preds*.

```
((every-pred odd? positive?) 3)  ⇒ #t
((every-pred odd? positive?) 4)  ⇒ #f
((every-pred odd? positive?) -3) ⇒ #f

(define safe-length (every-pred list? length))
(safe-length '(a b c))  ⇒ 3
(safe-length "aaa")     ⇒ #f
```

6.18.4 Optional argument parsing

Gauche supports optional and keyword arguments in extended lambda syntax (see Section 4.3 [Making procedures], page 42). However, you can also use the following macros to parse optional and keyword arguments, without relying Gauche's extension.

```
(define (foo a b :optional (c #f) (d 'none))
  body ...)

;; is roughly equivalent to ...

(define (foo a b . args)
  (let-optionals* args ((c #f) (d 'none))
    body ...))
```

Explicitly parsing the extended arguments may be useful for portable programs, since it is rather straightforward to implement those macros rather than extend lambda syntax.

Those macros can also be useful to factor out common argument parsing routines.

`let-optionals*` *restargs* (*var-spec ...*) *body ...* [Macro]
`let-optionals*` *restargs* (*var-spec ... . restvar*) *body ...* [Macro]

Given a list of values *restargs*, binds variables according to *var-spec*, then evaluates *body*.

Var-spec can be either a symbol, or a list of two elements and its car is a symbol. The symbol is the bound variable name. The values in *restargs* are bound to the symbol in order. If there are not as many values in *restargs* as *var-spec*, the rest of *symbols* are bound to the default values, determined as follows: If *var-spec* is just a symbol, the default value is undefined. If *var-spec* is a list, the default value is the result of evaluation of the second element of the list. In the latter case the second element is only evaluated when there are not enough arguments. The binding proceeds in the order of *var-spec*, so the second element may refer to the bindings of previous *var-spec*.

In the second form, *restvar* must be a symbol and bound to the list of values whatever left from *restargs* after binding to *var-spec*.

It is not an error if *restarg* has more values than *var-specs*. The extra values are simply ignored in the first form.

```
(define (proc x . args)
  (let-optionals* args ((a 'a)
                        (b 'b)
```

```
                          (c 'c))
    (list x a b c)))

(proc 0)          ⇒ (0 a b c)
(proc 0 1)        ⇒ (0 1 b c)
(proc 0 1 2)      ⇒ (0 1 2 c)
(proc 0 1 2 3)    ⇒ (0 1 2 3)

(define (proc2 . args)
  (let-optionals* args ((a 'a) . b)
    (list a b)))

(proc2)           ⇒ (a ())
(proc2 0)         ⇒ (0 ())
(proc2 0 1)       ⇒ (0 (1))
(proc2 0 1 2)     ⇒ (0 (1 2))

(define (proc3 . args)
  (let-optionals* args ((a 0)
                        (b (+ a 1))
                        (c (+ b 1)))
    (list a b c)))

(proc3)           ⇒ (0 1 2)
(proc3 8)         ⇒ (8 9 10)
(proc3 8 2)       ⇒ (8 2 3)
(proc3 8 2 -1)    ⇒ (8 2 -1)
```

`get-optional` *restargs default* [Macro]

This is a short version of `let-optionals*` where you have only one optional argument. Given the optional argument list *restargs*, this macro returns the value of optional argument if one is given, or the result of *default* otherwise. *Default* is not evaluated unless *restargs* is an empty list.

```
(define (proc x . maybe-opt)
  (let ((option (get-optional maybe-opt #f)))
    (list x option)))

(proc 0)          ⇒ (0 #f)
(proc 0 1)        ⇒ (0 1)
```

`let-keywords` *restarg* (*var-spec* ...) *body* ... [Macro]
`let-keywords` *restarg* (*var-spec* ... . *restvar*) *body* ... [Macro]

This macro is for keyword arguments. *Var-spec* can be one of the following forms:

(*symbol expr*)
: If the *restarg* contains keyword which has the same name as *symbol*, binds *symbol* to the corresponding value. If such a keyword doesn't appear in *restarg*, binds *symbol* to the result of *expr*.

(*symbol keyword expr*)
: If the *restarg* contains keyword *keyword*, binds *symbol* to the corresponding value. If such a keyword doesn't appear in *restarg*, binds *symbol* to the result of *expr*.

The default value *expr* is only evaluated when the keyword is not given to the *restarg*.

If you use the first form, `let-keyword` throws an error when `restarg` contains a keyword argument that is not listed in *var-specs*. When you want to allow keyword arguments other than listed in *var-specs*, use the second form.

In the second form, *restvar* must be either a symbol or `#f`. If it is a symbol, it is bound to a list of keyword arguments that are not processed by *var-specs*. If it is `#f`, such keyword arguments are just ignored.

```
(define (proc x . options)
  (let-keywords options ((a 'a)
                         (b :beta 'b)
                         (c 'c)
                         . rest)
    (list x a b c rest)))

(proc 0)          ⇒ (0 a b c ())
(proc 0 :a 1)     ⇒ (0 1 b c ())
(proc 0 :beta 1) ⇒ (0 a 1 c ())
(proc 0 :beta 1 :c 3 :unknown 4) ⇒ (0 a 1 3 (:unknown 4))
```

`let-keywords*` *restarg* (*var-spec* . . .) *body* . . . [Macro]
`let-keywords*` *restarg* (*var-spec* . . . . *restvar*) *body* . . . [Macro]

Like `let-keywords`, but the binding is done in the order of *var-specs*. So each *expr* can refer to the variables bound by preceding *var-specs*.

6.18.5 Procedure arity

Interface to query procedure's arity. The API is taken from MzScheme (PLT Scheme).

`arity` *proc* [Function]

Given procedure *proc*, returns an integer, an *arity-at-least* object, or a list of integer(s) and *arity-at-least* objects.

An integer result indicates *proc* takes exactly that number of arguments. An *arity-at-least* indicates *proc* takes at least `(arity-at-least-value` *arity-at-least*`)` arguments. The list indicates there are multiple procedures with different arities.

Since one can add methods to an existing procedure or generic function at any moment in Gauche, the value returned by `arity` only indicates the current state of the procedure. It will change if new method is added to the procedure/generic-function.

```
(arity cons) ⇒ 2
(arity list) ⇒ #<arity-at-least 0>
(arity make) ⇒ (#<arity-at-least 1>)
```

`arity-at-least?` *obj* [Function]

Returns true if *obj* is an arity-at-least object.

`arity-at-least-value` *arity-at-least* [Function]

Returns the number of required arguments the arity-at-least object indicates.

`procedure-arity-includes?` *proc k* [Function]

If a procedure *proc* can take *k* arguments, returns `#t`. Otherwise returns `#f`.

6.18.6 Applicable objects

Gauche has a special hook to make an arbitrary object *applicable*.

`object-apply` *object arg* ... [Generic Function]

If an object that is neither a procedure nor a generic function is applied to some arguments, the object and the arguments are passed to a generic function `object-apply`.

This can be explained better by examples.

For example, suppose you try to evaluate the following expression:

```
("abcde" 2)
```

The operator evaluates to a string, which is neither a procedure nor a generic function. So Gauche interprets the expression as if it were like this:

```
(object-apply "abcde" 2)
```

Gauche doesn't define a method of `object-apply` that takes `<string>` and `<integer>` by default, so this signals an error. However, if you define such a method:

```
(define-method object-apply ((s <string>) (i <integer>))
  (string-ref s i))
```

Then the first expression works as if a string is *applied* on the integer:

```
("abcde" 2) ⇒ #\c
```

This mechanism works on almost all occasions where a procedure is allowed.

```
(apply "abcde" '(1))    ⇒ (#\b)
(map "abcde" '(3 2 1)) ⇒ (#\d #\c #\b)
```

Among Gauche built-in objects, `<regexp>` object and `<regmatch>` object have `object-apply` defined. See Section 6.13 [Regular expressions], page 162.

`(setter object-apply)` *object arg* ... *value* [Generic Function]

If a form of applying an applicable object appears in the first position of `set!` form, this method is called, that is:

```
(set! (object arg ...) value)
 ⇒ ((setter object-apply) object arg ... value)
```

6.18.7 Continuations

`call-with-current-continuation` *proc* [Function]
`call/cc` *proc* [Function]

[R7RS base] Encapsulates the current continuation to a procedure ("continuation procedure"), and calls *proc* with it. When *proc* returns, its value becomes `call/cc`'s value. When the continuation procedure is invoked with zero or more arguments somewhere, the further calculation is abandoned and `call/cc` returns with the arguments given to the continuation procedure.

First class continuation is one of the most distinct feature of Scheme, but this margin is too small to contain explanation. Please consult to the appropriate documents.

There's a nontrivial interaction between C language runtime and Scheme continuation. Suppose the following scenario:

1. An application's C runtime calls back a Scheme routine. For example, GUI framework calls back a draw routine written in Scheme.
2. A continuation is captured in the Scheme routine.
3. The Scheme routine returns to the C runtime.
4. The continuation captured in 2 is invoked.

It is no problem to invoke the continuation, but if the control is about to return to the Scheme routine to the C runtime (that is, to execute step 3 again), an error is signaled as follows.

```
*** ERROR: attempt to return from a ghost continuation.
```

This is because C routines don't expect the calling function to return more than once. The C stack frame on which the Scheme callback was originally called is likely to be deallocated or modified at the time the continuation is invoked.

If you think of a continuation as a chain of control frames, growing from root towards upward, you can imagine that, once a control returns to the C world, the chain is cut at the boundary. You can still execute such rootless continuations, but you have to move the control away from it before it tries to return to its root that no longer exists. You can call another continuation, or raise an exception, for example.

Using partial continuations (or delimited continuations) is another way to avoid such complications. See Section 9.24 [Partial continuations], page 417.

`let/cc` *var body* ... [Macro]

This macro expands to : `(call/cc (lambda (var) body ...))`. The API is taken from PLT Scheme.

`dynamic-wind` *before body after* [Function]

[R7RS base] This is a primitive to manage *dynamic environment*. Dynamic environment is a set of states which are kept during execution of a certain expression. For example, the current output ports are switched *during* execution of `with-output-to-port`. They can be nested dynamically, as opposed to the lexical environment, in which nesting is determined statically from the program source.

Before, *body* and *after* are all procedures with no arguments. In normal situation, `dynamic-wind` calls *before*, then *body*, then *after*, then returns whatever value(s) *body* returned.

The intention is that the *before* thunk sets up the dynamic environment for execution of *body*, and the *after* thunk restores it to the previous state.

If a control flow goes out from *body* by invoking a continuation captured outside of the dynamic scope of `dynamic-wind` (for example, an error is signaled in *body*), *after* is called.

If a control flow goes into *body* by invoking a continuation captured inside *body* from outside of the dynamic scope of `dynamic-wind`, *before* is called.

```
(letrec ((paths '())
         (c #f)
         (add (lambda (s) (push! paths s))))
  (dynamic-wind
   (lambda () (add 'connect))
   (lambda ()
     (add (call/cc (lambda (c0) (set! c c0) 'talk1))))
   (lambda () (add 'disconnect)))
  (if (< (length paths) 4)
      (c 'talk2)
      (reverse paths)))
 ⇒ (connect talk1 disconnect connect talk2 disconnect)
```

Note: Since *after* is guaranteed to be called when an error causes *body* to abort, it may appear tempting to use `dynamic-wind` to use resource clean-up, just like `try-catch` construct in Java. It's *not* for that. Since the control may return to *body*, the situation `dynamic-wind` handles should be considered more like a context switch.

For resource clean-up, you can use exception handling mechanism such as `guard` and `unwind-protect` (see Section 6.20.3 [Handling exceptions], page 208), which is built on top of `dynamic-wind`.

As a rule of thumb, *after* should do things that can be reverted by *before*, such as manipulating error handler stack (instead of actually handling errors).

6.18.8 Multiple values

`values` *obj* . . . [Function]

[R7RS base] Returns *obj* . . . as multiple values. Caller can capture multiple values by a built-in syntax `receive` or `let-values` (Section 4.6 [Binding constructs], page 52), or the R7RS procedure `call-with-values` described below.

```
(values 1 2) ⇒ 1 and 2
```

`call-with-values` *producer consumer* [Function]

[R7RS base] Call a procedure *producer* with no argument. Then applies a procedure *consumer* on the value(s) *producer* returned. Returns the value(s) *consumer* returns.

```
(call-with-values (lambda () (values 1 2)) cons)
  ⇒ (1 . 2)
```

`values-ref` *mv-expr k* [Macro]

Returns *k*-th value of what *mv-expr* returns. Conceptually, it is the same as the following code.

```
(call-with-values (lambda () mv-expr) (lambda r (list-ref r k)))
```

This macro uses shortcuts for the typical cases like *k* is zero.

Similar to Common Lisp's `nth-value`, but the argument order is flipped to match other Scheme's `*-ref` procedures.

`values->list` *mv-expr* [Macro]

Evaluates *mv-expr*, puts all the results into a list and returns it. It is called `multiple-value-list` in Common Lisp.

```
(values->list (div-and-mod 10 3)) ⇒ (3 1)

(values->list 1) ⇒ (1)
```

6.18.9 Folding generated values

Sometimes a procedure is used as a *generator* of a series of values, by yielding one value at a time. Customary an EOF object is used to mark the end of the series. For example, `read-char` is such a procedure that yields a series of characters, terminated by EOF.

Since it is such a handy abstraction, Gauche provides a set of utilities (see Section 9.11 [Generators], page 372) to construct and generators out of various sources, including other generators.

The generated values needs to be consumed eventually. Here we provide several procedures to do that. These are useful when combined with input procedures like `read`, so we have them built-in instead of putting them in a separate module.

`generator-fold` *proc seed gen gen2* . . . [Function]

[R7RS generator] Works like `fold` on the generated values by generator procedures *gen gen2* . . . (See Section 6.6.5 [Walking over lists], page 131, for the details of `fold`).

When one generator is given, for each value *v* generated by *gen*, *proc* is called as `(proc v r)`, where *r* is the current accumulated result; the initial value of the accumulated result is *seed*, and the return value from *proc* becomes the next accumulated result. When *gen* returns EOF, the accumulated result at that time is returned from `generator-fold`.

When more than one generator is given, *proc* is called as `(proc v1 v2 ... r)`, where *v1*, *v2* ... are the values yielded from *gen*, *gen2*, ..., respectively, and *r* is the current accumulated result. The iteration terminates when any one of the generators returns EOF.

```
(with-input-from-string "a b c d e"
  (cut generator-fold cons 'z read))
  ⇒ (e d c b a . z)
```

`generator-fold-right` *proc seed gen gen2 ...* [Function]

Works like `fold-right` on the generated values by generator procedures *gen gen2* ... (see Section 6.6.5 [Walking over lists], page 131, for the details of `fold-right`).

This is provided for completeness, but it isn't a good way to handle generators; in order to combine values right-associatively, we should read all the values from the generators (until any one of the generator returns EOF), then start calling *proc* as

```
(proc v0_0 v1_0 ... (proc v0_1 v1_1 ... (proc v0_n v1_n ... seed) ...))
```

where *vn_m* is the *m*-th value yielded by *n*-th generator.

```
(with-input-from-string "a b c d e"
  (cut generator-fold-right cons 'z read))
  ⇒ (a b c d e . z)
```

As you see, keeping all intermediate values kind of defeats the benefit of generators.

`generator-for-each` *proc gen gen2 ...* [Function]

[R7RS generator] A generator version of `for-each`. Repeatedly applies *proc* on the values yielded by *gen*, *gen2* ... until any one of the generators yields EOF. The values returned from *proc* are discarded.

This is a handy procedure to consume generated values with side effects.

`generator-map` *proc gen gen2 ...* [Function]

A generator version of `map`. Repeatedly applies *proc* on the values yielded by *gen*, *gen2* ... until any one of the generators yields EOF. The values returned from *proc* are collected into a list and returned.

```
(with-input-from-string "a b c d e"
  (cut generator-map symbol->string read))
  ⇒ ("a" "b" "c" "d" "e")
```

The same effects can be achieved by combining `generator->list` and `gmap` (see Section 9.11.2 [Generator operations], page 376). This procedure is provided for the backward compatibility.

```
(generator->list (gmap proc gen gen2 ...))
```

`generator-find` *pred gen* [Function]

[R7RS generator] Returns the first item from the generator *gen* that satisfies the predicate *pred*.

The following example returns the first line matching the regexp `#/XYZ/` from the file `foo.txt`.

```
(with-input-from-file "foo.txt"
  (cut generator-find #/XYZ/ read-line))
```

Note: If you want to pick all the lines matching the regexp, like the `grep` command, you can use `gfilter` and `generator->list`.

6.19 Lazy evaluation

Gauche has two primitive lazy evaluation mechanisms.

The first one is an explicit mechanism, defined in the Scheme standard: You mark an expression to be evaluated lazily by `delay`, and you use `force` to make the evaluation happen when needed. Gauche also support another primitive `lazy`, as defined in srfi-45, for space-efficient tail-recursive lazy algorithms.

The second one is a lazy sequence, in which evaluation happens implicitly. From a Scheme program, a lazy sequence just looks as a list—you can take its `car` and `cdr`, and you can apply `map` or other list procedures on it. However, internally, its element isn't calculated until it is required.

6.19.1 Delay, force and lazy

Scheme has traditionally provided an explicit delayed evaluation mechanism using `delay` and `force`. After R5RS, however, it is found that it didn't mix well with tail-recursive algorithms: It required unbound memory, despite that the body of the algorithm could be expressed in iterative manner. Srfi-45 showed that introducing another primitive syntax `lazy` addresses the issue. For the detailed explanation please look at the srfi-45 document. Here we explain how to use those primitives.

`delay` *expression* [Special Form]
`lazy` *expression* [Special Form]

[R7RS lazy][SRFI-45] These forms creates a *promise* that delays the evaluation of *expression*. *Expression* will be evaluated when the promise is passed to `force`.

If *expression* itself is expected to yield a promise, you should use `lazy`. Otherwise, you should use `delay`. If you can think in types, the difference may be clearer.

```
lazy  : Promise a -> Promise a
delay : a -> Promise a
```

Since we don't have static typing, we can't enforce this usage. The programmer has to choose appropriate one from the context. Generally, `lazy` appears only to surround the entire body of function that express a lazy algorithm.

NB: In R7RS, `lazy` is called `delay-force`, for the operation is conceptually similar to `(delay (force expr))` (note that the type of `force` is `Promise a -> a`).

For the real-world example of use of `lazy`, you may want to check the implementation of `util.stream` (see Section 12.71 [Stream library], page 809).

`eager` *obj* [Function]

[SRFI-45] Returns a promise that returns the value of *obj*. Since that `eager` is a procedure, *obj* is evaluated before *eager* is called; so it works as a type converter (<code>a -> Promise a</code>) without delaying the evaluation. Used mainly to construct promise-returning functions.

`force` *promise* [Function]

[R7RS lazy] If *promise* is not a promise, it is just returned.

Otherwise, if *promise*'s value hasn't been computed, `force` makes *promise*'s encapsulated expression be evaluated, and returns the result.

Once *promise*'s value is computed, it is memorized in it so that subsequent `force` on it won't cause the computation.

`promise?` *obj* [Function]

[R7RS lazy] Returns `#t` iff *obj* is a promise object.

6.19.2 Lazy sequences

Introduction

A lazy sequence is a list-like structure whose elements are calculated lazily. Internally we have a special type of pairs, whose `cdr` is evaluated on demand. However, in Scheme level, you'll never see a distinct "lazy-pair" type. As soon as you try to access a lazy pair, Gauche automatically *force* the delayed calculation, and the lazy pair turns into an ordinary pair.

It means you can pass lazy sequences to ordinary list-processing procedures such as `car`, `cdr` or `map`.

Look at the following example; `generator->lseq` takes a procedure that generates one value at a time, and returns a lazy sequence that consists of those values.

```
(with-input-from-file "file"
  (^[] (let loop ([cs (generator->lseq read-char)] [i 0])
         (match cs
           [() #f]
           [(#\c (or #\a #\d) #\r . _) i]
           [(c . cs) (loop cs (+ i 1))]))))
```

It returns the position of the first occurrence of character sequence "car" or "cdr" in the file `file`. The loop treats the lazy sequence just like an ordinary list, but characters are read as needed, so once the sequence is found, the rest of the file won't be read. If we do it eagerly, we would have to read the entire file first no matter how big it is, or to give up using the mighty `match` macro and to write a basic state machine that reads one character one at a time.

Other than implicit forcing, Gauche's lazy sequences are slightly different than the typical lazy stream implementations in Scheme in the following ways:

1. When you construct a lazy sequence in an iterative lazy algorithm, only `cdr` side of the lazy pair is lazily evaluated; the `car` side is evaluated immediately. On the other hand, with `stream-cons` in `util.stream` (see Section 12.71 [Stream library], page 809), both `car` and `cdr` sides won't be evaluated until it is absolutely needed.
2. Gauche's lazy sequence always evaluates *one item ahead*. Once you get a lazy pair, its `car` part is already calculated, even if you don't use it. In most cases you don't need to care, for calculating one item more is a negligible overhead. However, when you create a self-referential lazy structure, in which the earlier elements of a sequence is used to calculate the latter elements of itself, a bit of caution is needed; a valid code for fully lazy circular structure may not terminate in Gauche's lazy sequences. We'll show a concrete example later. This bit of eagerness is also visible when side effects are involved; for example, lazy character sequence reading from a port may read one character ahead.

Note: R7RS `scheme.lseq` (srfi-127) provides a portable alternative of lazy sequence (see Section 10.3.10 [R7RS lazy sequences], page 544). It uses dedicated APIs (e.g. `lseq-cdr`) to operate on lazy sequences so that portable implementation is possible. In Gauche, we just use our built-in lazy sequence as srfi-127 lazy sequence; if you want your code to be portable, consider using srfi-127, but be careful not to mix lazy sequences and ordinary lists; Gauche won't complain, but other Scheme implementation may choke on it.

Primitives

`generator->lseq` *generator* [Function]
`generator->lseq` *item ... generator* [Function]

[R7RS lseq] Creates a lazy sequence that consists of items produced by *generator*, which is just a procedure with zero arguments that yields an item at a time. Returning EOF marks the end of the sequence (EOF itself isn't included in the sequence). For example, `read-char` can work as a generator. Gauche has a set of convenient utilities to deal with generators (see Section 9.11 [Generators], page 372).

In the second form, the returned lazy sequence is prepended by *item* .... Since there's no way to distinguish lazy pairs and ordinary pairs, you can write it as `(cons* item ... (generator->lseq generator))`, but that's more verbose.

Internally, Gauche's lazy sequence is optimized to be built on top of generators, so this procedure is the most efficient way to build lazy sequences.

Note: Srfi-127 also has `generator->lseq`, which is exactly the same as this in Gauche.

`lcons` *car cdr* [Macro]

Returns a lazy pair consists of *car* and *cdr*. The expression *car* is evaluated at the call of `lcons`, but evaluation of *cdr* is delayed.

You can't distinguish a lazy pair from an ordinary pair. If you access either its `car` or `cdr`, or even you ask `pair?` to it, its cdr part is implicitly forced and you get an ordinary pair.

Unlike `cons`, *cdr* should be an expression that yields a (lazy or ordinary) list, including an empty list. In other words, lazy sequences can always be a null-terminated list when entirely forced; there are no "improper lazy sequences". (Since Scheme isn't statically typed, we can't force the *cdr* expression to be a proper list before actually evaluating it. Currently if *cdr* expression yields non-list, we just ignore it and treat as if it yielded an empty list.)

```
(define z (lcons (begin (print 1) 'a) (begin (print 2) '())))
 ⇒ ; prints '1', since the car part is evaluated eagerly.

(cdr z) ⇒ () ;; and prints '2'

;; This also prints '2', for accessing car of a lazy pair forces
;; its cdr, even the cdr part isn't used.
(car (lcons 'a (begin (print 2) '()))) ⇒ a

;; So as this; asking pair? to a lazy pair causes forcing its cdr.
(pair? (lcons 'a (begin (print 2) '()))) ⇒ #t

;; To clarify: This doesn't print '2', because the second lazy
;; pair never be accessed, so its cdr isn't evaluated.
(pair? (lcons 'a (lcons 'b (begin (print 2) '())))) ⇒ #t
```

Now, let me show you a case where "one item ahead" evaluation becomes an issue. The following is an elegant definition of infinite Fibonacci sequence using self-referential lazy structure (`lmap` is a lazy map, defined in `gauche.lazy` module):

```
(use gauche.lazy)  ;; for lmap
(define *fibs* (lcons* 0 1 (lmap + *fibs* (cdr *fibs*)))) ;; BUGGY
```

Unfortunately, Gauche can't handle it well.

```
(car *fibs*)
 ⇒ 0
(cadr *fibs*)
 ⇒ *** ERROR: Attempt to recursively force a lazy pair.
```

When we want to access the second argument (`cadr`) of `*fibs*`, we take the car of the second pair, which is a lazy pair of `1` and `(lmap ...)`. The lazy pair is forced and its cdr part needs to be calculated. The first thing `lmap` returns needs to see the first and second element of `*fibs*`, but the second element of `*fibs*` is what we're calculating now!

We can workaround this issue by avoiding accessing the immediately preceding value. Fibonacci numbers F(n) = F(n-1) + F(n-2) = 2*F(n-2) + F(n-3), so we can write our sequence as follows.

```
(define *fibs*
  (lcons* 0 1 1 (lmap (^[a b] (+ a (* b 2))) *fibs* (cdr *fibs*))))
```

And this works!

```
(take *fibs* 20)
  ⇒ (0 1 1 2 3 5 8 13 21 34 55 89 144 233
     377 610 987 1597 2584 4181)
```

Many lazy algorithms are defined in terms of fully-lazy cons at the bottom. When you port such algorithms to Gauche using `lcons`, keep this bit of eagerness in mind.

Note also that `lcons` needs to create a thunk to delay the evaluation. So the algorithm to construct lazy list using `lcons` has an overhead of making closure for each item. For performance-critical part, you want to use `generator->lseq` whenever possible.

Utilities

`lcons*` *x . . . tail* [Macro]
`llist*` *x . . . tail* [Macro]

A lazy version of `cons*` (see Section 6.6.3 [List constructors], page 126). Both `lcons*` and `llist*` do the same thing; both names are provided for the symmetry to `cons*/list*`.

The *tail* argument should be an expression that yields a (possibly lazy) list. It is evaluated lazily. Note that the preceding elements *x . . .* are evaluated eagerly. The following equivalences hold.

```
(lcons* a)            ≡ a
(lcons* a b)          ≡ (lcons a b)
(lcons* a b ... y z)  ≡ (cons* a b ... (lcons y z))
```

`lrange` *start :optional end step* [Function]

Creates a lazy sequence of numbers starting from *start*, increasing by *step* (default 1), to the maximum value that doesn't exceed *end*. The default of *end* is `+inf.0`, so it creates an infinite list. (Don't type just `(lrange 0)` in REPL, or it won't terminate!)

If any of *start* or `step` is inexact, the resulting sequence has inexact numbers.

```
(take (lrange -1) 3) ⇒ (-1 0 1)

(lrange 0.0 5 0.5)
  ⇒ (0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5)

(lrange 1/4 1 1/8)
  ⇒ (1/4 3/8 1/2 5/8 3/4 7/8)
```

`liota` *:optional (count +inf.0) (start 0) (step 1)* [Function]

A lazy version of `iota` (see Section 6.6.3 [List constructors], page 126); returns a lazy sequence of *count* integers (default: positive infinity), starting from *start* (default: 0), stepping by *step* (default: 1).

Just like `iota`, the result consists of exact numbers if and only if both *start* and *step* are exact; otherwise the result consists of inexact numbers.

`port->char-lseq` *:optional port* [Function]
`port->byte-lseq` *:optional port* [Function]
`port->string-lseq` *:optional port* [Function]

`port->sexp-lseq` *:optional port* [Function]

These are the same as the following expressions, respectively. They are provided for the convenience, since this pattern appears frequently.

```
(generator->lseq (cut read-char port))
(generator->lseq (cut read-byte port))
(generator->lseq (cut read-line port))
(generator->lseq (cut read port))
```

If *port* is omitted, the current input port is used.

Note that the lazy sequence may buffer some items, so once you make an lseq from a port, only use the resulting lseq and don't ever read from *port* directly.

Note that the lazy sequence terminates when EOF is read from the port, but the port isn't closed. The port should be managed in larger dynamic extent where the lazy sequence is used.

You can also convert input data into various lists by the following expressions (see Section 6.22.7.4 [Input utility functions], page 231). Those procedures read the port eagerly until EOF and returns the whole data in a list, while `lseq` versions read the port lazily.

```
(port->list read-char port)
(port->list read-byte port)
(port->string-list port)
(port->sexp-list port)
```

Those procedures make (lazy) lists out of ports. The opposite can be done by `open-input-char-list` and `open-input-byte-list`; See Section 9.38 [Virtual ports], page 493, for the details.

See also Section 9.14 [Lazy sequence utilities], page 386, for more utility procedures that creates lazy sequences.

Examples

Let's consider calculating an infinite sequence of prime numbers. (Note: If you need prime numbers in your application, you don't need to write one; just use `math.prime`. see Section 12.27 [Prime numbers], page 705).

Just pretend we already have some prime numbers calculated in a variable `*primes*`, and you need to find a prime number equal to or grater than *n* (for simplicity, we assume *n* is an odd number).

```
(define (next-prime n)
  (let loop ([ps *primes*])
    (let1 p (car ps)
      (cond [(> (* p p) n) n]
            [(zero? (modulo n p)) (next-prime (+ n 2))]
            [else (loop (cdr ps))]))))
```

This procedure loops over the list of prime numbers, and if no prime number *p* less than or equal to `(sqrt n)` divides *n*, we can say *n* is prime. (Actual test is done by `(> (* p p) n)` instead of `(> p (sqrt n))`, for the former is faster.) If we find some *p* divides *n*, we try a new value `(+ n 2)` with `next-prime`.

Using `next-prime`, we can make a generator that keeps generating prime numbers. The following procedure returns a generator that returns primes above *last*.

```
(define (gen-primes-above last)
  (^[] (set! last (next-prime (+ last 2))) last))
```

Using `generator->lseq`, we can turn the generator returned by `gen-primes-above` into a lazy list, which can be used as the value of `*prime*`. The only caveat is that we need to have some pre-calculated prime numbers:

```
(define *primes* (generator->lseq 2 3 5 (gen-primes-above 5)))
```

Be careful not to evaluate `*primes*` directly on REPL, since it contains an infinite list and it'll blow up your REPL. You can look the first 20 prime numbers instead:

```
(take *primes* 20)
 ⇒ (2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71)
```

Or find what the 10000-th prime number is:

```
(~ *primes* 10000)
 ⇒ 104743
```

Or count how many prime numbers there are below 1000000:

```
(any (^[p i] (and (>= p 1000000) i)) *primes* (lrange 0))
 ⇒ 78498
```

Note: If you're familiar with the lazy functional approach, this example may look strange. Why do we use side-effecting generators while we can define a sequence of prime numbers in pure functional way, as follows?

```
(use gauche.lazy)

(define (prime? n)
  (not (any (^p (zero? (mod n p)))
            (ltake-while (^k (<= (* k k) n)) *primes*))))

(define (primes-from k)
  (if (prime? k)
    (lcons k (primes-from (+ k 2)))
    (primes-from (+ k 2))))

(define *primes* (llist* 2 3 5 (primes-from 7)))
```

(The module `gauche.lazy` provides `ltake-while`, which is a lazy version of `take-while`. We don't need lazy version of `any`, since it immediately stops when the predicate returns a true value.)

The use of `lcons` and co-recursion in `primes-from` is a typical idiom in functional programming. It's perfectly ok to do so in Gauche; except that the generator version is *much* faster (when you take first 5000 primes, generator version ran 17 times faster than co-recursion version on the author's machine).

It doesn't mean you should avoid co-recursive code; if an algorithm can be expressed nicely in co-recursion, it's perfectly ok. However, watch out the subtle semantic difference from lazy functional languages—straightforward porting may or may not work.

6.20 Exceptions

Gauche's exception system consists of three components; (1) the way to signal an exceptional case has occurred, (2) the way to specify how to handle such a case, and (3) the standard objects (*conditions*) to communicate the code that signals an exceptional case and the code that handles it.

Those three components are typically used together, so first we explain the typical usage patterns using examples. Then we describe each feature in detail.

Note for terminology: some languages use the word *exception* to refer to an object used to communicate the code that encountered an exceptional situation with a handler that deals with it. Gauche uses a term *condition* to refer to such objects, following SRFI-35. *Exception* is the situation, and *condition* is a runtime object that describes it.

6.20.1 Exception handling overview

Catching specific errors

One of the most typical exception handling is to catch a specific error raised by some built-in or library procedures. A macro `guard` can be used for such a purpose. The code looks like this:

```
(guard (exc [(condition-has-type? exc <read-error>)
             (format #t "read error!")
             'read-error]
            [else 'other-error])
  (read-from-string "(abc"))
```

The cadr of `guard` clause is a form of `(variable clause ...)`. In this example, the variable is `exc`, and it has two clauses. Each *clause* has the form like the one in `cond`.

The cddr of `guard` is the body, a list of expressions. This example has only one expression, `(read-from-string "(abc")`.

`guard` starts executing its body. `read-from-string` raises an error of type `<read-error>` when it encounters syntactic errors. The form `guard` intercepts the error, and binds the condition object to the variable *exc*, then checks the clauses following *exc* in a similar manner to `cond`—in this case, the thrown condition is of type `<read-error>`, so the test of the first clause is satisfied, and the rest of clause is executed, i.e. `"read error!"` is printed and a symbol `read-error` is returned.

If you're familiar with other languages, you may recognize the pattern. The cddr of `guard` form is like *try* clause of C++/Java or the cadr of `handler-case` of Common Lisp; and the cdadr of `guard` form is like `catch` clauses or the cddr of `handler-case`.

In the test expressions it is common to check the type of thrown condition. The function `condition-has-type?` is defined in SRFI-35 but it's rather lengthy. Gauche's condition classes can also work like a predicate, so you can write the above expression like this.

```
(guard (exc [(<read-error> exc)
             (format #t "read error!")
             'read-error]
            [else 'other-error])
  (read-from-string "(abc"))
```

Note: Generally you can't use `is-a?` to test if the thrown condition is of a specific type, since a condition may be *compound*. See Section 6.20.4 [Conditions], page 212, about compound conditions.

If no tests of *clauses* satisfy and no `else` clause is given, the exception 'falls off' the `guard` construct, i.e. it will be handled by the outer level of `guard` form or top-level. For example, the following `guard` form only handles `<read-error>` and `<system-error>`; if the body throws other type of conditions, it must be handled by outer level.

```
(guard (exc [(<read-error> exc) (handle-read-error)]
            [(<system-error> exc) (handle-system-error)])
  body ...)
```

See Section 6.20.3 [Handling exceptions], page 208, for more details on `guard` and other lower-level exception handling constructs.

Signaling exceptions from your code

The generic way to signal an exception is to use `raise` procedure.

```
(raise condition)
```

You can pass any object to *condition*; its interpretation solely depends on the exception handler. If you know the code raises an integer as a condition, you can catch it by `guard` as this:

```
(guard (exc [(integer? exc) 'raised])
  (raise 3))
```

However, as a convention, it is preferable to use an instance of `<condition>` or one of its subclasses. A macro `condition` can be used to create a condition object. The following examples show how to create a condition with some slot values and then raise it.

```
;; create and raise an error condition
(raise (condition
        (<error> (message "An error occurred."))))

;; create and raise a system error condition
(raise (condition
        (<system-error> (message "A system error occurred.")
                        (errno EINTR))))
```

See Section 6.20.4 [Conditions], page 212, for the details of `condition` macro and what kind of condition classes are provided.

The most common type of condition is an error condition, so a convenience procedure `error` and `errorf` are provided. They create an error condition with a message and raise it.

```
;; 'error' concatenates the arguments into a message.
(unless (integer? obj)
  (error "Integer expected, but got:" obj))

;; 'errorf' uses format to create a message.
(unless (equal? x y)
  (errorf "~s and ~s don't match" x y))
```

Unlike the exception throwing constructs in some languages, such as `throw` of C++/Java, which abandons its continuation, Scheme's `raise` may return to its caller. If you don't want `raise` to return, a rule of thumb is always to pass one of error conditions to it; then Gauche guarantees `raise` wont return. See the description of `raise` in Section 6.20.2 [Signaling exceptions], page 207, for more details.

Note: R7RS adopted slightly different semantics; it splits `raise` and `raise-continuable`, the former is for noncontinuable exception (if the exception handler returns, it raises another error), and the latter is for continuable exception. When you're in R7RS environment, R7RS-compatible `raise` will be used instead of this `raise`.

Defining your own condition

You can also define your own condition classes to pass application-specific information from the point of raising exception to the handlers.

To fit to Gauche's framework (SRFI-35), it is desirable that the new condition class inherits a built-in `<condition>` class or one of its descendants, and also is an instance of a metaclass `<condition-meta>`.

One way of ensuring the above convention as well as increasing portability is to use `define-condition-type` macro, defined in `SRFI-35`.

```
(define-condition-type <myapp-error> <error>
```

```
    myapp-error?
    (debug-info myapp-error-debug-info)
    (reason myapp-error-reason))
```

This defines a condition type (which is a class in Gauche) `<myapp-error>`, with a predicate `myapp-error?` and slots with accessors. Then you can use the new condition type like the following code:

```
(guard (exc
         [(myapp-error? exc)
          (let ([debug-info (myapp-error-debug-info exc)]
                [reason (myapp-error-reason exc)])
            ... handle myapp-error ...)])
  ...
  ...
  (if (something-went-wrong)
    (raise (condition
             (<myapp-error> (debug-info "during processing xxx")
                            (reason "something went wrong")))))
  ...
  ...
  )
```

If you don't mind to lose srfi compatibility, you can use Gauche's extended `error` and `errorf` procedures to write more concise code to raise a condition of subtype of `<error>`:

```
(if (something-went-wrong)
  (error <myapp-error>
         :debug-info "during processing xxx"
         :reason "something went wrong"))
```

See the description of `define-condition-type` macro for how the condition type is implemented in Gauche's object system.

6.20.2 Signaling exceptions

Signaling errors

The most common case of exceptions is an error. Two convenience functions to signal an error condition in simple cases are provided. To signal a compound condition, you can use `raise` as explained below.

`error` *string arg . . .* [Function]
`error` *condition-type keyword-arg . . . string arg . . .* [Function]

[R7RS+][SRFI-23+] Signals an error. The first form creates an `<error>` condition, with a message consists of *string* and *arg* . . ., and raises it. It is compatible to R7RS and SRFI-23's `error` behavior.

```
gosh> (define (check-integer x)
        (unless (integer? x)
           (error "Integer required, but got:" x)))
check-integer
gosh> (check-integer "a")
*** ERROR: Integer required, but got: "a"
Stack Trace:
_______________________________________
```

The second form can be used to raise an error other than the `<error>` condition. *condition-type* must be a condition type (see Section 6.20.4 [Conditions], page 212, for more explanation of condition types). It may be followed by keyword-value list to initialize the condition slots, and then optionally followed by a string and other objects that becomes an error message.

```
(define-condition-type <my-error> <error> #f
  (reason)
  (priority))

...
  (unless (memq operation *supported-operations*)
    (error <my-error>
           :reason 'not-supported :priority 'urgent
           "Operation not supported:" operation))
...
```

`errorf` *fmt-string arg . . .* [Function]
`errorf` *condition-type keyword-arg . . . fmt-string arg . . .* [Function]

Similar to `error`, but the error message is formatted by `format`, i.e. the first form is equivalent to:

```
(define (errorf fmt . args)
  (error (apply format #f fmt args)))
```

The second form can be used to raise an error other than an `<error>` condition. Meaning of *condition-type* and *keyword-arg*s are the same as `error`.

Signaling generic conditions

`raise` *condition* [Function]

[SRFI-18][R7RS base] This is the base mechanism of signaling exceptions.

The procedure invokes the current exception handler. The argument *condition* represents the nature of the exception, and passed to the exception handler. Gauche's built-in and library functions always use an instance of `<condition>` or one of its subclasses as *condition*, but you can pass any Scheme object to `raise`. The interpretation of *condition* is up to the exception handler.

Note: Unlike some of the mainstream languages in which "throwing" an exception never returns, you can set up an exception handler in the way that `raise` may return. The details are explained in Section 6.20.3 [Handling exceptions], page 208.

If you don't want `raise` to return, the best way is to pass a condition which is an instance of `<serious-condition>` or one of its subclasses. Gauche's internal mechanism guarantees raising such an exception won't return. See Section 6.20.4 [Conditions], page 212, for the hierarchy of built-in conditions.

R7RS adopted slightly different semantics regarding returning from `raise`; in R7RS, `raise` never returns—if the exception handler returns, another exception is raised. R7RS has `raise-continuable` to explicitly allow returning from the exception handler. For portable programs, always pass `<serious-condition>` or its subclasses to `raise`.

6.20.3 Handling exceptions

High-level exception handling mechanism

`guard` (*var clause . . .*) *body . . .* [Macro]

[R7RS base] This is *the* high-level form to handle errors in Gauche.

var is a symbol, and *clause*s are the same form as `cond`'s clauses, i.e. each clause can be either one of the following forms:

1. `(`*test expr* `...)`

2. `(test => proc)`

The last *clause* may be `(else expr ...)`.

This form evaluates *body* ... and returns the value(s) of the last *body* expression in normal case. If an exception is raised during the evaluation of body expressions, the raised exception is bound to a variable *var*, then evaluates *test* expression of each clause. If one of *test* expressions returns true value, then the corresponding *expr*s are evaluated if the clause is the first form above, or an *proc* is evaluated and the result of *test* is passed to the procedure *proc* if the clause is the second form.

When the *test*(s) and *expr*(s) in the clauses are evaluated, the exception handler that is in effect of the caller of `guard` are installed; that is, if an exception is raised again within *clause*s, it is handled by the *outer* exception handler or `guard` form.

If no *test* returns true value and the last `clause` is `else` clause, then the associated *expr*s are evaluated. If no *test* returns true value and there's no `else` clause, the raised exception is re-raised, to be handled by the outer exception handler.

When the exception is handled by one of `clause`s, `guard` returns the value(s) of the last *expr* in the handling clause.

The *clause*s are evaluated in the same dynamic environment as the `guard` form, i.e. any `dynamic-wind`s inside `body` are unwound before evaluation of the *clause*s. It is different from the lower level forms `with-error-handler` and `with-exception-handler`, whose handler is evaluated before the dynamic environment are unwound.

```
(let ([z '()])
  (guard (e [else (push! z 'caught)])
    (dynamic-wind (lambda () (push! z 'pre))
                  (lambda () (error "foo"))
                  (lambda () (push! z 'post))))
  (reverse z))
 ⇒ (pre post caught)

(guard (e [else (print 'OUTER) #f])
  (with-output-to-string
    (lambda ()
      (print 'INNER)
      (error "foo"))))
 ⇒ prints OUTER to the current output port of guard,
     not to the string port.
```

`unwind-protect` *expr cleanup* ... [Macro]

Executes *expr*, then executes *cleanup*s, and returns the result(s) of *expr*. If an uncontinuable exception is raised within *expr*, *cleanup*s are executed before the exception escapes from the `unwind-protect` form. For example, the following code calls `start-motor`, `drill-a-hole`, and `stop-motor` in order if everything goes ok, and if anything goes wrong in `start-motor` or `drill-a-hole`, `stop-motor` is still called before the exception escapes `unwind-protect`.

```
(unwind-protect
  (begin (start-motor)
         (drill-a-hole))
  (stop-motor))
```

The *cleanup* forms are evaluated in the same dynamic environment as `unwind-protect`. If an exception is thrown within *cleanup*, it will be handled outside of the `unwind-protect` form.

Although this form looks similar to `dynamic-wind`, they work at different layers and should not be confused. `dynamic-wind` is the bottom-level building block and used to manage current exception handlers, current i/o ports, parameters, etc. `dynamic-wind`'s *before* and *after* thunks are called whenever any of those control flow transition occurs. On the other hand, `unwind-protect` only cares about the Gauche's exception system. `unwind-protect`'s *cleanup* is called only when *expr* exits normally or throws Gauche's exception. In the above example, if control escapes from `drill-a-hole` by calling a continuation captured outside of `unwind-protect`, *cleanup* is not called; because the control may return to `drill-a-hole` again. It can happen if user-level thread system is implemented by `call/cc`, for example.

You can go back to the body *expr* from outside of `unwind-protect` by invoking continuations captured within *expr*.

However, keep in mind that once *cleanup* are executed, some resources might not be available in *expr*. We still allow it since the reexecuted part of *expr* may not depend on the resources cleaned up with *cleanup*.

Even if *expr* returns (normally or abnormally), *cleanup* only executed once, in the first time.

The name of this form is taken from Common Lisp. Some Scheme systems have similar macros in different names, such as `try-finally`.

`with-error-handler` *handler thunk* [Function]

Makes *handler* the active error handler and executes *thunk*. If *thunk* returns normally, the result(s) will be returned. If an error is signaled during execution of *thunk*, *handler* is called with one argument, an exception object representing the error, with the continuation of `with-error-handler`. That is, `with-error-handler` returns whatever value(s) *handler* returns.

If *handler* signals an error, it will be handled by the handler installed when `with-error-handler` called.

The dynamic environment where *handler* is executed is the same as the error occurs. If `dynamic-wind` is used in *thunk*, its *after* method is called after *handler* has returned, and before `with-error-handler` returns.

Note: Using this procedure directly is *no longer recommended*, since `guard` is more safe and portable. We'll keep this for a while for the backward compatibility, but we recommend to rewrite code to use `guard` instead of this. The common idiom of "cleanup on error" code:

```
(with-error-handler (lambda (e) (cleanup) (raise e))
  (lambda () body ...))
```

should be written like this:

```
(guard (e [else (cleanup) (raise e)])
  body ...)
```

Behavior of unhandled exception

If an exception is raised where no program-defined exception handler is installed, the following action is taken.

If an unhandled exception occurs within a thread other than the primordial one, it terminates the thread, and the thrown condition is wrapped by `<uncaught-exception>` condition and stored in the thread object. If other thread calls `thread-join!` to retrieve result, the the `<uncaught-exception>` is thrown in that thread. Note that no messages are displayed when the original uncaught exception is thrown. See Section 9.33.1 [Thread programming tips], page 458, for the details.

1. Otherwise, if the program is running interactively (in repl), the information of the thrown exception and stack trace are displayed, and the program returns to the toplevel prompt.

2. If the program is running non-interactively, the information of the thrown exception and stack trace are displayed, then the program exits with an exit status `EX_SOFTWARE` (70).

The default error message and stack trace in the above case 2 and case 3 is printed by `report-error` procedure. You can use it in your error handler if you need the same information.

`report-error` *exn :optional sink* [Function]

Prints type and message of a thrown condition object *exn*, then print the current stack trace. This is the procedure the system calls when you see an error reported on REPL.

Since you can `raise` any object, *exn* can be any object; it's not needed to be an instance of `<condition>`. A suitable message is chosen by `report-error`.

You can specify where the output goes by the optional *sink* argument: If it is an output port, the output goes there; you can also pass `#t` for the current output port and `#f` for the output string port, just like `format`. That is, when you pass `#f`, the message goes to a temporary output string port, and gathered string is returned. For all the other cases, an undefined value is returned. If *sink* is omitted or any other object listed above, the current error port is used.

Note: As of 0.9.5, this procedure prints stack trace of the context where `report-error` is called, rather than the context where *exn* is thrown. It doesn't matter much as far as you call `report-error` directly inside the error handler, but in general what you want to print is the latter, and we have a plan to attach stack trace info to `<condition>` object in future.

Low-level exception handling mechanism

This layer provides SRFI-18 compatible simple exception mechanism. You can override the behavior of higher-level constructs such as `with-error-handler` by using `with-exception-handler`.

Note that it is a double-edged sword. You'll get a freedom to construct your own exception handling semantics, but the Gauche system won't save if something goes wrong. Use these primitives when you want to customize the system's higher-level semantics or you are porting from other SRFI-18 code.

`current-exception-handler` [Function]

[SRFI-18] Returns the current exception handler.

`with-exception-handler` *handler thunk* [Function]

[SRFI-18] A procedure *handler* must take one argument. This procedure sets *handler* to the current exception handler and calls *thunk*. (Note that this slightly differs from R7RS `with-exception-handler`; we'll explain it below.)

When an exception is raised by `raise` or `error`, *handler* is called with the thrown condition in the exactly same dynamic environment of `raise` or `error`. It means the exception handler is also the same, so calling `raise` in *handler* reinvokes *handler* again. It also means that when *handler* returns, it returns from *raise*.

The behavior is specified in SRFI-18, intending this procedure to be the most primitive building block of exception handling mechanism. If you need to switch handlers, you can do it by yourself.

If you need the typical semantics where raising exception in the exception handler is handled by outer handler, you should use `guard`. Use this procedure only when you want to play with the most primitive layer of exception handling.

R7RS has a procedure with the same name, but has one difference—it changes the active exception handler to the "outer" handler before calling *handler*. See Section 10.2.2 [R7RS base library], page 504, for the description of R7RS's `with-exception-handler`.

If an exception is raised by `error`, or the thrown condition inherits `<serious-condition>`, it is prohibited to return from *handler*. If *handler* ever returns in such cases, another error is signaled, with replacing the current exception handler to the outer handler. So the caller of `error`, or the caller of `raise` with `<serious-condition>`, can assume it never returns.

The behavior of those procedures can be explained in the following conceptual Scheme code.

```
;; Conceptual implementation of low-level exception mechanism.
;; Suppose %xh is a list of exception handlers

(define (current-exception-handler) (car %xh))

(define (raise exn)
  (receive r ((car %xh) exn)
    (when (uncontinuable-exception? exn)
      (set! %xh (cdr %xh))
      (raise (make-error "returned from uncontinuable exception")))
    (apply values r)))

(define (with-exception-handler handler thunk)
  (let ((prev %xh))
    (dynamic-wind
      (lambda () (set! %xh (cons handler %xh)))
      thunk
      (lambda () (set! %xh prev)))))
```

6.20.4 Conditions

Built-in Condition classes

Gauche currently has the following hierarchy of built-in condition classes. It approximately reflects SRFI-35 and SRFI-36 condition hierarchy, although they have Gauche-style class names. If there's a corresponding SRFI condition type, the class has the SRFI name as well.

```
<condition>
  +- <compound-condition>
  +- <serious-condition>
  |    +- <serious-compound-condition> ; also inherits <compound-condition>
  +- <message-condition>
       +- <error>                      ; also inherits <serious-condition>
            +- <system-error>
            +- <unhandled-signal-error>
            +- <read-error>
            +- <io-error>
                 +- <port-error>
                      +- <io-read-error>
                      +- <io-write-error>
                      +- <io-closed-error>
                      +- <io-unit-error>
```

Note that some conditions may occur simultaneously; for example, error during reading from a file because of device failure may consist both `<system-error>` *and* `<io-read-error>`. In such cases, a *compound condition* is raised. So you can't just use, for instance, `(is-a? obj <io-read-error>)` to check if `<io-read-error>` is thrown. See the "Condition API" section below.

`<condition-meta>` [Metaclass]

Every condition class is an instance of this class. This class defines `object-apply` so that you can use a condition class as a predicate, e.g.:

```
(<error> obj) ≡ (condition-has-type? obj <error>)
```

`<condition>` [Class]
`&condition` [Condition Type]

[SRFI-35] The root class of the condition hierarchy.

`<compound-condition>` [Class]

Represents a compound condition. A compound condition can be created from one or more conditions by `make-compound-condition`. Don't use this class directly.

A compound condition returns `#t` for `condition-has-type?` if any of the original conditions has the given type.

`<serious-condition>` [Class]
`&serious` [Condition Type]

[SRFI-35] Conditions of this class are for the situations that are too serious to ignore or continue. Particularly, you can safely assume that if you `raise` this type of condition, it never returns.

`<serious-compound-condition>` [Class]

This is an internal class to represent a compound condition with any of its component condition is serious. Inherits both `<compound-condition>` and `<serious-condition>`. `make-compound-condition` uses this class if the passed conditions includes a serious one. Don't use this class directly.

`<message-condition>` [Class]
`&message` [Condition Type]

[SRFI-35] This class represents a condition with a message. It has one slot.

`message` [Instance Variable of `<message-condition>`]

A message.

`<error>` [Class]
`&error` [Condition Type]

[SRFI-35] Indicates an error. Inherits `<serious-condition>` and `<message-condition>`, thus has `message` slot.

Note: SRFI-35 `&error` condition only inherits `&serious` and not `&message`, so you have to use compound condition to attach a message to the error condition. Gauche uses multiple inheritance here, largely because of backward compatibility. To write a portable code, an error condition should be used with a message condition, like this:

```
(condition
  (&message (message "Error message"))
  (&error))
```

`<system-error>` [Class]

A subclass of `<error>`. When a system call returns an error, this type of exception is thrown. The `message` slot usually contains the description of the error (like the one from `strerror(3)`). Besides that, this class has one more instance slot:

`errno` [Instance Variable of `<system-error>`]

Contains an integer value of system's error number.

Error numbers may differ among systems. Gauche defines constants for typical Unix error values (e.g. `EACCES`, `EBADF`, etc), so it is desirable to use them instead of literal numbers. See the description of `sys-strerror` in Section 6.25.8 [System inquiry], page 266, for available constants.

This class doesn't have corresponding SRFI condition type, but important to obtain OS's raw error code. In some cases, this type of condition is compounded with other condition types, like `<io-read-error>`.

`<unhandled-signal-error>` [Class]

A subclass of `<error>`. The default handler of most of signals raises this condition. See Section 6.25.7.3 [Handling signals], page 262, for the details.

`signal` [Instance Variable of `<unhandled-signal-error>`]

An integer indicating the received signal number. There are constants defined for typical signal numbers; see Section 6.25.7.1 [Signals and signal sets], page 260.

`<read-error>` [Class]
`&read-error` [Condition Type]

[SRFI-36] A subclass of `<error>`. When the reader detects a lexical or syntactic error during reading an S-expression, this type of condition is raised.

`port` [Instance Variable of `<read-error>`]

A port from which the reader is reading. (NB: SRFI-36's `&read-error` doesn't have this slot. Portable program shouldn't rely on this slot).

`line` [Instance Variable of `<read-error>`]

A line count (1-base) of the input where the reader raised this error. It may be -1 if the reader is reading from a port that doesn't keep track of line count.

`column` [Instance Variable of `<read-error>`]
`position` [Instance Variable of `<read-error>`]
`span` [Instance Variable of `<read-error>`]

These slots are defined in SRFI-36's `&read-error`. For the time being, these slots always hold `#f`.

`<io-error>` [Class]
`&io-error` [Condition Type]

[SRFI-36] A base class of I/O errors. Inherits `<error>`.

`<port-error>` [Class]
`&io-port-error` [Condition Type]

[SRFI-36] An I/O error related to a port. Inherits `<io-error>`.

`port` [Instance Variable of `<port-error>`]

Holds the port where the error occurred.

`<io-read-error>` [Class]
`&io-read-error` [Condition Type]

[SRFI-36] An I/O error during reading from a port. Inherits `<port-error>`.

`<io-write-error>` [Class]
`&io-write-error` [Condition Type]

[SRFI-36] An I/O error during writing to a port. Inherits `<port-error>`.

`<io-closed-error>` [Class]
`&io-closed-error` [Condition Type]

[SRFI-36] An I/O error when read/write is attempted on a closed port. Inherits `<port-error>`.

`<io-unit-error>` [Class]

An I/O error when the read/write is requested with a unit that is not supported by the port (e.g. a binary I/O is requested on a character-only port). Inherits `<port-error>`.

Condition API

`define-condition-type` *name supertype predicate field-spec . . .* [Macro]

[SRFI-35+] Defines a new condition type. In Gauche, a condition type is a class, whose metaclass is `<condition-meta>`.

Name becomes the name of the new type, and also the variable of that name is bound to the created condition type. *Supertype* is the name of the supertype (direct superclass) of this condition type. A condition type must inherit from `<condition>` or its descendants. (Multiple inheritance can't be specified by this form, and generally should be avoided in condition type hierarchy. Instead, you can use compound conditions, which don't introduce multiple inheritance.)

A variable *predicate* is bound to a predicate procedure for this condition type.

Each `field-spec` is a form of `(field-name accessor-name)`, and the condition will have fields named by *field-name*, and a variable *accessor-name* will be bound to a procedure that accesses the field. In Gauche, each field becomes a slot of the created class.

Gauche extends srfi-35 to allow *predicate* and/or `accessor-name` to be `#f`, or `accessor-name` to be omitted, if you don't need to them to be defined.

When `define-condition-type` is expanded into a class definition, each slot gets a `:init-keyword` slot option with the keyword whose name is the same as the slot name.

`condition-type?` *obj* [Function]

[SRFI-35] Returns `#t` iff *obj* is a condition type. In Gauche, it means `(is-a? obj <condition-meta>)`.

`make-condition-type` *name parent field-names* [Function]

[SRFI-35] A procedural version to create a new condition type.

`make-condition` *type field-name value . . .* [Function]

[SRFI-35] Creates a new condition of condition-type *type*, and initializes its fields as specified by `field-name` and `value` pairs.

`condition?` *obj* [Function]

[SRFI-35] Returns `#t` iff *obj* is a condition. In Gauche, it means `(is-a? obj <condition>)`.

`condition-has-type?` *obj type* [Function]

[SRFI-35] Returns `#t` iff *obj* belongs to a condition type *type*. Because of compound conditions, this is not equivalent to `is-a?`.

`condition-ref` *condition field-name* [Function]

[SRFI-35] Retrieves the value of field *field-name* of *condition*. If *condition* is a compound condition, you can access to the field of its original conditions; if more than one original condition have *field-name*, the first one passed to `make-compound-condition` has precedence.

You can use `slot-ref` and/or `ref` to access to the field of conditions; compound conditions define a `slot-missing` method so that `slot-ref` behaves as if the compound conditions have all the slots of the original conditions. Using `condition-ref` increases portability, though.

`make-compound-condition` *condition0 condition1 . . .* [Function]

[SRFI-35] Returns a compound condition that has all *condition0 condition1 . . .*. The returned condition's fields are the union of all the fields of given conditions; if any conditions have the same name of fields, the first one takes precedence. The returned condition also has condition-type of all the types of given conditions. (This is not a multiple inheritance. See `<compound-condition>` above.)

`extract-condition` *condition condition-type* [Function]

[SRFI-35] *Condition* must be a condition and have type *condition-type*. This procedure returns a condition of *condition-type*, with field values extracted from *condition*.

`condition` *type-field-binding . . .* [Macro]

[SRFI-35] A convenience macro to create a (possibly compound) condition. *Type-field-binding* is a form of `(`*`condition-type`* `(`*`field-name value-expr`*`) ...)`.

```
(condition
  (type0 (field00 value00) ...)
  (type1 (field10 value10) ...)
  ...)
 ≡
(make-compound-condition
  (make-condition type0 'field00 value00 ...)
  (make-condition type1 'field10 value10 ...)
  ...)
```

6.21 Eval and repl

`eval` *expr env* [Function]

[R7RS eval] Evaluate *expr* under the environment *env*. In Gauche, *env* is just a `<module>` object.

R5RS and R7RS provides a portable way to obtain environment specifiers. R5RS way is described below. R7RS way is described in Section 10.2.7 [R7RS eval], page 508.

`null-environment` *version* [Function]
`scheme-report-environment` *version* [Function]
`interaction-environment` [Function]

[R5RS] Returns an environment specifier which can be used as the second argument of `eval`. Right now an environment specifier is just a module. `(null-environment 5)` returns a `null` module, which contains just the syntactic bindings specified in R5RS, `(scheme-report-environment 5)` returns a `scheme` module, which contains syntactic and procedure bindings in R5RS, and `(interaction-environment)` returns a `user` module that contains all the Gauche built-ins plus whatever the user defined. It is possible that the Gauche adopts a first-class environment object in future, so do not rely on the fact that the environment specifier is just a module.

An error is signaled if a value other than 5 is passed as *version* argument.

`read-eval-print-loop` *:optional reader evaluator printer prompter* [Function]

This exports `Gosh`'s default read-eval-print loop to applications. Each argument can be `#f`, which indicates it to use Gauche's default procedure(s), or a procedure that satisfies the following conditions.

reader A procedure that takes no arguments. It is supposed to read an expression and returns it.

evaluator A procedure that takes two arguments, an expression and an environment specifier. It is supposed to evaluate the expression and returns zero or more value(s).

printer A procedure that takes zero or more arguments. It is supposed to print out these values. The result of this procedure is discarded.

prompter A procedure that takes no arguments. It is supposed to print out the prompt. The result of this procedure is discarded.

Given those procedures, `read-eval-print-loop` runs as follows:

1. Prints the prompt by calling *prompter*.
2. Reads an expression by calling *reader*. If it returns EOF, exits the loop and returns from `read-eval-print-loop`.
3. Evaluates an expression by calling *evaluator*
4. Prints the result by calling *printer*, then repeats from 1.

When an error is signaled from one of those procedures, it is captured and reported by the default escape handler, then the loop restarts from 1.

It is OK to capture a continuation within those procedures and re-invoke them afterwards.

6.22 Input and Output

6.22.1 Ports

`<port>` [Builtin Class]

A port class. A port is Scheme's way of abstraction of I/O channel. Gauche extends a port in number of ways so that it can be used in wide range of applications.

Textual and binary I/O

R7RS defines textual and binary ports. In Gauche, most ports can mix both text I/O and binary I/O. It is cleaner to think the two is distinct, for they are sources/sinks of different types of objects and you don't need to mix textual and binary I/O.

In practice, however, a port is often a tap to an untyped pool of bytes and you may want to decide interpret it later. One example is the standard I/O; in Unix-like environment, it's up to the program to use pre-opened ports for textual or binary I/O. R7RS defines the initial ports for `current-input-port` etc. are textual ports; in Gauche, you can use either way.

Conversion

Some ports can be used to convert a data stream from one format to another; one of such applications is character code conversion ports, provided by `gauche.charconv` module (see Section 9.4 [Character code conversion], page 339, for details).

Extra features

There are also a ports with special functionality. A coding-aware port (see Section 6.22.6 [Coding-aware ports], page 226) recognizes a special "magic comment" in the file to know which character encoding the file is written. Virtual ports (see Section 9.38 [Virtual ports], page 493) allows you to program the behavior of the port in Scheme.

6.22.2 Port and threads

When Gauche is compiled with thread support, the builtin port operations locks the port, so that port access from multiple threads will be serialized. (It is required by SRFI-18, BTW). Here, "builtin port operations" are the port access functions that takes a port and does some I/O or query on it, such as `read/write`, `read-char/write-char`, `port->string`, etc. Note that `call-with-*` and `with-*` procedures do not lock the port during calling the given procedures, since the procedure may pass the reference of the port to the other thread, and Gauche wouldn't know if that's the case.

This means you don't need to be too paranoia to worry about ports under multithreaded environment. However, keep it in mind that this locking mechanism is meant to be a safety net from breaking the port's internal state, and not to be a general mutex mechanism. It assumes port accesses rarely conflict, and uses spin lock to reduce the overhead of majority cases. If you know there will be more than one thread accessing the same port, you should use explicit mutex to avoid conflicts.

`with-port-locking` *port thunk* [Function]

Executes *thunk*, while making the calling thread hold the exclusive lock of *port* during the dynamic extent of *thunk*.

Calls of the builtin port functions during the lock is held would bypass mutex operations and yield better performance.

Note that the lock is held during the dynamic extent of *thunk*; so, if *thunk* invokes a continuation captured outside of `with-port-locking`, the lock is released. If the continuation captured within *thunk* is invoked afterwards, the lock is re-acquired.

`With-port-locking` may be nested. The lock is valid during the outermost call of `with-port-locking`.

Note that this procedure uses the port's built-in lock mechanism which uses busy wait when port access conflicts. It should be used only for avoiding fine-grain lock overhead; use explicit mutex if you know there will be conflicts.

6.22.3 Common port operations

`port?` *obj* [Function]
`input-port?` *obj* [Function]
`output-port?` *obj* [Function]

[R7RS base] Returns true if *obj* is a port, an input port and an output port, respectively. `Port?` is not listed in the R5RS standard procedures, but mentioned in the "Disjointness of Types" section.

`port-closed?` *port* [Function]

Returns true if *obj* is a port and it is already closed. A closed port can't be reopened.

`current-input-port` [Parameter]
`current-output-port` [Parameter]
`current-error-port` [Parameter]

[R7RS base] Returns the current input, output and error output port, respectively.

R7RS defines that the initial values of these ports are textual ports. In Gauche, initial ports can handle both textual and binary I/O.

Values of the current ports can be temporarily changed by `parameterize` (see Section 9.22 [Parameters], page 411), though you might want the convenience procedures such as `with-output-to-string` or `with-input-from-file` in typical cases.

```
(use gauche.parameter)
```

```
(let1 os (open-output-string)
  (parameterize ((current-output-port os))
    (display "foo"))
  (get-output-string os))
 ⇒ "foo"
```

`standard-input-port` [Parameter]
`standard-output-port` [Parameter]
`standard-error-port` [Parameter]

Returns standard i/o ports at the time the program started. These ports are the default values of `current-input-port`, `current-output-port` and `current-error-port`, respectively.

You can also change value of these procedures by `parameterize`, but note that (1) `current-*-port`s are initialized before the program execution, so changing values of `standard-*-port` won't affect them, and (2) changing values these procedures only affect Scheme-world, and does not change system-level stdio file descriptors low-level libraries referring.

`with-input-from-port` *port thunk* [Function]
`with-output-to-port` *port thunk* [Function]
`with-error-to-port` *port thunk* [Function]

Calls *thunk*. During evaluation of thunk, the current input port, current output port and current error port are set to *port*, respectively. Note that *port* won't be closed after thunk is executed.

`with-ports` *iport oport eport thunk* [Function]

Does the above three functions at once. Calls *thunk* while the current input, output, and error ports are set to *iport*, *oport*, and *eport*, respectively. You may pass `#f` to any port argument(s) if you don't need to alter the port(s).

Note that *port* won't be closed after thunk is executed. (Unfortunately, recent Scheme standards added a similar named procedure, `call-with-port`, which does close the port. See below.)

`close-port` *port* [Function]
`close-input-port` *port* [Function]
`close-output-port` *port* [Function]

[R7RS base] Closes the port. `Close-port` works both input and output ports, while `close-input-port` and `close-output-port` work only for the respective ports and throws an error if another type of port is passed.

Theoretically, only `close-port` would suffice; having those three is merely for historical reason. R5RS has `close-input-port` and `close-output-port`; R6RS and R7RS support all three.

`call-with-port` *port proc* [Function]

[R7RS base] Calls *proc* with one argument, *port*. After *proc* returns, or it throws an uncaptured error, *port* is closed. Value(s) returned from *proc* will be the return value(s) of `call-with-port`.

`port-type` *port* [Function]

Returns the type of *port* in one of the symbols `file`, `string` or `proc`.

`port-name` *port* [Function]

Returns the name of *port*. If the port is associated to a file, it is the name of the file. Otherwise, it is some description of the port.

`port-buffering` *port* [Function]
`(setter port-buffering)` *port buffering-mode* [Function]

If *port* is type of file port (i.e. `(port-type port)` returns `file`), these procedures gets and sets the port's buffering mode. For input ports, the port buffering mode may be either one of `:full`, `:modest` or `:none`. For output ports, `port-buffering`, it may be one of `:full`, `:line` or `:none`. See Section 6.22.4 [File ports], page 221, for explanation of those modes.

If `port-buffering` is applied to ports other than file ports, it returns `#f`. If the setter of port-buffering is applied to ports other than file ports, it signals an error.

`port-current-line` *port* [Function]

Returns the current line count of *port*. This information is only available on file-based port, and as long as you're doing sequential character I/O on it. Otherwise, this returns -1.

`port-file-number` *port* [Function]

Returns an integer file descriptor, if the *port* is associated to the system file I/O. Returns `#f` otherwise.

`port-seek` *port offset :optional whence* [Function]

If the given *port* allows random access, this procedure sets the read/write pointer of the *port* according to the given *offset* and *whence*, then returns the updated offset (number of bytes from the beginning of the data). If *port* is not random-accessible, `#f` is returned. In the current version, file ports and input string ports are fully random-accessible. You can only query the current byte offset of output string ports.

Note that port position is represented by byte count, not character count.

It is allowed to seek after the data if *port* is an output file port. See POSIX lseek(2) document for details of the behavior. For input file port and input string port, you can't seek after the data.

The *whence* argument must be a small integer that represents from where *offset* should be counted. The following constant values are defined.

`SEEK_SET` *Offset* represents the byte count from the beginning of the data. This is the default behavior when *whence* is omitted.

`SEEK_CUR` *Offset* represents the byte count relative to the current read/write pointer. If you pass 0 to *offset*, you can get the current port position without changing it.

`SEEK_END` *Offset* represents the byte count relative to the end of the data.

`port-tell` *port* [Function]

Returns the current read/write pointer of *port* in byte count, if *port* is random-accessible. Returns `#f` otherwise. This is equivalent to the following call:

```
(port-seek port 0 SEEK_CUR)
```

Note on the names: `Port-seek` is called `seek`, `file-position` or `input-port-position`/`output-port-position` on some implementations. `Port-tell` is called `tell`, `ftell` or `set-file-position!`. Some implementations have `port-position` for different functionality. CommonLisp has `file-position`, but it is not suitable for us since *port* need not be a file port. `Seek` and `tell` reflects POSIX name, and with Gauche naming convention we could use `sys-seek` and `sys-tell`; however, *port* deals with higher level of abstraction than system calls, so I dropped those names, and adopted new names.

`copy-port` *src dst :key* (*unit 0*) (*size #f*) [Function]

Copies data from an input port *src* to an output port *dst*, until eof is read from *src*.

The keyword argument *unit* may be zero, a positive exact integer, a symbol `byte` or a symbol `char`, to specify the unit of copying. If it is an integer, a buffer of the size (in case of zero, a system default size) is used to copy, using block I/O. Generally it is the fastest if you copy between normal files. If *unit* is a symbol `byte`, the copying is done byte by byte, using C-version of `read-byte` and `write-byte`. If *unit* is a symbol `char`, the copying is done character by character, using C-version of `read-char` and `write-char`.

If nonnegative integer is given to the keyword argument *size*, it specifies the maximum amount of data to be copied. If *unit* is a symbol `char`, *size* specifies the number of characters. Otherwise, *size* specifies the number of bytes.

Returns number of characters copied when *unit* is a symbol `char`. Otherwise, returns number of bytes copied.

6.22.4 File ports

`open-input-file` *filename :key if-does-not-exist buffering element-type encoding conversion-buffer-size* [Function]

`open-output-file` *filename :key if-does-not-exist if-exists buffering element-type encoding conversion-buffer-size* [Function]

[R7RS+] Opens a file *filename* for input or output, and returns an input or output port associated with it, respectively.

The keyword arguments specify precise behavior.

`:if-exists`
: This keyword argument can be specified only for `open-output-file`, and specifies the action when the *filename* already exists. One of the following value can be given.

  `:supersede`
  : The existing file is truncated. This is the default behavior.

  `:append`
  : The output data will be appended to the existing file.

  `:overwrite`
  : The output data will overwrite the existing content. If the output data is shorter than the existing file, the rest of existing file remains.

  `:error`
  : An error is signaled.

  `#f`
  : No action is taken, and the function returns `#f`.

`:if-does-not-exist`
: This keyword argument specifies the action when *filename* does not exist.

  `:error`
  : An error is signaled. This is the default behavior of `open-input-file`.

  `:create`
  : A file is created. This is the default behavior of `open-output-file`. The check of file existence and creation is done atomically; you can exclusively create the file by specifying `:error` or `#f` to *if-exists*, along this option. You can't specify this value for `open-input-file`.

  `#f`
  : No action is taken, and the function returns `#f`.

`:buffering`
: This argument specifies the buffering mode. The following values are allowed. The port's buffering mode can be get/set by `port-buffering`. (see Section 6.22.3 [Common port operations], page 218).

  `:full`
  : Buffer the data as much as possible. This is the default mode.

`:none` No buffering is done. Every time the data is written (to an output port) or read (from an input port), the underlying system call is used. Process's standard error port is opened in this mode by default.

`:line` This is valid only for output ports. The written data is buffered, but the buffer is flushed whenever a newline character is written. This is suitable for interactive output port. Process's standard output port is opened in this mode by default. (Note that this differs from the line buffering mode of C stdio, which flushes the buffer as well when input is requested from the same file descriptor.)

`:modest` This is valid only for input ports. This is almost the same as the mode `:full`, except that `read-uvector` may return less data than requested if the requested amount of data is not immediately available. (In the `:full` mode, `read-uvector` waits the entire data to be read). This is suitable for the port connected to a pipe or network.

`:element-type`
This argument specifies the type of the file.

`:binary` The file is opened in "binary" mode. (This is the default)

`:character`
The file is opened in "character" (or "text") mode.

Note: This flag makes difference only on Windows-native platforms, and only affect the treatment of line terminators. In character mode, writing `#\newline` on the output causes CR + LF characters to be written, instead of just LF. And reading CR + LF sequence returns just `#\newline`.

On Unix, both mode are the same.

Note that Gauche doesn't distinguish character (textual) port and binary port. So this flag really matters only on Windows line terminators.

`:encoding`
This argument specifies character encoding of the file. The argument is a string or a symbol that names a character encoding scheme (CES).

For `open-input-file`, it can be a wildcard CES (e.g. `*jp`) to guess the file's encoding heuristically (see Section 9.4.2 [Autodetecting the encoding scheme], page 341), or `#t`, in which case we assume the input file itself has magic encoding comment and use `open-coding-aware-port` (see Section 6.22.6 [Coding-aware ports], page 226).

If this argument is given, Gauche automatically loads `gauche.charconv` module and converts the input/output characters as you read to or write from the port. See Section 9.4.1 [Supported character encoding schemes], page 340, for the details of character encoding schemes.

`:conversion-buffer-size`
This argument may be used with the *encoding* argument to specify the buffer size of character encoding conversion. It is passed as a *buffer-size* argument of the conversion port constructors (see Section 9.4.3 [Conversion ports], page 342).

Usually you don't need to give this argument; but if you need to guess the input file encoding, larger buffer size may work better since guessing routine can have more data before deciding the encoding.

By combination of *if-exists* and *if-does-not-exist* flags, you can implement various actions:

```
(open-output-file "foo" :if-exists :error)
```

```
⇒ ;opens "foo" exclusively, or error

(open-output-file "foo" :if-exists #f)
 ⇒ ;opens "foo" exclusively, or returns #f

(open-output-file "foo" :if-exists :append
                        :if-does-not-exist :error)
 ⇒ ;opens "foo" for append only if it already exists
```

To check the existence of a file without opening it, use `sys-access` or `file-exists?` (see Section 6.25.4.4 [File stats], page 255).

Note for portability: Some Scheme implementations (e.g. STk) allows you to specify a command to *filename* and reads from, or writes to, the subprocess standard input/output. Some other scripting languages (e.g. Perl) have similar features. In Gauche, `open-input-file` and `open-output-file` strictly operates on files (what the underlying OS thinks as files). However, you can use "process ports" to invoke other command in a subprocess and to communicate it. See Section 9.25.4 [Process ports], page 429, for details.

`call-with-input-file` *string proc :key if-does-not-exist buffering element-type encoding conversion-buffer-size* [Function]

`call-with-output-file` *string proc :key if-does-not-exist if-exists buffering element-type encoding conversion-buffer-size* [Function]

[R7RS+] Opens a file specified by *string* for input/output, and call *proc* with one argument, the file port. When *proc* returns, or an error is signaled from *proc* that is not captured within *proc*, the file is closed.

The keyword arguments have the same meanings of `open-input-file` and `open-output-file`'s. Note that if you specify `#f` to *if-exists* and/or *if-does-not-exist*, *proc* may receive `#f` instead of a port object when the file is not opened.

Returns the value(s) `proc` returned.

`with-input-from-file` *string thunk :key if-does-not-exist buffering element-type encoding conversion-buffer-size* [Function]

`with-output-to-file` *string thunk :key if-does-not-exist if-exists buffering element-type encoding conversion-buffer-size* [Function]

[R7RS file] Opens a file specified by *string* for input or output and makes the opened port as the current input or output port, then calls *thunk*. The file is closed when *thunk* returns or an error is signaled from *thunk* that is not captured within *thunk*.

Returns the value(s) `thunk` returns.

The keyword arguments have the same meanings of `open-input-file` and `open-output-file`'s, except that when `#f` is given to *if-exists* and *if-does-not-exist* and the opening port is failed, *thunk* isn't called at all and `#f` is returned as the result of `with-input-from-file` and `with-output-to-file`.

Notes on semantics of closing file ports: R7RS states, in the description of `call-with-port` et al., that "*If proc does not return, then the port will not be closed automatically unless it is possible to prove that the port will never again be used for read or write operation.*"

Gauche's implementation slightly misses this criteria; the mere fact that an uncaptured error is thrown in *proc* does not prove the port will never be used. Nevertheless, it is very difficult to think the situation that you can do meaningful operation on the port after such an error is signaled; you'd have no idea what kind of state the port is in. In practical programs, you should capture error explicitly inside *proc* if you still want to do some meaningful operation with the port.

Note that if a continuation captured outside `call-with-input-file` et al. is invoked inside *proc*, the port is not closed. It is possible that the control returns later into the *proc*, if a continuation is captured in it (e.g. coroutines). The low-level exceptions (see Section 6.20.3 [Handling exceptions], page 208) also doesn't ensure closing the port.

`open-input-fd-port` *fd :key buffering name owner?* [Function]
`open-output-fd-port` *fd :key buffering name owner?* [Function]

Creates and returns an input or output port on top of the given file descriptor. *Buffering* specifies the buffering mode as described in `open-input-file` entry above; the default is `:full`. *Name* is used for the created port's name and returned by `port-name`. A boolean flag `owner?` specifies whether *fd* should be closed when the port is closed.

`port-fd-dup!` *toport fromport* [Function]

Interface to the system call `dup2(2)`. Atomically closes the file descriptor associated to *toport*, creates a copy of the file descriptor associated to *fromport*, and sets the new file descriptor to *toport*. Both *toport* and *fromport* must be file ports. Before the original file descriptor of *toport* is closed, any buffered output (when *toport* is an output port) is flushed, and any buffered input (when *toport* is an input port) is discarded.

'Copy' means that, even the two file descriptors differ in their values, they both point to the same system's open file table entry. For example they share the current file position; after *port-fd-dup!*, if you call `port-seek` on *fromport*, the change is also visible from *toport*, and vice versa. Note that this 'sharing' is in the system-level; if either *toport* or *fromport* is buffered, the buffered contents are not shared.

This procedure is mainly intended for programs that needs to control open file descriptors explicitly; e.g. a daemon process would want to redirect its I/O to a harmless device such as `/dev/null`, and a shell process would want to set up file descriptors before executing the child process.

6.22.5 String ports

String ports are the ports that you can read from or write to memory.

`open-input-string` *string :key name* [Function]

[R7RS base][SRFI-6] Creates an input string port that has the content *string*. This is a more efficient way to access a string in order rather than using `string-ref` with incremental index.

```
(define p (open-input-string "foo x"))
(read p) ⇒ foo
(read-char p) ⇒ #\space
(read-char p) ⇒ #\x
(read-char p) ⇒ #<eof>
(read-char p) ⇒ #<eof>
```

The *name* keyword argument is a Gauche extension. By default, the created port is named as `(input string port)`. It is mainly used for debugging. You can specify alternative name with this argument. As Gauche's convention, file ports has the source file path as its name, so port names for debugging information should be parenthesized not to be taken as pathnames.

```
gosh> (open-input-string "")
#<iport (input string port) 0x215c0c0>
gosh> (open-input-string "" :name "(user input)")
#<iport (user input) 0x22a4e40>
```

`get-remaining-input-string` *port* [Function]

Port must be an input string port. Returns the remaining content of the input port. The internal pointer of *port* isn't moved, so the subsequent read from *port* isn't affected. If *port* has already reached to EOF, a null string is returned.

```
(define p (open-input-string "abc\ndef"))
(read-line p)                   ⇒ "abc"
(get-remaining-input-string p) ⇒ "def"
(read-char p)                   ⇒ #\d
(read-line p)                   ⇒ "ef"
(get-remaining-input-string p) ⇒ ""
```

`open-output-string` *:key name* [Function]

[R7RS base][SRFI-6] Creates an output string port. Anything written to the port is accumulated in the buffer, and can be obtained as a string by `get-output-string`. This is a far more efficient way to construct a string sequentially than pre-allocate a string and fill it with `string-set!`.

The *name* keyword argument is a Gauche extension. By default, the created port is named as `(output string port)`. It is mainly used for debugging. You can specify alternative name with this argument. As Gauche's convention, file ports has the source file path as its name, so port names for debugging information should be parenthesized not to be taken as pathnames.

```
gosh> (open-output-string)
#<oport (output string port) 0x22a4c00>
gosh> (open-output-string :name "(temporary output)")
#<oport (temporary output) 0x22a49c0>
```

`get-output-string` *port* [Function]

[R7RS base][SRFI-6] Takes an output string port `port` and returns a string that has been accumulated to `port` so far. If a byte data has been written to the port, this function re-scans the buffer to see if it can consist a complete string; if not, an incomplete string is returned.

This doesn't affect the *port*'s operation, so you can keep accumulating content to *port* after calling `get-output-string`.

`call-with-input-string` *string proc* [Function]
`call-with-output-string` *proc* [Function]
`with-input-from-string` *string thunk* [Function]
`with-output-to-string` *thunk* [Function]

These utility functions are trivially defined as follows. The interface is parallel to the file port version.

```
(define (call-with-output-string proc)
  (let ((out (open-output-string)))
    (proc out)
    (get-output-string out)))

(define (call-with-input-string str proc)
  (let ((in (open-input-string str)))
    (proc in)))

(define (with-output-to-string thunk)
  (let ((out (open-output-string)))
    (with-output-to-port out thunk)
    (get-output-string out)))

(define (with-input-from-string str thunk)
  (with-input-from-port (open-input-string str) thunk))
```

`call-with-string-io` *str proc* [Function]
`with-string-io` *str thunk* [Function]

```
(define (call-with-string-io str proc)
  (let ((out (open-output-string))
        (in  (open-input-string str)))
    (proc in out)
    (get-output-string out)))

(define (with-string-io str thunk)
  (with-output-to-string
    (lambda ()
      (with-input-from-string str
        thunk))))
```

`write-to-string` *obj :optional writer* [Function]
`read-from-string` *string :optional start end* [Function]

These convenience functions cover common idioms using string ports.

```
(write-to-string obj writer)
  ≡
  (with-output-to-string (lambda () (writer obj)))

(read-from-string string)
  ≡
  (with-input-from-string string read)
```

The default value of *writer* is the procedure `write`. The default values of *start* and *end* is 0 and the length of *string*.

Portability note: Common Lisp has these functions, with different optional arguments. STk has `read-from-string` without optional argument.

6.22.6 Coding-aware ports

A coding-aware port is a special type of procedural input port that is used by `load` to read a program source. The port recognizes the magic comment to specify the character encoding of the program source, such as `;; -*- coding: utf-8 -*-`, and makes an appropriate character encoding conversion. See Section 2.3 [Multibyte scripts], page 11, for the details of coding magic comment.

`open-coding-aware-port` *iport* [Function]

Takes an input port and returns an input coding aware port, which basically just pass through the data from *iport* to its reader. However, if a magic comment appears within the first two lines of data from *iport*, the coding aware port applies the necessary character encoding conversion to the rest of the data as they are read.

The passed port, *iport*, is "owned" by the created coding-aware port. That is, when the coding-aware port is closed, *iport* is also closed. The content read from *iport* is buffered in the coding-aware port, so other code shouldn't read from *iport*.

By default, Gauche's `load` uses a coding aware port to read the program source, so that the coding magic comment works for the Gauche source programs (see Section 6.23.1 [Loading Scheme file], page 239). However, since the mechanism itself is independent from `load`, you can use this port for other purposes; it is particularly useful to write a function that processes Scheme source programs which may have the coding magic comment.

6.22.7 Input

For the input-related procedures, the optional *iport* argument must be an input port, and when omitted, the current input port is assumed.

6.22.7.1 Reading data

`read` *:optional iport* [Function]

[R7RS base] Reads an S-expression from *iport* and returns it. Gauche recognizes the lexical structure specified in R7RS, and some additional lexical structures listed in Section 4.1 [Lexical structure], page 38.

If *iport* has already reached to the end of file, an eof object is returned.

The procedure reads up to the last character that consists the S-expression, and leaves the rest in the port. It's not like CommonLisp's `read`, which consumes whitespaces after S-expression by default.

`read-with-shared-structure` *:optional iport* [Function]
`read/ss` *:optional iport* [Function]

[SRFI-38] These procedures are defined in srfi-38 to recognize shared substructure notation (`#n=`, `#n#`). Gauche's builtin `read` recognizes the srfi-38 notation, so these are just synonyms to `read`; these are only provided for srfi-38 compatibility.

`read-char` *:optional iport* [Function]

[R7RS base] Reads one character from *iport* and returns it. If *iport* has already reached to the end, returns an eof object. If the byte stream in *iport* doesn't consist a valid character, the behavior is undefined. (In future, a port will have a option to deal with invalid characters).

`peek-char` *:optional iport* [Function]

[R7RS base] Reads one character in *iport* and returns it, keeping the character in the *port*. If the byte stream in *iport* doesn't consist a valid character, the behavior is undefined. (In future, a port will have a option to deal with invalid characters).

`read-byte` *:optional iport* [Function]

Reads one byte from an input port *iport*, and returns it as an integer in the range between 0 and 255. If *iport* has already reached EOF, an eof object is returned.

This is called `read-u8` in R7RS.

`peek-byte` *:optional iport* [Function]

Peeks one byte at the head of an input port *iport*, and returns it as an integer in the range between 0 and 255. If *iport* has already reached EOF, an eof object is returned.

This is called `peek-u8` in R7RS.

`read-line` *:optional iport allow-byte-string?* [Function]

[R7RS+] Reads one line (a sequence of characters terminated by newline or EOF) and returns a string. The terminating newline is not included. This function recognizes popular line terminators (LF only, CRLF, and CR only). If *iport* has already reached EOF, an eof object is returned.

If a byte sequence is read from *iport* which doesn't constitute a valid character in the native encoding, `read-line` signals an error by default. However, if a true value is given to the argument *allow-byte-string?*, `read-line` returns a byte string (incomplete string) in such case, without reporting an error. It is particularly useful if you read from a source whose character encoding is not yet known; for example, to read XML document, you need to check the first line to see if there is a charset parameter so that you can then use an appropriate character conversion port. This optional argument is Gauche's extension to R7RS.

`read-string` *nchars :optional iport* [Function]
[R7RS base] Read *nchars* characters, or as many characters as available before EOF, and returns a string that consists of those characters. If the input has already reached EOF, an eof object is returned.

`read-block` *nbytes :optional iport* [Function]
This procedure is deprecated - use `read-uvector` instead (see Section 9.36.4 [Uvector block I/O], page 488).

Reads *nbytes* bytes from *iport*, and returns an incomplete string consisted by those bytes. The size of returned string may shorter than *nbytes* when *iport* doesn't have enough bytes to fill. If *nbytes* is zero, a null string is always returned.

If *iport* has already reached EOF, an eof object is returned.

If *iport* is a file port, the behavior of `read-block` differs by the buffering mode of the port (See Section 6.22.4 [File ports], page 221, for the detail explanation of buffering modes).

- If the buffering mode is `:full`, `read-block` waits until *nbytes* data is read, except it reads EOF.
- If the buffering mode is `:modest` or `:none`, `read-block` returns shorter string than *nbytes* even if it doesn't reach EOF, but the entire data is not available immediately.

If you want to write a chunk of bytes to a port, you can use either `display` if the data is in string, or `write-uvector` in `gauche.uvector` (see Section 9.36.4 [Uvector block I/O], page 488) if the data is in uniform vector.

`eof-object` [Function]
[R7RS base] Returns an EOF object.

`eof-object?` *obj* [Function]
[R7RS base] Returns true if *obj* is an EOF object.

`char-ready?` *:optional port* [Function]
[R7RS base] If a character is ready to be read from *port*, returns `#t`.

For now, this procedure actually checks only if next *byte* is immediately available from *port*. If the next byte is a part of a multibyte character, the attempt to read the whole character may block, even if `char-ready?` returns `#t` on the port. (It is unlikely to happen in usual situation, but theoretically it can. If you concern, use `read-uvector` to read the input as a byte sequence, then use input string port to read characters.)

`byte-ready?` *:optional port* [Function]
If one byte (octet) is ready to be read from *port*, returns `#t`.

In R7RS, this procedure is called `u8-ready?`

6.22.7.2 Reader lexical mode

`reader-lexical-mode` [Parameter]
Get/set the reader lexical mode. Changing this parameter switches behavior of the reader concerning some corner cases of the lexical syntax, where legacy Gauche syntax and R7RS syntax aren't compatible.

In general, you don't need to change this parameter directly. The lexical syntax matters at the read-time, while changing this parameter happens at the execution-time; unless you know the exact timing when each phase occurs, you might not get what you want.

The hash-bang directive `#!gauche-legacy` and `#!r7rs` indirectly affects this parameter; the first one sets the reader mode to `legacy`, and the second one to `strict-r7`.

The command-line argument `-fwarn-legacy` sets the default reader mode to `warn-legacy`.

Change to this parameter during `load` is delimited within that `load`; once `load` is done, the value of this parameter is reset to the value when `load` is started.

The parameter takes one of the following symbols as a value.

`permissive`
: This is the default mode. It tries to find a reasonable compromise between two syntax.

  In string literals, hex escape sequence is first interpreted as R7RS lexical syntax. If the syntax doesn't conform R7RS hex escape, it is interpreted as legacy Gauche hex escape syntax. For example, `"\x30;a"` is read as `"0a"`, for the hex escape sequence including the terminating semicolon is read as R7RS hex escape sequence. It also reads `"\x30a"` as `"0a"`, for the legacy Gauche hex escape always takes two hexadecimal digits without the terminator. With this mode, you can use R7RS hex escape syntax for the new code, and yet almost all legacy Gauche code can be read without a problem. However, if the legacy code has a semicolon followed by hex escape, it is interpreted as R7RS syntax and the incompatibility arises.

`strict-r7`
: Strict R7RS compatible mode. When the reader encounters the hash-bang directive `#!r7rs`, the rest of file is read with this mode.

  In this mode, Gauche's extended lexical syntax will raise an error.

  Use this mode to ensure the code can be read on other R7RS implementations.

`legacy`
: The reader works as the legacy Gauche (version 0.9.3.3 and before). When the reader encounters the hash-bang directive `#!gauche-legacy`, the rest of file is read with this mode.

  This only matters when you want to read two-digit hex escape followed by semicolon as a character plus a semicolon, e.g. `"\x30;a"` as `"0;a"` instead of `"0a"`. We expect such a sequence rarely appears in the code, but if you dump a data in a string literal format, you may have such sequence (especially in incomplete string literals).

`warn-legacy`
: The reader works as the `permissive` mode, but warns if it reads legacy hex-escape syntax. This mode is default when `-fwarn-legacy` command-line argument is given to `gosh`.

  This is useful to check if you have any incompatible escape sequence in your code.

6.22.7.3 Read-time constructor

Read-time constructor, defined in SRFI-10, provides an easy way to create an external representation of user-defined structures.

`#,(`*`tag arg ...`*`)` [Reader Syntax]

[SRFI-10] Gauche maintains a global table that associates a *tag* (symbol) to *a constructor procedure*.

When the reader encounters this syntax, it reads *arg* . . ., finds a reader constructor associated with *tag*, and calls the constructor with *arg* . . . as arguments, then inserts the value returned by the constructor as the result of reading the syntax.

Note that this syntax is processed inside the reader—the evaluator doesn't see any of *args*, but only sees the object the reader returns.

`define-reader-ctor` *tag procedure* [Function]

[SRFI-10] Associates a reader constructor *procedure* with *tag*.

Examples:

```
(define-reader-ctor 'pi (lambda () (* (atan 1) 4)))

#,(pi) ⇒ 3.141592653589793

'(#,(pi)) ⇒ (3.141592653589793)

(define-reader-ctor 'hash
  (lambda (type . pairs)
    (let ((tab (make-hash-table type)))
      (for-each (lambda (pair)
                  (hash-table-put! tab (car pair) (cdr pair)))
                pairs)
      tab)))

(define table
 #,(hash eq? (foo . bar) (duh . dah) (bum . bom)))

table ⇒ #<hash-table eq? 0x80f9398>
(hash-table-get table 'duh) ⇒ dah
```

Combined with `write-object` method (see Section 6.22.8 [Output], page 231), it is easy to make a user-defined class written in the form it can be read back:

```
(define-class <point> ()
  ((x :init-value 0 :init-keyword :x)
   (y :init-value 0 :init-keyword :y)))

(define-method write-object ((p <point>) out)
  (format out "#,(<point> ~s ~s)" (ref p 'x) (ref p 'y)))

(define-reader-ctor '<point>
  (lambda (x y) (make <point> :x x :y y)))
```

NOTE: The extent of the effect of `define-reader-ctor` is not specified in SRFI-10, and might pose a compatibility problem among implementations that support SRFI-10. (In fact, the very existence of `define-reader-ctor` is up to an implementation choice.)

In Gauche, at least for the time being, `define-reader-ctor` take effects as soon as the form is compiled and evaluated. Since Gauche compiles and evaluates each toplevel form in order, *tag* specified in `define-reader-ctor` can be used immediately after that. However, it doesn't work if the call of `define-reader-ctor` and the use of *tag* is enclosed in a `begin` form, for the entire `begin` form is compiled at once before being evaluated.

Other implementations may require to read the entire file before making its `define-reader-ctor` call effective. If so, it effectively prevents one from using `define-reader-ctor` and the defined *tag* in the same file. It is desirable to separate the call of `define-reader-ctor` and the use of *tag* in the different files if possible.

Another issue about the current `define-reader-ctor` is that it modifies the global table of Gauche system, hence it is not modular. The code written by different people might use the same tags, and yield an unexpected result. In future versions, Gauche may have some way to encapsulate the scope of *tag*, although the author doesn't have clear idea yet.

6.22.7.4 Input utility functions

`port->string` *port* [Function]
`port->list` *reader port* [Function]
`port->string-list` *port* [Function]
`port->sexp-list` *port* [Function]

Generally useful input procedures. The API is taken from scsh and STk.

`port->string` reads *port* until EOF and returns the accumulated data as a string.

`port->list` applies *reader* on *port* repeatedly, until *reader* returns an EOF, then returns the list of objects *reader* returned. Note that *port* isn't closed.

`port->string-list` is a `port->list` specialized by `read-line`, and `port->sexp-list` is a `port->list` specialized by `read`.

If the input contains an octet sequence that's not form a valid character in the Gauche's native character encoding, `port->string` and `port->string-list` may return incomplete string(s). If you want to deal with binary data, consider using `port->uvector` in `gauche.uvector` (see Section 9.36.4 [Uvector block I/O], page 488).

`port-fold` *fn knil reader* [Function]
`port-fold-right` *fn knil reader* [Function]
`port-for-each` *fn reader* [Function]
`port-map` *fn reader* [Function]

Convenient iterators over the input read by *reader*.

Since these procedures are not really about ports, they are superseded by `generator-fold`, `generator-fold-right`, `generator-for-each` and `generator-map`, respectively. See Section 6.18.9 [Folding generated values], page 197, for the details.

We provide these only for the backward compatibility.

6.22.8 Output

6.22.8.1 Layers of output routines

Gauche has quite a few output procedures which may confuse newcomers. The following table will help to understand how to use those procedures:

Object writers

Procedures that write out Scheme objects. Although there exist more low-level procedures, these are regarded as a basic layer of output routines, since it works on a generic Scheme object as a single unit. They come in two flavors:

- Write-family procedures: `write`, `write-shared`, `write-simple`–these are to produce *external representation* of Scheme objects, which can be generally read back by `read` without losing information as much as possible[1]. The external representation of most Scheme objects are the ones you write literal data in program, so this is the default way of writing Scheme objects out.
- Display-family procedures: `display`, `print`, `newline`. These are to produce plain-text output suitable for human readers.

High-level formatting output

To produce output in specific width, alignment, etc: `format`. This corresponds to C's `printf`.

[1] In a sense, this is somewhat similar to what is called "serialization" or "marshalling" in other programming language; you can `write` out a generic Scheme object on disk or to wire, and `read` it to get an object equivalent to the original one. In Lisp-family languages, this is called *read/write invariance* and is a built-in feature. Note that some objects do not have this invariance in nature, so sometimes you need to make your own serializer/marshaller.

Low-level type-specific output
Procedures that deal with raw data.

- To output a character or a byte: `write-char`, `write-byte`.
- To output a string or an array of binary data: `write-string`, `write-uvector`.
- To flush the output buffer: `flush`, `flush-all-ports`.

6.22.8.2 Output controls

`<write-controls>` [Class]

You can control several aspects of Lisp structure output via `<write-controls>` object. The object output routines (e.g. `write, display`) and the high-level output routines (e.g. `format`) can take optional write-controls.

The following example may give you some ideas on what write controls can do:

```
(write '(1 10 100 1000)
       (make-write-controls :base 16 :radix #t))
 prints (#x1 #xa #x64 #x3e8)

(write (iota 100)
       (make-write-controls :length 5))
 prints (0 1 2 3 4 ...)
```

The `make-write-controls` procedure returns a write-controls object, which has the following slots (those slot names are taken from Common Lisp's print control variables):

`length` [Instance Variable of `<write-controls>`]

If this slot has a nonnegative integer, it determines the maximum number of items displayed for lists and vectors (including uniform vectors). If the sequence has more elements than the limit, `...` is printed in place. If this slot is `#f` (default), sequence will be written out fully.

`level` [Instance Variable of `<write-controls>`]

If this slot has a nonnegative integer, it determines the maximum depth of the structure (lists and vectors) to be displayed. If the structure has deeper node, it will be printed as `#`. If this slot is `#f` (default), no depth limit is enforced.

`base` [Instance Variable of `<write-controls>`]

This slot must have an integer between 2 and 36, inclusive, and specifies the radix used to print exact numbers. The default value is 10.

`radix` [Instance Variable of `<write-controls>`]

This slot must have a boolean value. If it is true, radix prefix is always printed before exact numbers. The default value is `#f`.

`pretty` [Instance Variable of `<write-controls>`]

If this slot has true value, *pretty printing* is used, that is, newlines and indentations are inserted to show nested data structures fit in the specified width of columns.

`width` [Instance Variable of `<write-controls>`]

If this slot has a nonnegative integer, it specifies the display column width used for pretty printing.

A write-controls object is immutable. If you need a controls object with a slight variation of an existing controls object, use `write-controls-copy`.

Note: When we introduced `<write-controls>` object in 0.9.5, we used slot names as `print-length`, `print-pretty` etc., mirroring Common Lisp's special variables. However, the print- part is redundant, as it is a part of a class dedicated to print control. So we changed the slot names as of 0.9.6. The procedures make-write-controls and write-controls-copy accepts both old and new names for the backward compatibility. The old code that directly refers to the slots needs to be rewritten (we think there're a not a lot). We'll drop the old name support in 1.0 release.

make-write-controls *:key length level base radix pretty width* [Function]
Creates and returns a write-controls object.

write-controls-copy *controls :key length level base radix pretty width* [Function]
Returns a copy of another write-controls object *controls*. If keyword arguments are given, those values override the original values.

Note: The high-level output procedures can be recursively called via write-object method. In that case, the write controls of the root output call will be automatically inherited to the recursive output calls to the same port.

6.22.8.3 Object output

For the following procedures, the optional *port* argument must be an output port, and when omitted, the current output port is assumed.

Some procedures take *port/controls* argument, which can be either an output port or <write-controls> object. For example, write takes up to two such optional arguments; that is, you can call it as (write obj), (write obj port), (write obj controls), (write obj port controls) or (write obj controls port). When omitted, the port is assumed to be the current output port, and the controls is assumed to be the default controls.

write *obj :optional port/controls1 port/controls2* [Function]
write-shared *obj :optional port/controls1 port/controls2* [Function]
write-simple *obj :optional port/controls1 port/controls2* [Function]
[R7RS+ write] The write-family procedures are used to write an external representation of Scheme object, which can be read back by read procedure. The three procedures differ in a way to handle shared or circular structures.

Write is circular-safe; that is, it uses datum label notation (#n= and #n#) to show cycles. It does not use datum label notation for non-circular structures that are merely shared (see the second example).

```
(let1 x (list 1)
  (set-cdr! x x)   ; create a cycle
  (write x))
 ⇒ shows #0=(1 . #0#)

(let1 x (list 1)
  (write (list x x)))
 ⇒ shows ((1) (1))
```

Write-shared is also circular-safe, and it also shows shared structures using datum labels. Use this if you need to preserve topology of a graph structure.

```
(let1 x (list 1)
  (write (list x x)))
 ⇒ shows (#0=(1) #0#)
```

Finally, write-simple writes out the object recursively without taking account of shared or circular structures. This is fast, for it doesn't need to scan the structure before actually writing out. However, it won't stop when a circular structure is passed.

When these procedures encounter an object of a user-defined class, they call the generic function `write-object`.

Historical context: `Write` has been in Scheme standards, but handling of circular structures hasn't been specified until R7RS. In fact, until Gauche 0.9.4, `write` diverged for circular structures. SRFI-38 introduced the datum-label notation and `write-with-shared-structure` and `write/ss` procedures to produce such notation, and Gauche supported it. R7RS clarified this issue, and Gauche 0.9.4 followed.

`write-with-shared-structure` *obj :optional port* [Function]
`write/ss` *obj :optional port* [Function]
`write*` *obj :optional port* [Function]

[SRFI-38] These are aliases of `write-shared` above.

Gauche has been used the name `write*` for long, which is taken from STklos. SRFI-38 defines `write-with-shared-structure` and `write/ss`. These names are kept for the backward compatibility. New code should use `write-shared`.

`display` *obj :optional port/controls1 port/controls2* [Function]

[R7RS write] Produces a human-friendly representation of an object *obj* to the output port.

If *obj* contains cycles, `display` uses datum-label notation.

When `display` encounters an object of a user-defined class, it calls the generic function `write-object`.

```
(display "\"Mahalo\", he said.")
 ⇒ shows "Mahalo", he said.

(let ((x (list "imua")))
  (set-cdr! x x)
  (display x))
 ⇒ shows #0=(imua . #0#)
```

`print` *expr . . .* [Function]

Displays *exprs* (using `display`) to the current output port, then writes a newline.

`pprint` *obj :key port controls width length level newline* [Function]

Pretty prints *obj* to *port*, which is defaulted to the current output port. The same effect is achieved by passing the `write` procedure a write control with `pretty` slot setting to `#t` (in fact, it is how `pprint` is implemented), but this procedure provides more convenient interface when you want to play with the pretty printer.

By default, `pprint` prints a newline after writing *obj*. You can suppress this newline by passing `#f` to *newline* keyword argument.

To customize pretty printing, you can pass a write control object to the *controls* keyword argument (the `pretty` slot of *controls* is ignored; it'll always printed prettily). Furthermore, you can override `width`, `length` and `level` slots of *controls*. If you omit *controls*, a reasonable default value is assumed. See Section 6.22.8.2 [Output controls], page 232, for the detail of write controls.

```
(pprint (make-list 6 '(gauche droite)))
 ⇒ prints
  ((gauche droite) (gauche droite) (gauche droite) (gauche droite)
   (gauche droite) (gauche droite))

(pprint (make-list 6 '(gauche droite)) :width 20)
 ⇒ prints
```

```
((gauche droite)
 (gauche droite)
 (gauche droite)
 (gauche droite)
 (gauche droite)
 (gauche droite))

(pprint (make-list 6 '(gauche droite)) :length 3)
 ⇒ prints
  ((gauche droite) (gauche droite) (gauche droite) ....)

(pprint (make-list 6 '(gauche droite)) :level 1)
 ⇒ prints
  (# # # # # #)
```

`write-object` (*obj* <*object*>) *port* [Method]

You can customize how the object is printed out by this method.

`newline` *:optional port* [Function]

[R7RS base] Writes a newline character to *port*. This is equivalent to `(write-char #\newline port)`, `(display "\n" port)`. It is kept for a historical reason.

6.22.8.4 Formatting output

`format` *dest controls string arg . . .* [Function]
`format` *controls dest string arg . . .* [Function]
`format` *dest string arg . . .* [Function]
`format` *controls string arg . . .* [Function]
`format` *string arg . . .* [Function]

[SRFI-28+] Format *arg* . . . according to *string*. This function is a subset of CommonLisp's `format` function, with a bit of extension. It is also a superset of SRFI-28, Basic format strings ([SRFI-28], page 830).

The *dest* argument specifies the destination; if it is an output port, the formatted result is written to it; if it is `#t`, the result is written to the current output port; if it is `#f`, the formatted result is returned as a string. *Dest* can be omitted, as SRFI-28 `format`; it has the same effects as giving `#f` to the *dest*.

The *controls* argument is `<write-controls>` object (see Section 6.22.8.2 [Output controls], page 232), which affects the output of `~s` and `~a`. This is Gauche's extension.

(The unusual function signature of `format` is for the convenience; both *dest* and *controls* are optional and they can appear in either order.)

string is a string that contains format directives. A format directive is a character sequence begins with tilde, '`~`', and ends with some specific characters. A format directive takes the corresponding *arg* and formats it. The rest of string is copied to the output as is.

```
(format #f "the answer is ~s" 42)
  ⇒ "the answer is 42"
```

The format directive can take one or more *parameters*, separated by comma characters. A parameter may be an integer or a character; if it is a character, it should be preceded by a quote character. Parameter can be omitted, in such case the system default value is used. The interpretation of the parameters depends on the format directive.

Furthermore, a format directive can take two additional flags: atmark '`@`' and colon '`:`'. One or both of them may modify the behavior of the format directive. Those flags must be placed immediately before the directive character.

If a character 'v' or 'V' is in the place of the parameter, the value of the parameter is taken from the format's argument. The argument must be either an integer, a character, or #f (indicating that the parameter is effectively omitted).

Some examples:

`~10,2s` A format directive `~s`, with two parameters, 10 and 2.

`~12,,,'*A`
: A format directive `~a`, with 12 for the first parameter and a character '*' for the fourth parameter. The second and third parameters are omitted.

`~10@d` A format directive `~d`, with 10 for the first parameter and '@' flag.

`~v,vx` A format directive `~x`, whose first and second parameter will be taken from the arguments.

The following is a complete list of the supported format directives. Either upper case or lower case character can be used for the format directive; usually they have no distinction, except noted.

~A Parameters: *mincol*,*colinc*,*minpad*,*padchar*,*maxcol*

Ascii output. The corresponding argument is printed by `display`. If an integer *mincol* is given, it specifies the minimum number of characters to be output; if the formatted result is shorter than *mincol*, a whitespace is padded to the right (i.e. the result is left justified).

The *colinc*, *minpad* and *padchar* parameters control, if given, further padding. A character *padchar* replaces the padding character for the whitespace. If an integer *minpad* is given and greater than 0, at least *minpad* padding character is used, regardless of the resulting width. If an integer *colinc* is given, the padding character is added (after *minpad*) in chunk of *colinc* characters, until the entire width exceeds *mincol*.

If atmark-flag is given, the format result is right justified, i.e. padding is added to the left.

The *maxcol* parameter, if given, limits the maximum number of characters to be written. If the length of formatted string exceeds *maxcol*, only *maxcol* characters are written. If colon-flag is given as well and the length of formatted string exceeds *maxcol*, *maxcol* - 4 characters are written and a string " ..." is attached after it.

```
(format #f "|~a|" "oops")
  ⇒ "|oops|"
(format #f "|~10a|" "oops")
  ⇒ "|oops      |"
(format #f "|~10@a|" "oops")
  ⇒ "|      oops|"
(format #f "|~10,,,'*@a|" "oops")
  ⇒ "|******oops|"

(format #f "|~,,,,10a|" '(abc def ghi jkl))
  ⇒ "|(abc def gh|"
(format #f "|~,,,,10:a|" '(abc def ghi jkl))
  ⇒ "|(abc de ...|"
```

~S Parameters: *mincol*,*colinc*,*minpad*,*padchar*,*maxcol*

S-expression output. The corresponding argument is printed by `write`. The semantics of parameters and flags are the same as `~A` directive.

```
(format #f "|~s|" "oops")
  ⇒ "|\"oops\"|"
(format #f "|~10s|" "oops")
  ⇒ "|\"oops\"    |"
(format #f "|~10@s|" "oops")
  ⇒ "|    \"oops\"|"
(format #f "|~10,,,'*@s|" "oops")
  ⇒ "|****\"oops\"|"
```

~C Parameters: None

Character output. The argument must be a character, or an error is signaled. If no flags are given, the character is printed with `display`. If atmark-flag is given, the character is printed with `write`.

~D Parameters: *mincol,padchar,commachar,interval*

Decimal output. The argument is formatted as an decimal integer. If the argument is not an integer, all parameters are ignored (after processing '`v`' parameters) and it is formatted by `~A` directive.

If an integer parameter *mincol* is given, it specifies minimum width of the formatted result; if the result is shorter than it, *padchar* is padded on the left (i.e. the result is right justified). The default of *padchar* is a whitespace.

```
(format #f "|~d|" 12345)
  ⇒ "|12345|"
(format #f "|~10d|" 12345)
  ⇒ "|     12345|"
(format #f "|~10,'0d|" 12345)
  ⇒ "|0000012345|"
```

If atmark-flag is given, the sign '`+`' is printed for the positive argument.

If colon-flag is given, every *interval*-th digit of the result is grouped and *commachar* is inserted between them. The default of *commachar* is '`,`', and the default of *interval* is 3.

```
(format #f "|~:d|" 12345)
  ⇒ "|12,345|"
(format #f "|~,,'_,4:d|" -12345678)
  ⇒ "|-1234_5678|"
```

~B Parameters: *mincol,padchar,commachar,interval*

Binary output. The argument is formatted as a binary integer. The semantics of parameters and flags are the same as the `~D` directive.

~O Parameters: *mincol,padchar,commachar,interval*

Octal output. The argument is formatted as an octal integer. The semantics of parameters and flags are the same as the `~D` directive.

~X
~x Parameters: *mincol,padchar,commachar,interval*

Hexadecimal output. The argument is formatted as a hexadecimal integer. If '`X`' is used, upper case alphabets are used for the digits larger than 10. If '`x`' is used, lower case alphabets are used. The semantics of parameters and flags are the same as the `~D` directive.

```
(format #f "~8,'0x" 259847592)
```

```
  ⇒ "0f7cf5a8"
(format #f "~8,'0X" 259847592)
  ⇒ "0F7CF5A8"
```

~F Parameters: *width*,*digis*,*scale*,*ovfchar*,*padchar*

Floating-number output. If the argument is a real number, it is formatted as a decimal floating number. The *width* parameter defines the width of the field; the number is written out right-justified, with the left room padded with *padchar*, whose default is `#\space`. When the formatted output can't fit in *width*, *ovfchar* is output *width* times if it is given, or the entire output is shown if *ovfchar* is omitted.

```
(format "~6f" 3.14)           ⇒ "  3.14"
(format "~6f" 3.141592)       ⇒ "3.141592"
(format "~6,,,'#f" 3.141592) ⇒ "######"
(format "~6,,,,'*f" 3.14)     ⇒ "**3.14"
```

The *digits* parameter specifies number of digits shown below the decimal point. Must be nonnegative integer. When omitted, enough digits to identify the flonum uniquely is generated (same as using `write` and `display`—when you read back the number, you'll get exactly the same flonum.)

```
(format "~6,3f" 3.141592)     ⇒ " 3.142"
(format "~6,0f" 3.141592)     ⇒ "    3."
(format "~10,4f" 355/113)     ⇒ "    3.1416"
(format "~10,4f" 3)           ⇒ "    3.0000"
```

If the *scale* parameter is given, the argument is multiplied by `(expt 10 scale)` before printing.

If the `@` flag is given, plus sign is printed before the non-negative number.

```
(format "~8,3@f" 3.141592)    ⇒ "  +3.142"
```

When *digits* is smaller than the digits required to represent the flonum unambiguously, we round at *digits*+1 position. By default, it is done based on the value the flonum represents—that is, we choose the rounded value closer to the actual value of the flonum. It can sometimes lead to unintuitive results, however. Suppose you want to round 1.15 at 100ths (that is, round to nearest 10ths). Unlike elementary math class, it gives you 1.1. That's because the flonum represented by 1.15 is actually tiny bit smaller than 1.15, so it's closer to 1.1 than 1.2. We show it as 1.15 since no other flonums are closer to 1.15.

But in casual applications, users may perplexed with this behavior. So we support another rounding mode, which we call notational rounding. It is based on the notation used for the flonum. In that mode, rounding 1.15 to nearest 10ths yields 1.2. You can get it by adding `:` flag.

```
(format "~6,1f" 1.15)  ⇒ "   1.1"
(format "~6,1:f" 1.15) ⇒ "   1.2"
```

~* Parameter: *count*

Moves the argument counter *count* times forward, effectively skips next *count* arguments. The default value of *count* is 1, hence skip the next argument. If a colon-flag is given, moves the argument counter backwards, e.g. `~:*` makes the next directive to process last argument again. If an atmark-flag is given, *count* specifies absolute position of the arguments, starting from 0.

~~ Output a single tilda `~`.

~% Output a newline character.

6.22.8.5 Low-level output

`write-char` *char :optional port* [Function]

[R7RS base] Write a single character *char* to the output port *port*.

`write-byte` *byte :optional port* [Function]

Write a byte *byte* to the port. *byte* must be an exact integer in range between 0 and 255.

This procedure is called `write-u8` in R7RS.

`write-string` *string :optional oport start end* [Function]

[R7RS base] If the optional *start* and *end* arguments are omitted, it is the same as `(display string oport)`. The optional arguments restricts the range of *string* to be written.

`flush` *:optional port* [Function]
`flush-all-ports` [Function]

Output the buffered data in *port*, or all ports, respectively.

The function "flush" is called in variety of ways on the various Scheme implementations: `force-output` (Scsh, SCM), `flush-output` (Gambit), or `flush-output-port` (Bigloo). The name `flush` is taken from STk and STklos. R7RS calls this `flush-output-port`

6.23 Loading Programs

6.23.1 Loading Scheme file

`load` *file :key paths (error-if-not-found #t) environment ignore-coding* [Function]

[R7RS+] Loads *file*, that is, read Scheme expressions in *file* and evaluates them. An extension "`.scm`" may be omitted from *file*.

If *file* doesn't begin with "/" or "./" or "../", it is searched from the system file search list, stored in a variable `*load-path*`. Or you can explicitly specify the search path by passing a list of directory names to the keyword argument *paths*.

On success, `load` returns `#t`. If the specified file is not found, an error is signaled unless the keyword argument *error-if-not-found* is `#f`, in which case `load` returns `#f`.

By default, `load` uses a coding-aware port (see Section 6.22.6 [Coding-aware ports], page 226) so that the `"coding:"` magic comment at the beginning of the source file is effective. (See Section 2.3 [Multibyte scripts], page 11, for the details of the coding magic comment). If a true value is given to the keyword argument *ignore-coding*, `load` doesn't create the coding-aware port and directly reads from the file port.

If a module is given to the keyword argument *environment*, `load` works as if the given module is selected at the beginning of the loaded file.

The current module is preserved; even `select-module` is called in *file*, the module in which `load` is called is restored afterwards.

Gauche's `load` is upper-compatible to R5RS `load`, but R7RS `load` differs in optional arguments; see Section 10.2.11 [R7RS load], page 510.

If you want to load a library file, it's better to use '`use`' (see Section 4.13.3 [Defining and selecting modules], page 73), or '`require`' described below. See Section 2.7 [Compilation], page 14, for difference between `load` and `require`.

`*load-path*` [Variable]

Keeps a list of directories that are searched by `load` and `require`.

If you want to add other directories to the search path, do not modify this variable directly; use `add-load-path`, described below, instead.

`add-load-path` *path flag . . .* [Special Form]

Adds a path *path* to the library load path list. *Path* must be a literal string, for load paths must be known at compilation time. If *path* is a relative path, it is resolved relative to the current working directory, unless `:relative` flag is given.

Path doesn't need to exist; nonexisting paths in load path list are simply ignored. However, if *path* does exist, `add-load-path` searches for architecture-dependent paths; see below.

Each *flag* argument may be one of the followings.

`:after` Append *path* to the end of the current list of load paths. By default, *path* is added in front of the load path list.

`#t` The same as `:after`. This is for the backward compatibility.

`:relative`
Interpret *path* as a relative path to the directory of the current file, instead of the current working directory. If the current file can't be determined (e.g. evaluated in REPL, or the expression is read from a socket), this flag is ignored.

Use this form instead of changing `*load-path*` directly. This form is a special form and recognized by the compiler; if you change `*load-path*`, it is in effect at run time, and that may be too late for "use" or "require".

Furthermore, `add-load-path` looks for the architecture dependent directories under the specified path and if it exists, sets up the internal path list for dynamic loading correctly. Suppose you have your Scheme module in `/home/yours/lib`, and that requires a dynamic loadable library. You can put the library under `/home/yours/lib/`*ARCH*`/`, where *ARCH* is the value `(gauche-architecture)` returns (see Section 6.25.3 [Environment inquiry], page 248). Then you can have compiled libraries for multiple platforms and Gauche can still find the right library.

`load-from-port` *port* [Function]

Reads Scheme expressions from an input port *port* and evaluates them, until EOF is read.

Note that unless you pass a coding-aware port to *port*, the `"coding:"` magic comment won't be handled.

`current-load-port` [Function]
`current-load-path` [Function]
`current-load-history` [Function]
`current-load-next` [Function]

These procedures allows you to query the current context of loading. They returns the following values when called inside a file being loaded:

`current-load-port`
Returns the port object from which this form is being loaded.

`current-load-path`
Returns the pathname of the file from which this form is being loaded. Note that this may return `#f` if the source of `load` is not a file.

`current-load-history`
Returns a list of pairs of a port and a line number (integer), representing the nesting of loads. Suppose you load `foo.scm`, and from its line 7 it loads `bar.scm`, and from its line 18 it loads `baz.scm`. If you call `current-load-history` in the file `baz.scm`, you'll get

```
((#<port "foo.scm"> . 7) (#<port "bar.scm"> . 18))
```

`current-load-next`

Returns a list of remaining directories to be searched at the time this file is found. Suppose the `*load-path*` is `("." "../lib" "/home/gauche/lib" "/share/gauche/lib")` and you load `foo.scm`, which happens to be in `../lib/`. Then, inside `foo.scm`, `current-load-next` returns:

```
("/home/gauche/lib" "/share/gauche/lib")
```

When called outside of `load`, these procedures returns `#f`, `#f`, `()` and `()`, respectively.

6.23.2 Load dynamic library

`dynamic-load` *file :key init-function* [Function]

Loads and links a dynamic loadable library (shared library) *file*. *File* shouldn't contain the suffix (".so" on most systems); `dynamic-load` adds it, for it may differ among platforms.

The keyword argument *init-function* specifies the initialization function name of the library in a string. By default, if the file basename (without extension) is "foo", the initialization function name is "Scm_Init_foo".

Usually a dynamic loadable library is provided with wrapping Scheme module, so the user doesn't have to call this function directly.

There's no way to unload the loaded libraries.

6.23.3 Require and provide

`Require` and `provide` are a traditional Lisp way to ensure loading a library file only once. If you require a *feature* for the first time, a library file that provides it is loaded and the fact that the *feature* is provided is memorized. Subsequent request of the same feature doesn't need to load the file.

In Gauche, the `use` syntax (see Section 4.13.4 [Using modules], page 73) hides the require mechanism under the hood so you hardly need to see these forms. These are provided just in case if you want to do some non-trivial management of libraries and thus want to bypass Gauche's standard mechanism.

`require` *feature* [Special Form]

If *feature* is not loaded, load it. *Feature* must be a string, and it is taken as a file name (without suffix) to be loaded. This loading takes place at compile time.

If you load SLIB module, `require` is extended. see Section 12.45 [SLIB], page 746, for details.

If the loaded file does not contain `provide` form at all, the *feature* is automatically provided, as if `(provide feature)` is called at the end of the loaded file. We call this *autoprovide* feature.

Note that `require` first sets the current module to an immutable module called `gauche.require-base` and then load the file. The files loaded by `require` usually have `define-module`/`select-module` or `define-library` for the first thing, so you rarely notice the `gauche.require-base` module. However, if the loaded file has toplevel defines or imports (`use`'s) without specifying a module, you'll get an error like the following:

```
*** ERROR: Attempted to create a binding (a) in a sealed
module: #<module gauche.require-base>
```

Rationale: Generally it's difficult to guarantee when the specified file is loaded by `require` (because some other module may already have required it). If we just used the caller's current module, there would be a couple of issues: The form `define-module` or `define-library` may not be visible from the current module, and you can't guarantee if the toplevel defines without specifying modules in the loaded file inserts the caller's current module, since they may have been loaded into a different module. It is just a bad idea to insert toplevel definitions or to import other modules without specifying which module you put them in. So we made them an error.

`provide` *feature* [Function]

Adds *feature* to the system's provided feature list, so that the subsequent `require` won't load the same file again.

Because of the autoproviding, i.e. `require` automatically provides the required feature, you hardly need to use a `provide` form explicitly. There are a couple of scenarios that you may want to use a `provide` form:

- To provide a feature (or features) that is/are different from the one that caused loading the file.

  Suppose feature X supersedes feature Y and providing compatible APIs of Y but with different implementation. Once `X.scm` is loaded, you don't want `Y.scm` to be loaded; so you want to tell the user that `X.scm` also provides the feature Y. Adding `(provide "X")` and `(provide "Y")` at the end of `X.scm` accomplish that. (Note: If you add a provide form, `require` no longer autoprovides the feature, so you need to specify `(provide "X")` in `X.scm` explicitly to provide X as well.)

  Of course, this doesn't prevent users from loading `Y.scm` by specifying `(require "Y")` before `(require "X")`. It should be considered just as a workaround in a production where other solutions are costly, instead of a permanent solution.

- To provide no features at all. Passing `#f` as *feature* prevents autoproviding by `require` without providing any feature.

  This should also be a temporary solution. One possible scenario is that you are changing `X.scm` very frequently during development and you want `(require "X")` always causes loading the file. Don't forget to remove `(provide #f)` when you release the file, though. Besides, for interactive reloading, consider using `gauche.reload` (see Section 9.27 [Reloading modules], page 439) instead.

`provided?` *feature* [Function]

Returns `#t` if *feature* is already provided.

6.23.4 Autoload

`autoload` *file/module item . . .* [Macro]

Sets up *item* . . . to be autoloaded. That is, when an *item* is referenced for the first time, *file/module* is loaded before the *item* is evaluated. This delays the loading of *file/module* until it is needed.

You can specify either a string file name or a symbol module name to *file/module*. If it is a string, the named file is loaded. If it is a symbol, the named module is loaded (using the same rule as of `use`), then the binding of *item* in the *file/module* is imported to the module used the autoload (See Section 4.13.3 [Defining and selecting modules], page 73, for details of `use`).

Item can be either a variable name (symbol), or a form `(:macro` *symbol*`)`. If it is a variable, the named file/module is loaded when the variable is about to be evaluated. If it is the latter form, the named file/module is loaded when a form `(`*symbol* `arg ...)` is about to be *compiled*, which enables autoloading macros.

file/module must define *symbol* in it, or an error is signaled when *file/module* is autoloaded.

The following is an example of autoloading procedures.

```
(autoload "foo" foo0 foo1)
(autoload "bar" bar0 bar1)
```

```
(define (foobar x)
  (if (list? x)
      (map bar0 x)
      (foo0)))

(foobar '(1 2)) ; "bar" is loaded at this moment

(foobar #f)     ; "foo" is loaded at this moment
```

Note that if you set to autoload macro, the file/module is loaded immediately when such form that uses the macro is compiled, regardless of the piece of the code is executed or not.

6.23.5 Operations on libraries

There are several procedures you can use to check if certain libraries and/or modules are installed in the system.

In the following descriptions, *pattern* is either a symbol or a string. If it is a symbol, it specifies a module name (e.g. `foo.bar`). If it is a string, it specifies a partial pathname of the library (e.g. `"foo/bar"`), which will be searched under library search paths. You can also use glob-like metacharacters `*` and `?` in *pattern*.

`library-fold` *pattern proc seed :key paths strict? allow-duplicates?* [Function]

A basic iterator for library/module files. This procedure searches Scheme program files which matches *pattern*, under directories listed in *paths* (the default is the standard file load paths, `*load-path*`). For each matched file, it calls *proc* with three arguments: the matched module or library name, the full path of the program file, and the state value. *Seed* is used as the initial state value, and the value *proc* returns is used as the state value for the next call of *proc*. The value returned from the last *proc* becomes the return value of `library-fold`.

If *pattern* is a symbol and the keyword argument *strict?* is `#t` (which is the default), this procedure calls `library-has-module?` on the files whose name seems to match the given pattern of module name, in order to find out the file really implements the module. It can be a time consuming process if you try to match large number of modules; you can pass `#f` to *strict?* to avoid the extra check. If *pattern* is a string, matching is done only for file names so *strict?* is ignored.

By default, if there are more than one files that have the same name that matches *pattern* in *paths*, only the first one appears in *paths* is taken. This gives you the file you'll get if you use `require` or `use` for that library. If you want to iterate all of matching files, pass `#t` to the *allow-duplicates?* keyword argument.

Here are some examples (the result may differ in your environment).

```
(library-fold 'srfi-1 acons '())
 ⇒ ((srfi-1 . "../lib/srfi-1.scm"))

(library-fold "srfi-1" acons '())
 ⇒ (("srfi-1" . "../lib/srfi-1.scm"))

;; Note the returned list is in a reverse order of
;; how acons is called.
(library-fold 'srfi-1 acons '() :allow-duplicates? #t)
 ⇒ ((srfi-1 . "/usr/share/gauche/0.7.1/lib/srfi-1.scm")
   (srfi-1 . "../lib/srfi-1.scm"))

;; In the following cases, the module name doesn't match,
```

```
;; but the filename does.
(library-fold 'srfi-19.* acons '())
 ⇒ ()

(library-fold "srfi-19/*" acons '())
 ⇒ (("srfi-19/read-tai" . "../lib/srfi-19/read-tai.scm")
   ("srfi-19/format" . "../lib/srfi-19/format.scm"))

;; Finds available dbm implementations
(library-fold 'dbm.* acons '())
 ⇒ ((dbm.cdb . "/usr/share/gauche/0.7.1/lib/dbm/cdb.scm")
   (dbm.gdbm . "../lib/dbm/gdbm.scm")
   (dbm.ndbm . "../lib/dbm/ndbm.scm")
   (dbm.odbm . "../lib/dbm/odbm.scm"))
```

`library-map` *pattern proc :key paths allow-duplicates? strict?* [Function]
`library-for-each` *pattern proc :key paths allow-duplicates? strict?* [Function]

`Map` and `for-each` version of iterator over matched libraries/modules. See `library-fold` above for detailed operation of matching and the meanings of keyword arguments.

Proc receives two arguments, the matched module/library name and full path of the file. `Library-map` returns a list of results of *proc*. `Library-for-each` discards the results.

```
(library-map 'srfi-4 list :allow-duplicates? #t)
 ⇒ ((srfi-4 "../lib/srfi-4.scm")
            (srfi-4 "/usr/share/gauche/0.7.1/lib/srfi-4.scm"))

(library-map 'dbm.* (lambda (m p) m))
 ⇒ (dbm.odbm dbm.ndbm dbm.gdbm dbm.cdb)
```

`library-exists?` *mod/path :key paths force-search? strict?* [Function]

Search a library or a module specified by *mod/path*, and returns a true value if it finds one. *Paths* and *strict?* keyword arguments have the same meaning as `library-fold`.

Unlike the iterator procedures above, this procedure first checks loaded libraries and modules in the calling process, and returns true if it finds *mod/path* in it, without looking into the filesystem. Passing `#t` to *force-search?* keyword arguments skips the checking of loaded libraries and modules.

`library-has-module?` *path module* [Function]

Returns `#t` iff a file specified by *path* exists and appears to implement a module named by `module`. *path* must be an actual filename.

```
(library-has-module? "./test/foo/bar.scm" 'foo.bar)
 ⇒ #t ;; if ./test/foo/bar.scm implements module foo.bar.
```

This procedure assumes a typical layout of the source code to determine if the given file implements the module, i.e., it reads the first form of the code and see if it is a `define-module` form that is defining the given module.

6.24 Sorting and merging

The interface of sorting and merging API complies SRFI-95, with the following extensions:

- You can sort not only lists, vectors and strings, but any sequence (an instance of `<sequence>`).
- You can use both comparison procedures and comparators (see Section 6.2.4 [Basic comparators], page 103) to specify the order.

- You can omit comparison procedure; in that case, elements are compared with `default-comparator`.

`sort` *seq :optional cmp keyfn* [Function]
`sort!` *seq :optional cmp keyfn* [Function]

[SRFI-95+] Sorts elements in a sequence *seq* in ascending order and returns the sorted sequence. `sort!` destructively reuses the original sequence.

You can pass an instance of any `<sequence>` as *seq*; the same type of sequence will be returned. For `sort`, the sequence type must have builder interface so that `sort` can build a new sequence of the same type (See Section 9.5.4 [Fundamental iterator creators], page 349, for the builder interface). For `sort!`, *seq* must be mutable.

The sorting order is specified by `cmp`. It must be either a procedure or a comparator. If it is a procedure, it must take two elements of *seq*, and returns `#t` if the first argument strictly precedes the second. If it is a comparator, it must have the comparison procedure. If omitted, `default-comparator` is used.

If the optional argument *keyfn* is given, the elements are first passed to it and the results are used for comparison. It is guaranteed that *keyfn* is called at most once per element.

```
(sort '(("Chopin" "Frederic")
        ("Liszt" "Franz")
        ("Alkan" "Charles-Valentin"))
      string<?
      car)
  ⇒ (("Alkan" "Charles-Valentin")
      ("Chopin" "Frederic")
      ("Liszt" "Franz"))
```

In the current implementation, quicksort and heapsort algorithm is used when both *cmp* and *keyfn* is omitted, and merge sort algorithm is used otherwise. That is, the sort is stable if you pass at least *cmp* (note that to guarantee stability, *cmp* must return `#f` when given identical arguments.) SRFI-95 requires stability, but also requires *cmp* argument, so those procedures are upper-compatible to SRFI-95.

If you want to keep a sorted set of objects to which you add objects one at at time, you can also use treemaps (see Section 6.16 [Treemaps], page 182). If you only need to find out a few maximum or minimum elements instead of sorting all the elements, heaps can be used (see Section 12.9 [Heap], page 652).

`sorted?` *seq :optional cmp keyfn* [Function]

[SRFI-95+] Returns `#t` iff elements in *seq* are in sorted order. You can pass any sequence to *seq*. The optional argument *cmp* and *keyfn* are the same as `sort`.

In SRFI-95, *cmp* can't be omitted.

`merge` *a b :optional cmp keyfn* [Function]
`merge!` *a b :optional cmp keyfn* [Function]

[SRFI-95+] Arguments *a* and *b* are lists, and their elements are sorted using a compare function or a comparator *cmp*. These procedures merges two list and returns a list, whose elements are sorted using *cmp*. The destructive version `merge!` reuses cells in *a* and *b*; the returned list is `eq?` to either *a* or *b*.

In SRFI-95, *cmp* can't be omitted.

The following procedures are for the backward compatibility. Their features are already covered by extended `sort` and `sort!`.

`stable-sort` *seq :optional cmp keyfn* [Function]
`stable-sort!` *seq :optional cmp keyfn* [Function]

Sort a sequence *seq*, using stable sort algorithm. Arguments *cmp* and *keyfn* are the same as `sort` and `sort!`.

In fact, `sort` and `sort!` now uses stable algorithm when *cmp* is provided, so these procedures are redundant, unless you want to omit *cmp* and yet guarantee stable sort.

`sort-by` *seq keyfn :optional cmp* [Function]
`sort-by!` *seq keyfn :optional cmp* [Function]
`stable-sort-by` *seq keyfn :optional cmp* [Function]
`stable-sort-by!` *seq keyfn :optional cmp* [Function]

Variations of sort procedures that takes a key extracting function. These are redundant now, for `sort` etc. takes optional *keyfn*.

6.25 System interface

Gauche supports most of POSIX.1 functions and other system functions popular among Unix variants as built-in procedures.

Lots of Scheme implementations provide some sort of system interface under various APIs. Some are just called by different names (e.g, `delete-file` or `remove-file` or `unlink` to delete a file), some do more abstraction introducing new Scheme objects. Instead of just picking one of such interfaces, I decided to implement Gauche's system interface API in two layers; the lower level layer, described in this section, follows the operating system's API as close as possible. On top of that, the higher-level APIs are provided, with considering compatibility to the existing systems.

The low level system interface has the name `sys-`*name* and usually correspond to the system call *name*. I tried to keep the interface similar whenever reasonable.

Gauche restarts a system call after it is interrupted by a signal by default. See Section 6.25.7 [Signal], page 260, for the details.

If you are familiar with system programming in C, see also Appendix B [C to Scheme mapping], page 832, which shows correspondence between C standard library functions and Gauche procedures.

6.25.1 Program termination

Gauche has a few ways to terminate itself (other than returning from `main`). The `exit` procedure is a graceful way with all proper cleanups. `sys-exit` and `sys-abort` may be used in emergency where proper cleanup is impossible.

`exit` *:optional* (*code 0*) (*fmtstr #f*) *args . . .* [Function]

[R7RS+] Terminates the current process with the exit code *code*. *Code* must be zero or positive exact integer. When a string is given to *fmtstr*, it is passed to `format` (see Section 6.22.8 [Output], page 231), with the rest arguments *args*, to produce a message to the standard error port (*not* the current error port; see Section 6.22.3 [Common port operations], page 218).

In fact, the exiting procedure is a bit more complicated. The precise steps of exiting is as follow.

1. The value of parameter `exit-handler` is checked. If it is not `#f`, the value is called as a procedure with three arguments: *code*, *fmtstr*, and a list of rest arguments. It is the default procedure of `exit-handler` that prints out the message to the standard error port. If an error occurs within exit handler, it is captured and discarded. Other exceptions are not caught.

2. The *after* thunks of the active dynamic winds are invoked. Any exceptions raised in *after* thunks are captured and discarded.
3. The clean-up handlers registered via C API `Scm_AddCleanupHandler` are invoked. These are usually responsible for under-the-hood cleanup jobs for each application that embeds Gauche. From the Scheme world there's not much to care.
4. The unclosed output buffered ports are flushed.
5. The process exits with *code* as an exit code, via `exit(3)`.

The `exit-handler` mechanism allows the application to hook its exit operation. Note that it is not for simple cleanup jobs; `dynamic-wind`, `guard` or `unwind-protect` are more appropriate. `exit-handler` is for more specific use just upon application exit. For example, GUI applications may want to post a dialog instead of printing to stderr.

For this reason, the library code shouldn't change `exit-handler`; only the application knows what to do when it exits.

Another useful case is when you want to call a third-party code which calls `exit` inside. In that case you may swap the `exit-handler` for the one that raises a non-error exception while calling the third-party code. Non-error exception isn't caught in `exit`, effectively interrupts the steps explained above. (Yet the *after* thunks of dynamic handlers are processed just like normal exception handling case.) Your application code can then capture the exception. You can use `parameterize` to swap `exit-handler` dynamically and thread-safely (see Section 9.22 [Parameters], page 411).

```
(guard (e [(eq? e 'exit-called) (handle-exit-as-desired)])
  (parameterize ((exit-handler (lambda (c f a) (raise 'exit-called))))
    (call-third-party-library)))
```

Generally, calling `exit` while other threads are running should be avoided, since it only rewinds the dynamic handlers active in the calling threads, and other threads will be killed abruptly. If you have to do so for some reason, you may be able to use `exit-handler` to tell to other threads that the application is exiting. (There's no general way, and Gauche doesn't even have a list of all running threads; it's application's responsibility).

Note on design: Some languages integrates exit handling into exception handling, treating exit as a kind of exception. It is a tempting idea, so much that we've tried it. It didn't work out well in Gauche; a big annoyance was that when an *after* thunk raised an exception during rewinding `dynamic-winds`, it shadowed the original *exit* exception.

`exit-handler` *:optional new-handler* [Function]

When called without argument, returns the value of the current exit handler. When called with an argument, sets *new-handler* as the value of the exit handler, and returns the previous value of the exit handler. *new-handler* must be a procedure that takes three arguments, or `#f`.

The value of exit handler is thread-specific, and the default value is inherited from the value of the current exit handler of the parent thread. `exit-handler` can be used as if it's a parameter in the `parameterize` macro (see Section 9.22 [Parameters], page 411).

`sys-exit` *code* [Function]

[POSIX] Terminates the current process with the exit code *code*. *Code* must be zero or positive exact integer. This procedure calls `_exit(2)` directly. No cleanup is done. Unflushed file output is discarded.

`sys-abort` [Function]

[POSIX] Calls POSIX abort(). This usually terminates the running process and dumps core. No cleanup is done.

6.25.2 Command-line arguments

The recommended way to get command-line arguments passed to a Scheme script is the argument to the `main` procedure (see Section 3.3 [Writing Scheme scripts], page 25). For the convenience, there are a few ways to access to the command-line arguments globally.

Note that a Scheme code may not always be called with a command-line argument—for example, an application-embedded Scheme scriptlet may not have the concept of command-line at all. That's why the `main` argument is preferred, since it is an explicit interface; if `main` is called, the caller is responsible to pass in something.

That said, here are how to access the command-line arguments:

`command-line` [Parameter]

[R7RS+] When called without arguments, it returns a list of command-line arguments, including the program name in the first element, as a list of strings.

When Gauche is used as an embedded language, it is application's discretion to set up this parameter. If the application does nothing, this parameter will have an empty list. When you use this parameter in the library you have to deal with that situation.

When called with one argument, a list of string, it will become the new value of the parameter. You can use `parameterize` to switch the value of `command-line` dynamically (see Section 9.22 [Parameters], page 411). Note that R7RS only defines zero-argument `command-line`.

`*program-name*` [Variable]
`*argv*` [Variable]

These variables are bound to the program name and the list of command-line arguments, respectively. In Gauche scripts that are invoked by `gosh` command, `*program-name*` is usually the name of the script, as given to `gosh`. When `gosh` is invoked interactively, `*program-name*` is `gosh` itself.

These variables exist in `user` module.

They are mainly kept for the backward compatibility. These names are compatible to STk, but other Scheme implementation uses different conventions. The `command-line` parameter above is preferred.

When Gauche is used as an embedded language, it's the host application's discretion to set up these variables. Generally, you can't count on those variables to exist. That's another reason you should avoid using them.

6.25.3 Environment inquiry

`sys-getenv` *name* [Function]

[POSIX] Returns the value of the environment variable *name* as a string, or `#f` if the environment variable is not defined.

For the portable code, you may want to use SRFI-98's `get-environment-variable` (see Section 11.19 [Accessing environment variables], page 622), which is also in R7RS.

Note: Most systems doesn't guarantee thread-safety of `getenv` while environment is being modified; however, Gauche mutexes environment accessing/mutating APIs internally, so you don't need to worry about the race condition as far as you use Gauche procedures.

`sys-environ` [Function]

Returns the current environment as a list of strings. Each string is a form of `NAME=VALUE`, where `NAME` is the name of the environment variable and `VALUE` is its value. `NAME` never contains a character `#\=`. This is useful when you want to obtain the all environment variables of the current process. Use `sys-getenv` if you want to query a specific environment variable.

sys-environ->alist *:optional envlist* [Function]

A convenience procedure for `sys-environ`. When the list of environment strings (like what `sys-environ` returns) is given to *envlist*, this procedure splits name and value of each environment variable and returns an assoc list.

When *envlist* is omitted, this procedure calls `sys-environ` to get the current environment variables.

For the portable code, you may want to use SRFI-98's `get-environment-variables` (see Section 11.19 [Accessing environment variables], page 622), which is also in R7RS.

```
(sys-environ->alist '("A=B" "C=D=E"))
  => (("A" . "B") ("C" . "D=E"))
```

sys-setenv *name value :optional overwrite* [Function]
sys-putenv *name=value* [Function]

sys-setenv inserts an environment variable *name* with the value *value*. Both *name* and *value* must be a string. If the optional argument *overwrite* is `#f` (default), the environment is untouched if a variable with *name* already exists. If *overwrite* is true, the variable is overwritten.

For `sys-putenv`, you have to give a single string with the form of `NAME=VALUE`, that is, concatenating *name* and *value* with `#\=`. If the environment variable with the same name exists, it will be overwritten.

These API reflects POSIX `setenv(3)` and `putenv(3)`. However, unlike `putenv(3)`, modifying the string passed to `sys-putenv` afterwards won't affect the environment.

These procedures are only available when a feature identifier `gauche.sys.setenv` exists. Use `cond-expand` (see Section 4.12 [Feature conditional], page 68) to check their availability.

```
(cond-expand
 [gauche.sys.setenv
   ... use sys-setenv or sys-putenv ... ]
 [else
   ... fallback code ...])
```

These procedures are thread-safe as far as you access and modify the environment through Gauche API.

sys-unsetenv *name* [Function]
sys-clearenv [Function]

Remove the environment variable with *name* (`sys-unsetenv`), or all environment variables. `sys-clearenv` is handy when you need to run subprocess, but you cannot trust the inherited environment.

These procedures are only available when a feature identifier `gauche.sys.unsetenv` exists. Use `cond-expand` (see Section 4.12 [Feature conditional], page 68) to check their availability.

```
(cond-expand
 [gauche.sys.unsetenv
   ... use sys-unsetenv or sys-clearenv ... ]
 [else
   ... fallback code ...])
```

SRFI-98 (see Section 11.19 [Accessing environment variables], page 622) also defines a subset of above procedures to access to the environment variables. Portable programs may want to use them instead.

gauche-version [Function]
gauche-architecture [Function]

`gauche-library-directory` [Function]
`gauche-architecture-directory` [Function]
`gauche-site-library-directory` [Function]
`gauche-site-architecture-directory` [Function]

These functions returns a string that tells information about Gauche interpreter itself.

`sys-available-processors` [Function]

Returns the number of available processors on the running platform. Return value is always a positive exact integer. If Gauche can't get the information, 1 is returned.

However, If an environment variable `GAUCHE_AVAILABLE_PROCESSORS` is defined and its value can be interpreted as a positive integer, then the value is returned regardless of what the hardware/OS tells.

6.25.4 Filesystems

System calls that deal with filesystems. See also Section 12.24 [Filesystem utilities], page 692, which defines high-level APIs on top of the procedures described here.

6.25.4.1 Directories

See also Section 12.24.1 [Directory utilities], page 693, for high-level API.

`sys-readdir` *path* [Function]

path must be a string that denotes valid pathname of an existing directory. This function returns a list of strings of the directory entries. The returned list is not sorted. An error is signaled if *path* doesn't exists or is not a directory.

`glob` *pattern :key separator folder sorter* [Function]
`sys-glob` *pattern :key separator folder sorter* [Function]

Provides a traditional Unix glob(3) functionality; returns a list of pathnames that matches the given *pattern*.

This feature used to be a wrapper of system-provided `glob` function, hence it was named `sys-glob`. However, as of Gauche version 0.8.12, it was reimplemented in Scheme on top of other system calls, to overcome incompatibilities between platforms and for the opportunity to put more functionalities. So we renamed it `glob`. The old name `sys-glob` is kept for compatibility, but new programs should use `glob`.

The *pattern* argument may be a single *glob pattern*, or a list of glob patterns. If a list is given, pathnames that matches any one of the pattern are returned. If you're a unix user, you already know how it works.

```
gosh> (glob "*.scm")
("ext.scm" "test.scm")
gosh> (glob "src/*.[ch]")
("src/ext.c" "src/ext.h")
gosh> (glob '("*.scm" "src/*.c"))
("ext.scm" "src/ext.c" "test.scm")
```

Unlike shell's glob, if there's no matching pathnames, `()` is returned.

By default, the result is sorted using built-in `sort` procedure (see Section 6.24 [Sorting and merging], page 244). You can pass alternative procedure to *sorter* argument; it should be a procedure that takes single list, and returns a sorted list. It can also be `#f`, in which case the result isn't sorted at all.

In fact, globbing is a very useful tool to search hierarchical data structure in general, not limited to the filesystems. So the `glob` function is implemented separately from the filesystem.

Using keyword arguments, you can glob from any kind of tree data structure. It is just that their default values are set to look at the filesystems.

The *separator* argument should be a char-set, and used to split the *pattern* into components. Its default is `#[/]`. It is not used to the actual pathnames to match.

The *folder* is a procedure that walks through the data structure. It is called with five arguments:

```
(folder proc seed parent regexp non-leaf?)
```

proc is a procedure that takes two arguments. The *folder* should call *proc* with every node in the *parent* whose component name matches *regexp*, passing around the seed value just like `fold`. It should return the final value returned by *proc*. For example, if `cons` is given to *proc* and `()` is given to *seed*, the return value of the folder is a list of nodes that matches the *regexp*.

The representation of a node is up to the implementation of *folder*. It can be a pathname, or some sort of objects, or anything. The `glob` procedure does not care what it is; the `glob` procedure merely passes the node to subsequent call to `folder` as *parent* argument, or returns a list of nodes as the result.

The *parent* argument is basically a node, and *folder* traverses its children to find the match. The exception is the initial call of *folder*— at the beginning `glob` knows nothing about each node. When `glob` needs to match an absolute path, it passes `#t`, and when `glob` needs to match a relative path, it passes `#f`, as the initial *parent* value.

The *regexp* argument is used to filter the child nodes. It should be matched against the component name of the child, not including its directory names. As a special case, it can be a symbol `dir`; if that's the case, the folder should return *node* itself, but it may indicate *node as a directory*; e.g. if *node* is represented as a pathname, the folder returns a pathname with trailing directory separator. As special cases, if *node* is a boolean value and *regexp* is `dir`, the folder should return the node representing root node or current node, respectively; e.g. if *node* is represented as a pathname, the folder may return `"/"` and `"./"` for those cases.

The *non-leaf* argument is a boolean flag. If it is true, the filter should omit the leaf nodes from the result (e.g. only include the directories).

Now, here's the precise spec of glob pattern matching.

Each glob pattern is a string to match pathname-like strings.

A pathname-like string is a string consists of one or more *components*, separated by *separator*s. The default separator is `#[/]`; you can change it with *separator* keyword argument. A component cannot contain separators, and cannot be a null string. Consecutive separators are regarded as a single separator. A pathname-like string optionally begins with, and/or ends with a separator character.

A glob pattern also consists of components and separator characters. In a component, following characters/syntax have special meanings.

`*`
: When it appears at the beginning of a component, it matches zero or more characters except a period (`.`). And it won't match if the component of the input string begins with a period.

  Otherwise, it matches zero or more sequence of any characters.

`**`
: If a component is just `**`, it matches zero or more number of components that match `*`. For example, `src/**/*.h` matches all of the following patterns.

```
src/*.h
src/*/*.h
src/*/*/*.h
src/*/*/*/*.h
```

...

`?` When it appears at the beginning of a component, it matches a character except a period (`.`). Otherwise, it matches any single character.

`[chars]` Specifies a character set. Matches any one of the set. The syntax of *chars* is the same as Gauche's character set syntax (see Section 6.11 [Character set], page 147). For the compatibility of the traditional glob, the `!` character can be used to complement the character set, e.g. `[!abc]` is the same as `[^abc]`.

`glob-fold` *pattern proc seed :key separator folder sorter* [Function]

This is actually a low-level construct of the glob function. Actually, *glob* is simply written like this:

```
(define (glob patterns . opts)
  (apply glob-fold patterns cons '() opts))
```

The meaning of *pattern*, *separator*, *folder* and *sorter* is the same as explained above.

For each pathname that matches *pattern*, `glob-fold` calls *proc* with the pathname and a seed value. The initial seed value is *seed*, and the value *proc* returns becomes the next seed value. The result of the last call to *proc* becomes the result of `glob-fold`. If there's no matching pathnames, *proc* is never called and *seed* is returned.

`make-glob-fs-fold` *:key root-path current-path* [Function]

This is a utility function to generate a procedure suitable to pass the *folder* keyword argument of `glob-fold` and `glob`. Without arguments, this returns the same procedure which is used in `glob-fold` and `glob` by default.

The keyword arguments *root-path* and *current-path* specify the paths where `glob-fold` starts to search.

```
gosh> (glob "/tmp/*.scm")
("/tmp/x.scm" "/tmp/y.scm")
gosh> (glob "/*.scm"
            :folder (make-glob-fs-fold :root-path "/tmp"))
("/tmp/x.scm" "/tmp/y.scm")
gosh> (glob "*.scm"
            :folder (make-glob-fs-fold :current-path "/tmp"))
("/tmp/x.scm" "/tmp/y.scm")
```

See Section 6.25.4.4 [File stats], page 255, to check if a path is actually a directory.

6.25.4.2 Directory manipulation

`sys-remove` *filename* [Function]

[POSIX] If *filename* is a file it is removed. On some systems this may also work on an empty directory, but portable scripts shouldn't depend on it.

`sys-rename` *old new* [Function]

[POSIX] Renames a file *old* to *new*. The new name can be in different directory from the old name, but both paths must be on the same device.

`sys-tmpnam` [Function]

[POSIX] Creates a file name which is supposedly unique, and returns it. This is in POSIX, but its use is discouraged because of potential security risk. Use `sys-mkstemp` below if possible.

`sys-mkstemp` *template* [Function]

Creates and opens a file that has unique name, and returns two values; opened port and the created filename. The file is created exclusively, avoiding race conditions. *template* is used as the prefix of the file. Unlike Unix's mkstemp, you don't need padding characters. The file is opened for writing, and its permission is set to 600.

`sys-mkdtemp` *template* [Function]
Creates a directory that has unique name, and returns the name. *template* is used as the prefix of the directory. Unlike Unix's mkdtemp, you don't need padding characters. The directory's permission is set to 700.

`sys-link` *existing new* [Function]
[POSIX] Creates a hard link named *new* to the existing file *existing*.

`sys-unlink` *pathname* [Function]
[POSIX] Removes *pathname*. It can't be a directory. Returns `#t` if it is successfully removed, or `#f` if *pathname* doesn't exist. An error is signaled otherwise.

There are similar procedures, `delete-file/remove-file` in `file.util` module, while they raises an error when the named pathname doesn't exist (see Section 12.24.4 [File operations], page 699).

R7RS defines `delete-file`, which you may want to use in portable programs.

`sys-symlink` *existing new* [Function]
Creates a symbolic link named *new* to the pathname *existing*. On systems that doesn't support symbolic links, this function is unbound.

`sys-readlink` *path* [Function]
If a file specified by *path* is a symbolic link, its content is returned. If *path* doesn't exist or is not a symbolic link, an error is signaled. On systems that don't support symbolic links, this function is unbound.

`sys-mkdir` *pathname mode* [Function]
[POSIX] Makes a directory *pathname* with mode *mode*. (Note that *mode* is masked by the current umask; see `sys-umask` below). The parent directory of *pathname* must exist and be writable by the process. To create intermediate directories at once, use `make-directory*` in `file.util` (Section 12.24.1 [Directory utilities], page 693).

`sys-rmdir` *pathname* [Function]
[POSIX] Removes a directory *pathname*. The directory must be empty. To remove a directory with its contents, use `remove-directory*` in `file.util` (Section 12.24.1 [Directory utilities], page 693).

`sys-umask` *:optional mode* [Function]
[POSIX] Sets umask setting to *mode*. Returns previous umask setting. If *mode* is omitted or `#f`, just returns the current umask without changing it. See `man umask` for more details.

6.25.4.3 Pathnames

See also Section 12.24.2 [Pathname utilities], page 696, for high-level APIs.

`sys-normalize-pathname` *pathname :key absolute expand canonicalize* [Function]
Converts *pathname* according to the way specified by keyword arguments. More than one keyword argument can be specified.

`absolute` If this keyword argument is given and true, and *pathname* is not an absolute pathname, it is converted to an absolute pathname by appending the current working directory in front of *pathname*.

`expand` If this keyword argument is given and true, and *pathname* begins with '~', it is expanded as follows:

- If *pathname* is consisted entirely by "~", or begins with "~/", then the character "~" is replaced for the pathname of the current user's home directory.
- Otherwise, characters following '~' until either '/' or the end of *pathname* are taken as a user name, and the user's home directory is replaced in place of it. If there's no such user, an error is signaled.

`canonicalize`
Tries to remove pathname components "." and "..". The pathname interpretation is done purely in textural level, i.e. it doesn't access filesystem to see the conversion reflects the real files. It may be a problem if there's a symbolic links to other directory in the path.

`sys-basename` *pathname* [Function]
`sys-dirname` *pathname* [Function]

`sys-basename` returns a basename, that is the last component of *pathname*. `sys-dirname` returns the components of *pathname* but the last one. If *pathname* has a trailing '/', it is simply ignored.

```
(sys-basename "foo/bar/bar.z") ⇒ "bar.z"
(sys-basename "coo.scm") ⇒ "coo.scm"
(sys-basename "x/y/") ⇒ "y"
(sys-dirname "foo/bar/bar.z") ⇒ "foo/bar"
(sys-dirname "coo.scm") ⇒ "."
(sys-dirname "x/y/") ⇒ "x"
```

These functions doesn't check if *pathname* really exists.

Some boundary cases:

```
(sys-basename "") ⇒ ""
(sys-dirname "") ⇒ "."

(sys-basename "/") ⇒ ""
(sys-dirname "/") ⇒ "/"
```

Note: The above behavior is the same as Perl's `basename` and `dirname`. On some systems, the command `basename` may return `"/"` for the argument `"/"`, and `"."` for the argument `"."`.

`sys-realpath` *pathname* [Function]

`sys-realpath` returns an absolute pathname of *pathname* that does not include ".", ".." or symbolic links. If *pathname* does not exist, it includes a dangling symbolic link, or the caller doesn't have enough permission to access to the path, an error is signaled.

Note: the POSIX `realpath(3)` function is known to be unsafe, so Gauche avoids using it and implements `sys-realpath` in its own.

`sys-tmpdir` [Function]

Returns the default directory name suitable to put temporary files.

On Unix-like systems, the environment variable `TMPDIR` and `TMP` are first checked, then falls back to `/tmp`.

On Windows-native systems, it uses `GetTempPath` Windows API. It checks environment variables `TMP`, `TEMP`, and `USERPROFILE` in this order, and falls back to Windows system directory.

On both platforms, the returned pathname may not exist, or may not be writable by the calling process.

In general, user programs and libraries are recommended to use `temporary-directory` (see Section 12.24.1 [Directory utilities], page 693) instead; `sys-tmpdir` should be used only if you now the raw value the platform provides.

6.25.4.4 File stats

See also Section 12.24.3 [File attribute utilities], page 698, for high-level APIs.

`file-exists?` *path* [Function]

[R7RS file] Returns true if *path* exists.

`file-is-regular?` *path* [Function]
`file-is-directory?` *path* [Function]

Returns true if *path* is a regular file, or is a directory, respectively. They return false if *path* doesn't exist at all.

`<sys-stat>` [Builtin Class]

An object that represents `struct stat`, attributes of an entry in the filesystem. It has the following read-only slots.

`type` [Instance Variable of `<sys-stat>`]

A symbol represents the type of the file.

`regular`	a regular file
`directory`	a directory
`character`	a character device
`block`	a block device
`fifo`	a fifo
`symlink`	a symbolic link
`socket`	a socket

If the file type is none of the above, `#f` is returned.

Note: Some operating systems don't have the `socket` file type and returns `fifo` for socket files. Portable programs should check both possibilities to see if the given file is a socket.

`perm` [Instance Variable of `<sys-stat>`]

An exact integer for permission bits of `struct stat`. It is the same as lower 9-bits of "mode" slot; provided for the convenience.

`mode` [Instance Variable of `<sys-stat>`]
`ino` [Instance Variable of `<sys-stat>`]
`dev` [Instance Variable of `<sys-stat>`]
`rdev` [Instance Variable of `<sys-stat>`]
`nlink` [Instance Variable of `<sys-stat>`]
`uid` [Instance Variable of `<sys-stat>`]
`gid` [Instance Variable of `<sys-stat>`]
`size` [Instance Variable of `<sys-stat>`]

An exact integer for those information of `struct stat`.

`atime` [Instance Variable of `<sys-stat>`]
`mtime` [Instance Variable of `<sys-stat>`]
`ctime` [Instance Variable of `<sys-stat>`]

A number of seconds since Unix Epoch for those information of `struct stat`.

`sys-stat` *path* [Function]
`sys-fstat` *port-or-fd* [Function]

[POSIX] Returns a `<sys-stat>` object of *path*, or the underlying file of *port-or-fd*, which may be a port or a positive exact integer file descriptor, respectively.

If *path* is a symbolic link, a stat of the file the link points to is returned from `sys-stat`.

If *port-or-fd* is not associated to a file, `sys-fstat` returns `#f`.

`sys-lstat` *path* [Function]

Like `sys-stat`, but it returns a stat of a symbolic link if *path* is a symbolic link.

```
gosh> (describe (sys-stat "gauche.h"))
#<<sys-stat> 0x815af70> is an instance of class <sys-stat>
slots:
  type      : regular
  perm      : 420
  mode      : 33188
  ino       : 845140
  dev       : 774
  rdev      : 0
  nlink     : 1
  uid       : 400
  gid       : 100
  size      : 79549
  atime     : 1020155914
  mtime     : 1020152005
  ctime     : 1020152005
```

`sys-stat->mode` *stat* [Function]
`sys-stat->ino` *stat* [Function]
`sys-stat->dev` *stat* [Function]
`sys-stat->rdev` *stat* [Function]
`sys-stat->nlink` *stat* [Function]
`sys-stat->size` *stat* [Function]
`sys-stat->uid` *stat* [Function]
`sys-stat->gid` *stat* [Function]
`sys-stat->atime` *stat* [Function]
`sys-stat->mtime` *stat* [Function]
`sys-stat->ctime` *stat* [Function]
`sys-stat->file-type` *stat* [Function]

Deprecated. Use `slot-ref` to access information of `<sys-stat>` object.

`sys-access` *pathname amode* [Function]

[POSIX] Returns a boolean value of indicating whether access of *pathname* is allowed in *amode*. This procedure signals an error if used in a suid/sgid program (see the note below). *amode* can be a combinations (logical or) of following predefined flags.

`R_OK` Checks whether *pathname* is readable by the current user.

`W_OK` Checks whether *pathname* is writable by the current user.

`X_OK` Checks whether *pathname* is executable (or searchable in case *pathname* is a directory) by the current user.

`F_OK` Checks whether *pathname* exists or not, regardless of the access permissions of *pathname*. (But you need to have access permissions of the directories containing *pathname*).

Note: Access(2) is known to be a security hole if used in suid/sgid program to check the real user's privilege of accessing the file.

`sys-chmod` *path mode* [Function]
`sys-fchmod` *port-or-fd mode* [Function]
Change the mode of the file named *path* or an opened file specified by *port-or-fd* to *mode*. *mode* must be a small positive integer whose lower 9 bits specifies POSIX style permission.

`sys-chown` *path owner-id group-id* [Function]
Change the owner and/or group of the file named *path* to *owner-id* and *group-id* respectively. *owner-id* and *group-id* must be an exact integer. If either of them is -1, the corresponding ownership is not changed.

`sys-utime` *path :optional atime mtime* [Function]
Change the file's access time and modification time to *atime* and *mtime*, respectively. If *atime* and/or *mtime* are omitted or `#f`, they are set to the current time. See also `touch-file` (see Section 12.24.4 [File operations], page 699).

6.25.4.5 Other file operations

`sys-chdir` *dir* [Function]
[POSIX] An interface to `chdir(2)`. See also `current-directory` (see Section 12.24.1 [Directory utilities], page 693).

`sys-pipe` *:key* (*buffering :line*) [Function]
[POSIX] Creates a pipe, and returns two ports. The first returned port is an input port and the second is an output port. The data put to the output port can be read from the input port.

Buffering can be `:full`, `:line` or `:none`, and specifies the buffering mode of the ports opened on the pipe. See Section 6.22.4 [File ports], page 221, for details of the buffering mode. The default mode is sufficient for typical cases.

```
(receive (in out) (sys-pipe)
  (display "abc\n" out)
  (flush out)
  (read-line in)) ⇒ "abc"
```

Note: the returned value is changed from version 0.3.15, in which `sys-pipe` returned a list of two ports.

`sys-mkfifo` *path mode* [Function]
[POSIX] creates a fifo (named pipe) with a name *path* and mode *mode*. *Mode* must be a positive exact integer to represent the file mode.

`sys-isatty` *port-or-fd* [Function]
[POSIX] *port-or-fd* may be a port or an integer file descriptor. Returns `#t` if the port is connected to the console, `#f` otherwise.

`sys-ttyname` *port-or-fd* [Function]
[POSIX] *port-or-fd* may be a port or an integer file descriptor. Returns the name of the terminal connected to the port, or `#f` if the port is not connected to a terminal.

`sys-truncate` *path length* [Function]
`sys-ftruncate` *port-or-fd length* [Function]
[POSIX] Truncates a regular file named by *path* or referenced by *port-or-fd* to a size of *length* bytes. If the file is larger than *length* bytes, the extra data is discarded. If the file is smaller than that, zero is padded.

6.25.5 Unix groups and users

Unix groups

`<sys-group>` [Builtin Class]

Unix group information. Has following slots.

`name` [Instance Variable of `<sys-group>`]

Group name.

`gid` [Instance Variable of `<sys-group>`]

Group id.

`passwd` [Instance Variable of `<sys-group>`]

Group password.

`mem` [Instance Variable of `<sys-group>`]

List of user names who are in this group.

`sys-getgrgid` *gid* [Function]
`sys-getgrnam` *name* [Function]

[POSIX] Returns `<sys-group>` object from an integer group id *gid* or a group name *name*, respectively. If the specified group doesn't exist, `#f` is returned.

`sys-gid->group-name` *gid* [Function]
`sys-group-name->gid` *name* [Function]

Convenience function to convert between group id and group name.

Unix users

`<sys-passwd>` [Builtin Class]

Unix user information. Has following slots.

`name` [Instance Variable of `<sys-passwd>`]

User name.

`uid` [Instance Variable of `<sys-passwd>`]

User ID.

`gid` [Instance Variable of `<sys-passwd>`]

User's primary group id.

`passwd` [Instance Variable of `<sys-passwd>`]

User's (encrypted) password. If the system uses the shadow password file, you just get obscure string like `"x"`.

`gecos` [Instance Variable of `<sys-passwd>`]

Gecos field.

`dir` [Instance Variable of `<sys-passwd>`]

User's home directory.

`shell` [Instance Variable of `<sys-passwd>`]

User's login shell.

`class` [Instance Variable of `<sys-passwd>`]

User's class (only available on some systems).

`sys-getpwuid` *uid* [Function]
`sys-getpwnam` *name* [Function]

[POSIX] Returns `<sys-passwd>` object from an integer user id *uid* or a user name *name*, respectively. If the specified user doesn't exist, `#f` is returned.

`sys-uid->user-name` *uid* [Function]
`sys-user-name->uid` *name* [Function]

Convenience functions to convert between user id and user name.

Password encryption

`sys-crypt` *key salt* [Function]

This is the interface to `crypt(3)`. *Key* and *salt* must be a string, and an encrypted string is returned. On systems where `crypt(3)` is not available, call to this function signals an error.

This routine is only for the code that needs to check password against the system's password database. If you are building user database on your own, you *must* use `crypt.bcrypt` module (see Section 12.7 [Password hashing], page 648) instead of this routine.

6.25.6 Locale

`sys-setlocale` *category locale* [Function]

[POSIX] Sets the locale of the category *category* to the locale *locale*. *category* must be an exact integer; the following pre-defined variables are available. *locale* must be a string locale name. Returns the locale name on success, or `#f` if the system couldn't change the locale.

`LC_ALL` [Variable]
`LC_COLLATE` [Variable]
`LC_CTYPE` [Variable]
`LC_MONETARY` [Variable]
`LC_NUMERIC` [Variable]
`LC_TIME` [Variable]

Predefined variables for possible *category* value of `sys-setlocale`.

`sys-localeconv` [Function]

[POSIX] Returns an assoc list of various information for formatting numbers in the current locale.

An example session. It may differ on your system settings.

```
(sys-localeconv)
 ⇒
   ((decimal_point . ".") (thousands_sep . "")
    (grouping . "") (int_curr_symbol . "")
    (currency_symbol . "") (mon_decimal_point . "")
    (mon_thousands_sep . "") (mon_grouping . "")
    (positive_sign . "") (negative_sign . "")
    (int_frac_digits . 127) (frac_digits . 127)
    (p_cs_precedes . #t) (p_sep_by_space . #t)
    (n_cs_precedes . #t) (n_sep_by_space . #t)
    (p_sign_posn . 127) (n_sign_posn . 127))

(sys-setlocale LC_ALL "fr_FR")
 ⇒ "fr_FR"

(sys-localeconv)
```

```
⇒
 ((decimal_point . ",") (thousands_sep . "")
  (grouping . "") (int_curr_symbol . "FRF ")
  (currency_symbol . "F") (mon_decimal_point . ",")
  (mon_thousands_sep . " ") (mon_grouping . "\x03\x03")
  (positive_sign . "") (negative_sign . "-")
  (int_frac_digits . 2) (frac_digits . 2)
  (p_cs_precedes . #f) (p_sep_by_space . #t)
  (n_cs_precedes . #f) (n_sep_by_space . #t)
  (p_sign_posn . 1) (n_sign_posn . 1))
```

6.25.7 Signal

Gauche can send out operating system's signals to the other processes (including itself) and can handle the incoming signals.

In multithread environment, all threads share the signal handlers, and each thread has its own signal mask. See Section 6.25.7.5 [Signals and threads], page 265, for details.

When a system call is interrupted by a signal, and a programmer defines a handler for the signal that doesn't transfer control to other context, the system call is restarted after the handler returns by default.

Here are some calls that are not simply restarted by signal interruption.

`close`
: This may be called through `sys-close`, `close-port`, or even implicitly when the underlying file is closed. When a signal interrupts this call, the passed fd is no longer in use. Simply restarting it can be a race if another thread just grabs the fd.

`dup2`
: This is called via `port-fd-dup!`. When `dup2` returns `EINTR`, the *newfd* is no longer in use and some other thread may grab it before restarting it.

`sleep`
`nanosleep`
: These calls tells the remaining time when they are interrupted. We restart them with that remaining time, not the original argument, so that the total sleep time would be close to what was given originally.

On Windows native platforms, signals don't work except some limited support of `sys-kill`.

6.25.7.1 Signals and signal sets

Each signal is referred by its signal number (a small integer) defined on the underlying operating system. Variables are pre-defined to the system's signal number. System's signal numbers may be architecture dependent, so you should use those variables rather than using literal integers.

`SIGABRT` [Variable]
`SIGALRM` [Variable]
`SIGCHLD` [Variable]
`SIGCONT` [Variable]
`SIGFPE` [Variable]
`SIGHUP` [Variable]
`SIGILL` [Variable]
`SIGINT` [Variable]
`SIGKILL` [Variable]
`SIGPIPE` [Variable]
`SIGQUIT` [Variable]
`SIGSEGV` [Variable]

SIGSTOP [Variable]
SIGTERM [Variable]
SIGTSTP [Variable]
SIGTTIN [Variable]
SIGTTOU [Variable]
SIGUSR1 [Variable]
SIGUSR2 [Variable]

These variables are bound to the signal numbers of POSIX signals.

SIGTRAP [Variable]
SIGIOT [Variable]
SIGBUS [Variable]
SIGSTKFLT [Variable]
SIGURG [Variable]
SIGXCPU [Variable]
SIGXFSZ [Variable]
SIGVTALRM [Variable]
SIGPROF [Variable]
SIGWINCH [Variable]
SIGPOLL [Variable]
SIGIO [Variable]
SIGPWR [Variable]

These variables are bound to the signal numbers of system-dependent signals. Not all of them may be defined on some systems.

Besides each signal numbers, you can refer to a set of signals using a `<sys-sigset>` object. It can be used to manipulate the signal mask, and to install a signal handler to a set of signals at once.

`<sys-sigset>` [Class]

A set of signals. An empty sigset can be created by

```
(make <sys-sigset>) ⇒ #<sys-sigset []>
```

`sys-sigset` *signal . . .* [Function]

Creates and returns an instance of `<sys-sigset>` with members *signal* . . . . Each *signal* may be either a signal number, another `<sys-sigset>` object, or `#t` for all available signals.

```
(sys-sigset SIGHUP SIGINT) ⇒ #<sys-sigset [HUP|INT]>
```

`sys-sigset-add!` *sigset signal . . .* [Function]
`sys-sigset-delete!` *sigset signal . . .* [Function]

Sigset must be a `<sys-sigset>` object. Those procedures adds and removes the specified signals from *sigset* respectively, and returns the result. *sigset* itself is also modified.

signal may be either a signal number, another `<sys-sigset>` object, or `#t` for all available signals.

`sys-sigset-fill!` *sigset* [Function]
`sys-sigset-empty!` *sigset* [Function]

Fills *sigset* by all available signals, or empties *sigset*.

`sys-signal-name` *signal* [Function]

Returns the human-readable name of the given signal number. (Note that signal numbers are system-dependent.)

```
(sys-signal-name 2) ⇒ "SIGINT"
```

6.25.7.2 Sending signals

To send a signal, you can use `sys-kill` which works like `kill(2)`.

`sys-kill` *pid sig* [Function]

[POSIX] Sends a signal *sig* to the specified process(es). *Sig* must be a positive exact integer. *pid* is an exact integer and specifies the target process(es):

- If *pid* is positive, it is the target process id.
- If *pid* is zero, the signal is sent to every process in the process group of the current process.
- If *pid* is less than -1, the signal is sent to every process in the process group *-pid*.

On Windows native platforms, `sys-kill` may take positive integer or a process handle (`<win:handle>` instance) as *pid*. Only `SIGKILL`, `SIGINT` and `SIGABRT` are allowed as *sig*; Gauche uses `TerminateProcess` to terminate the target process for `SIGKILL`, and sends the target process `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` for `SIGINT` and `SIGABRT`, respectively.

There's no Scheme equivalence for `raise()`, but you can use `(sys-kill (sys-getpid) *sig*)`.

6.25.7.3 Handling signals

You can register signal handling procedures in Scheme. (In multithread environment, signal handlers are shared by all threads; see Section 6.25.7.5 [Signals and threads], page 265, for details).

When a signal is delivered to the Scheme process, the VM just records it and processes it later at a 'safe point' where the state of VM is consistent. We call the signal is *pending* when it is registered by the VM but not processed yet.

(Note that this makes handling of some signals such as `SIGILL` useless, for the process can't continue sensible execution after recording the signal).

If the same signal is delivered more than once before VM processes the first one, the second one and later have no effect. (This is consistent to the traditional Unix signal model.) In other words, for each VM loop a signal handler can be invoked at most once per each signal.

When too many signals of the same kind are pending, Gauche assumes something has gone wrong (e.g. infinite loop inside C-routine) and aborts the process. The default of this limit is set rather low (3), to allow unresponsive interactive script to be terminated by typing Ctrl-C three times. Note that the counter is individual for each signal; Gauche won't abort if one `SIGHUP` and two `SIGINT`s are pending, for example. You can change this limit by `set-signal-pending-limit` described below.

When you're using the `gosh` interpreter, the default behavior for each signal is as in the following table.

`SIGABRT, SIGILL, SIGKILL, SIGCONT, SIGSTOP, SIGSEGV, SIGBUS`
: Cannot be handled in Scheme. `Gosh` follows the system's default behavior.

`SIGCHLD, SIGTSTP, SIGTTIN, SIGTTOU, SIGWINCH`
: No signal handles are installed for these signals by `gosh`, so the process follows the system's default behavior. Scheme programs can install its own signal handler if necessary.

`SIGHUP, SIGQUIT, SIGTERM`
: `Gosh` installs a signal handler for these signals that exits from the application with code 0.

`SIGPIPE`
: `Gosh` installs a signal handler that does nothing—that is, this signal is effectively ignored by default.

It is a design choice. Since Gauche delays actual handling of signals, `SIGPIPE` would be handled after the system call that tries to write to a broken pipe returns with `EPIPE`. That makes the signal a lot less useful, for we can handle the situation with error handlers for `<system-error>` with `EPIPE`.

The default Unix behavior of `SIGPIPE` is to terminate the process. It is useful for the traditional command-line tools that are often piped together—if one of downstream commands fails, the upstream process receives `SIGPIPE` and the entire command chain is shut down without a fuss. The signal is, however, rather an annoyance for other types of output such as sockets.

Gauche does support this "exit when pipe gets stuck" convention by ports. A port can be configured as *sigpipe sensitive*; if writing to that port caused `EPIPE`, it terminates the process. By default, standard output and standard error output are configured in that way.

`SIGPWR, SIGXCPU, SIGUSR1, SIGUSR2`

On Linux platforms with thread support, these signals are used by the system and not available for Scheme. On other systems, these signals behaves the same as described below.

`other signals`

`Gosh` installs the default signal handler, which raises `<unhandled-signal-error>` condition (see Section 6.20.4 [Conditions], page 212). Scheme programs can override it by its own signal handler.

If you're using Gauche embedded in some other application, it may redefine the default behavior.

Use the following procedures to get/set signal handlers from Scheme.

`set-signal-handler!` *signals handler :optional sigmask* [Function]

Signals may be a single signal number or a `<sys-sigset>` object, and *handler* should be either `#t`, `#f`, `#<undef>`, or a procedure that takes one argument. If *handler* is a procedure, it will be called when the process receives one of specified signal(s), with the received signal number as an argument.

By default, the signals in *signals* are blocked (in addition to the signal mask in effect at that time) during *handler* is executed, so that *handler* won't be reentered by the same signal(s). You can provide a `<sys-sigset>` object to the *sigmask* arg to specify the signals to be blocked explicitly. Note that the signal mask is per-thread; if more than one thread unblocks a signal, the handler may still be invoked during execution of the handler (in other thread) even if you specify *sigmask*. You have to set the threads' signal mask properly to avoid such situation.

It is safe to do anything in *handler*, including throwing an error or invoking continuation captured elsewhere. (However, continuations captured inside *handler* will be invalid once you return from *handler*).

If *handler* is `#t`, the operating system's default behavior is set to the specified signal(s). If *handler* is `#f`, the specified signals(s) will be ignored.

If *handler* is `#<undef>` (see Section 6.5 [Undefined values], page 124), it indicates Gauche to leave the current OS's signal handler as it is. This value isn't as much use in `set-signal-handler!` as in `get-signal-handler`: If `#<undef>` is passed to `set-signal-handler!`, it immediately returns without modifying anything. However, if you get `#<undef>` from `get-signal-handler`, you can know that the signal handler behavior hasn't been modified by Gauche. (Note that once Gauche ever installs a signal handler, there is no way to revert back to make `get-signal-handler` return `#<undef>`).

Note that signal handler setting is shared among threads in multithread environment. The handler is called from the thread which is received the signal. See Section 6.25.7.5 [Signals and threads], page 265, for details.

`get-signal-handler` *signum* [Function]
`get-signal-handler-mask` *signum* [Function]

Returns the handler setting, or signal mask setting, of a signal *signum*, respectively. See `set-signal-handler!` for the meaning of the return value of `get-signal-handler`.

`get-signal-handlers` [Function]

Returns an associative list of all signal handler settings. Car of each element of returned list is a `<sys-sigset>` object, and cdr of it is the handler (a procedure or a boolean value) of the signals in the set.

`get-signal-pending-limit` [Function]
`set-signal-pending-limit` *limit* [Function]

Gets/sets the maximum number of pending signals per each signal type. If the number of pending signals exceeds this limit, Gauche aborts the process. See the explanation at the beginning of this section for the details. *Limit* must be a nonnegative exact integer. In the current implementation the maximum number of *limit* is 255. Setting limit to zero makes the number of pending signals unlimited.

`with-signal-handlers` (*handler-clause* ...) *thunk* [Macro]

A convenience macro to install signal handlers temporarily during execution of *thunk*. (Note: though this is convenient, this has certain dangerous properties described below. Use with caution.)

Each *Handler-clause* may be one of the following forms.

`(signals expr ...)`
: *Signals* must be an expression that will yield either a signal, a list of signals, or a `<sys-sigset>` object. Installs a signal handler for *signals* that evaluates *expr* ... when one of the signals in *signals* is delivered.

`(signals => handler)`
: This form sets the handler of *signals* to *handler*, where *handler* should be either `#t`, `#f` or a procedure that takes one argument.

  If *handler* is a procedure, it will be called when the process receives one of specified signal(s), with the received signal number as an argument. If *handler* is `#t`, the operating system's default behavior is set to the specified signal(s). If *handler* is `#f`, the specified signals(s) will be ignored.

When the control exits from *thunk*, the signal handler setting before `with-signal-handlers` are recovered.

CAVEAT: If you're setting more than one signal handlers, they are installed in serial. If a signal is delivered before all the handlers are installed, the signal handler state may be left inconsistent. Also note that the handler setting is a global state; you can't set "thread local" handler by `with-signal-handlers`, although the form may be misleading.

6.25.7.4 Masking and waiting signals

A Scheme program can set a signal mask, which is a set of signals to be blocked from delivery. If a signal is delivered which is completely blocked in the process, the signal becomes "pending". The pending signal may be delivered once the signal mask is changed not to block the specified signal. (However, it depends on the operating system whether the pending signals are queued or not.)

In multithread environment, each thread has its own signal mask.

sys-sigmask *how mask* [Function]

Modifies the current thread's signal mask, and returns the previous signal mask. *Mask* should be a `<sys-sigset>` object to specify the new mask, or `#f` if you just want to query the current mask without modifying one.

If you give `<sys-sigset>` object to *mask*, *how* argument should be one of the following integer constants:

`SIG_SETMASK`
: Sets *mask* as the thread's signal mask.

`SIG_BLOCK`
: Adds signals in *mask* to the thread's signal mask.

`SIG_UNBLOCK`
: Removes signals in *mask* from the thread's signal mask.

sys-sigsuspend *mask* [Function]

Atomically sets thread's signal mask to *mask* and suspends the calling thread. When a signal that is not blocked and has a signal handler installed is delivered, the associated handler is called, then `sys-sigsuspend` returns.

sys-sigwait *mask* [Function]

[POSIX] *Mask* must be a `<sys-sigset>` object. If any of signals in *mask* is/are pending in the OS, atomically clears one of them and returns the signal number of the cleared one. If there's no signal in *mask* pending, `sys-sigwait` blocks until any of the signals in *mask* arrives.

You have to block all signals in *mask* in all threads before calling `sys-sigwait`. If there's a thread that doesn't block the signals, the behavior of `sys-sigwait` is undefined.

Note: `Sys-sigwait` uses system's `sigwait` function, whose behavior is not defined if there's a signal handler on the signals it waits. To avoid complication, `sys-sigwait` resets the handlers set to the signals included in *mask* before calling `sigwait` to `SIG_DFL`, and restores them after `sigwait` returns. If another thread changes signal handlers while `sys-sigwait` is waiting, the behavior is undefined; you shouldn't do that.

6.25.7.5 Signals and threads

The semantics of signals looks a bit complicated in the multithread environment. Nevertheless, it is pretty comprehensible once you remember a small number of rules. Besides, Gauche sets up the default behavior easy to use, while allowing programmers to do tricky stuff.

If you don't want to be bothered by the details, just remember one thing, with one sidenote. **By default**, signals are handled by the primordial (main) thread. However, if the main thread is suspended on mutex or condition variable, the signal may not be handled at all, so be careful.

Now, if you are curious about the details, here are the rules:

- The signal handler setting is shared by all threads.
- The signal mask is thread-specific.
- If a process receives an asynchronous signal (think it as a signal delivered from other processes), one thread is chosen, out of threads which don't block that signal.
- The signal handler is run on the chosen thread. However, if the chosen thread is waiting for acquiring a mutex lock or a condition variable, the handling of signal will be delayed until the thread is restarted. Signal delivery itself doesn't restart the thread.

Now, these rules have several implications.

If there are more than one thread that don't block a particular signal, you can't know which thread receives the signal. Such a situation is much less useful in Gauche than C programs because of the fact that the signal handling can be delayed indefinitely if the receiver thread is waiting on mutex or condition variable. So, it is recommended to make sure, for each signal, there is only one thread that can receive it.

In Gauche, all threads created by `make-thread` (see Section 9.33.2 [Thread procedures], page 459) blocks all the signals by default (except the reserved ones). This lets all the signals to be directed to the primordial (main) thread.

Another strategy is to create a thread dedicated for handling signals. To do so, you have to block the signals in the primordial thread, then create the signal-handling thread, and within that thread you unblock all the signals. Such a thread can just loop on `sys-pause`.

```
(thread-start!
  (make-thread
    (lambda ()
      (sys-sigmask SIG_SETMASK (make <sys-sigset>)) ;;empty mask
      (let loop () (sys-pause) (loop)))))
```

Complicated application may want to control per-thread signal handling precisely. You can do so, just make sure that at any moment only the designated thread unblocks the desired signal.

6.25.8 System inquiry

`sys-uname` [Function]

[POSIX] Returns a list of five elements, `(`*`sysname nodename release version machine`*`)`.

`sys-gethostname` [Function]

Returns the host name. If the system doesn't have gethostname(), the second element of the list returned by `sys-uname` is used.

`sys-getdomainname` [Function]

Returns the domain name. If the system doesn't have getdomainname(), `"localdomain"` is returned.

`sys-getcwd` [Function]

[POSIX] Returns the current working directory by a string. If the current working directory couldn't be obtained from the system, an error is signaled. See also `sys-chdir` (see Section 6.25.4.5 [Other file operations], page 257), `current-directory` (see Section 12.24.1 [Directory utilities], page 693).

`sys-getgid` [Function]
`sys-getegid` [Function]

[POSIX] Returns integer value of real and effective group id of the current process, respectively. Use `sys-gid->group-name` or `sys-getgrgid` to obtain the group's name and other information associated to the returned group id (see Section 6.25.5 [Unix groups and users], page 257).

`sys-setgid` *gid* [Function]

[POSIX] Sets the effective group id of the current process.

`sys-getuid` [Function]
`sys-geteuid` [Function]

[POSIX] Returns integer value of real and effective user id of the current process, respectively. Use `sys-uid->user-name` or `sys-getpwuid` to obtain the user's name and other information associated to the returned user id (see Section 6.25.5 [Unix groups and users], page 257).

`sys-setuid` *uid* [Function]

[POSIX] Sets the effective user id of the current process.

`sys-getgroups` [Function]
[POSIX] Returns a list of integer ids of supplementary groups.

`sys-setgroups` *gids* [Function]
Sets the current process's groups to the given list of integer group ids. The caller must have the appropriate privilege.

This procedure is only available when the feature id `gauche.sys.setgroups` exists. Use `cond-expand` for the portable program:

```
(cond-expand
  [gauche.sys.setgroups (sys-setgroups '(0 1))]
  [else])
```

`sys-getlogin` [Function]
[POSIX] Returns a string of the name of the user logged in on the controlling terminal of the current process. If the system can't determine the information, `#f` is returned.

`sys-getpgrp` [Function]
[POSIX] Returns a process group id of the current process.

`sys-getpgid` *pid* [Function]
Returns a process group id of the process specified by *pid*. If *pid* is zero, the current process is used.

Note that `getpgid()` call is not in POSIX. If the system doesn't have `getpgid()`, `sys-getpgid` still works if *pid* is zero (it just calls `sys-getpgrp`), but signals an error if *pid* is not zero.

`sys-setpgid` *pid pgid* [Function]
[POSIX] Sets the process group id of the process *pid* to *pgid*. If *pid* is zero, the process ID of the current process is used. If *pgid* is zero, the process ID of the process specified by `pid` is used. (Hence `sys-setpgid(0, 0)` sets the process group id of the current process to the current process id).

`sys-setsid` [Function]
[POSIX] Creates a new session if the calling process is not a process group leader.

`sys-getpid` [Function]
`sys-getppid` [Function]
[POSIX] Returns the current process id and the parent process id, respectively.

`sys-times` [Function]
[POSIX]

`sys-ctermid` [Function]
[POSIX] Returns the name of the controlling terminal of the process. This may be just a `"/dev/tty"`. See also `sys-ttyname`.

`sys-getrlimit` *resource* [Function]
`sys-setrlimit` *resource current :optional maximum* [Function]
[POSIX] Get and set resource limits respectively. *Resource* is an integer constant to specify the resource of concern. The following constants are defined. (The constants marked as bsd and/or linux indicates that they are not defined in POSIX but defined in BSD and/or Linux. Other systems may or may not have them. Consult `getrlimit` manpage of your system for the details.)

`RLIMIT_AS` `RLIMIT_CORE`

`RLIMIT_CPU`	`RLIMIT_DATA`
`RLIMIT_FSIZE`	`RLIMIT_LOCKS`
`RLIMIT_MEMLOCK (bsd/linux)`	`RLIMIT_MSGQUEUE (linux)`
`RLIMIT_NICE (linux)`	`RLIMIT_NOFILE`
`RLIMIT_NPROC (bsd/linux)`	`RLIMIT_RSS (bsd/linux)`
`RLIMIT_RTPRIO (linux)`	`RLIMIT_SIGPENDING (linux)`
`RLIMIT_SBSIZE`	`RLIMIT_STACK`
`RLIMIT_OFILE`	

`sys-strerror` *errno* [Function]

Errno must be an exact nonnegative integer representing a system error number. This function returns a string describing the error.

To represent *errno*, the following constants are defined. Each constant is bound to an exact integer representing the system's error number. Note that the actual value may differ among systems, and some of these constants may not be defined on some systems.

E2BIG	EHOSTDOWN	ENETDOWN	ENXIO
EACCES	EHOSTUNREACH	ENETRESET	EOPNOTSUPP
EADDRINUSE	EIDRM	ENETUNREACH	EOVERFLOW
EADDRNOTAVAIL	EILSEQ	ENFILE	EPERM
EADV	EINPROGRESS	ENOANO	EPFNOSUPPORT
EAFNOSUPPORT	EINTR	ENOBUFS	EPIPE
EAGAIN	EINVAL	ENOCSI	EPROTO
EALREADY	EIO	ENODATA	EPROTONOSUPPORT
EBADE	EISCONN	ENODEV	EPROTOTYPE
EBADF	EISDIR	ENOENT	ERANGE
EBADFD	EISNAM	ENOEXEC	EREMCHG
EBADMSG	EKEYEXPIRED	ENOKEY	EREMOTE
EBADR	EKEYREJECTED	ENOLCK	EREMOTEIO
EBADRQC	EKEYREVOKED	ENOLINK	ERESTART
EBADSLT	EL2HLT	ENOMEDIUM	EROFS
EBFONT	EL2NSYNC	ENOMEM	ESHUTDOWN
EBUSY	EL3HLT	ENOMSG	ESOCKTNOSUPPORT
ECANCELED	EL3RST	ENONET	ESPIPE
ECHILD	ELIBACC	ENOPKG	ESRCH
ECHRNG	ELIBBAD	ENOPROTOOPT	ESRMNT
ECOMM	ELIBEXEC	ENOSPC	ESTALE
ECONNABORTED	ELIBMAX	ENOSR	ESTRPIPE
ECONNREFUSED	ELIBSCN	ENOSTR	ETIME
ECONNRESET	ELNRNG	ENOSYS	ETIMEDOUT
EDEADLK	ELOOP	ENOTBLK	ETOOMANYREFS
EDEADLOCK	EMEDIUMTYPE	ENOTCONN	ETXTBSY
EDESTADDRREQ	EMFILE	ENOTDIR	EUCLEAN
EDOM	EMLINK	ENOTEMPTY	EUNATCH
EDOTDOT	EMSGSIZE	ENOTNAM	EUSERS
EDQUOT	EMULTIHOP	ENOTSOCK	EWOULDBLOCK
EEXIST	ENAMETOOLONG	ENOTTY	EXDEV
EFAULT	ENAVAIL	ENOTUNIQ	EXFULL
EFBIG			

`sys-errno->symbol` *k* [Function]
`sys-symbol->errno` *symbol* [Function]

These procedures convert between integer error number and the symbol of its unix name (e.g. `EINTR`).

If the given error number or name isn't available on the running platform, those procedures return `#f`. See `sys-strerror` above for potentially available error names.

Valid error names and their actual values differ among platforms. These procedures make it easy to write portable meta-code that deal with system errors.

6.25.9 Time

Gauche has two representations of time, one is compatible to POSIX API, and the other is compatible to SRFI-18, SRFI-19 and SRFI-21. Most procedures accept both representations; if not, the representation the procedure accepts is indicated as either 'POSIX time' or 'SRFI time'.

POSIX time is represented by a real number which is a number of seconds since Unix Epoch (Jan 1, 1970, 0:00:00GMT). Procedure `sys-time`, which corresponds to POSIX `time(2)`, returns this time representation.

SRFI-compatible time is represented by an object of `<time>` class, which keeps seconds and nanoseconds, as well as the type of the time (UTC, TAI, duration, process time, etc). `Current-time` returns this representation.

POSIX time

`sys-time` [Function]

[POSIX] Returns the current time in POSIX time (the time since Epoch (00:00:00 UTC, January 1, 1970), measured in seconds). It may be a non-integral number, depending on the architecture.

Note that POSIX's definition of "seconds since the Epoch" doesn't take leap seconds into account.

`sys-gettimeofday` [Function]

Returns two values. The first value is a number of seconds, and the second value is a fraction in a number of microseconds, since 1970/1/1 0:00:00 UTC. If the system doesn't have `gettimeofday` call, this function calls `time()`; in that case, microseconds portion is always zero.

`<sys-tm>` [Builtin Class]

Represents `struct tm`, a calendar date. It has the following slots.

`sec` [Instance Variable of `<sys-tm>`]

Seconds. 0-61.

`min` [Instance Variable of `<sys-tm>`]

Minutes. 0-59.

`hour` [Instance Variable of `<sys-tm>`]

Hours. 0-23.

`mday` [Instance Variable of `<sys-tm>`]

Day of the month, counting from 1. 1-31.

`mon` [Instance Variable of `<sys-tm>`]

Month, counting from 0. 0-11.

`year` [Instance Variable of `<sys-tm>`]
Years since 1900, e.g. 102 for the year 2002.

`wday` [Instance Variable of `<sys-tm>`]
Day of the week. Sunday = 0 .. Saturday = 6.

`yday` [Instance Variable of `<sys-tm>`]
Day of the year. January 1 = 0 .. December 31 = 364 or 365.

`isdst` [Instance Variable of `<sys-tm>`]
A flag that indicates if the daylight saving time is in effect. Positive if DST is in effect, zero if not, or negative if unknown.

`sys-gmtime` *time* [Function]
`sys-localtime` *time* [Function]
[POSIX] Converts *time* to `<sys-tm>` object, represented in GMT or local timezone, respectively. *Time* can be either POSIX-time or SRFI-time.

`sys-ctime` *time* [Function]
[POSIX] Converts *time* to it string representation, using POSIX ctime(). *Time* can be either POSIX-time or SRFI-time.

`sys-difftime` *time1 time0* [Function]
[POSIX] Returns the difference of two times in the real number of seconds. *Time0* and *time1* can be either POSIX-time or SRFI-time.

`sys-asctime` *tm* [Function]
[POSIX] Converts `<sys-tm>` object *tm* to a string representation.

`sys-strftime` *format tm* [Function]
[POSIX] Converts `<sys-tm>` object *tm* to a string representation, according to a format string *format*.

`sys-mktime` *tm* [Function]
[POSIX] Converts `<sys-tm>` object *tm*, expressed as local time, to the POSIX-time (number of seconds since Epoch).

`sys-tm->alist` *tm* [Function]
(Deprecated function)

SRFI time

`<time>` [Builtin Class]
The `<time>` object also represents a point of time.

`type` [Instance Variable of `<time>`]
Indicates time type. `time-utc` is the default, and that represents the number of seconds since Unix Epoch. SRFI-19 (see Section 11.7 [Time data types and procedures], page 600) adds more types.

`second` [Instance Variable of `<time>`]
Second part of the time.

`nanosecond` [Instance Variable of `<time>`]
Nanosecond part of the time.

`current-time` [Function]

[SRFI-18][SRFI-21] Returns the `<time>` object representing the current time in `time-utc`. See Section 11.7 [Time data types and procedures], page 600, for it redefines `current-time` to allow optional argument to specify time type.

`time?` *obj* [Function]

[SRFI-18][SRFI-19][SRFI-21] Returns `#t` if *obj* is a time object.

`time->seconds` *time* [Function]
`seconds->time` *seconds* [Function]

[SRFI-18][SRFI-21] Converts between time object and the number of seconds (POSIX-time). *Time* argument of `time->seconds` has to be a `<time>` object.

6.25.10 Process management

The following procedures provide pretty raw, direct interface to the system calls. See also Section 9.25 [High-level process interface], page 421, which provides more convenient process handling on top of these primitives.

Fork and exec

`sys-system` *command* [Function]

[POSIX] Runs *command* in a subprocess. *command* is usually passed to `sh`, so the shell metacharacters are interpreted.

This function returns an integer value `system()` returned. Since POSIX doesn't define what `system()` returns, you can't interpret the returned value in a portable way.

On Windows native platforms this will pass the argument to `cmd.exe`.

`sys-fork` [Function]

[POSIX] Fork the current process. Returns 0 if you're in the child process, and a child process' pid if you're in the parent process. All the opened file descriptors are shared between the parent and the child. See `fork(2)` of your system for details.

If the child process runs some Scheme code and exits instead of calling `sys-exec`, it should call `sys-exit` instead of `exit` to terminate itself. Normal exit call tries to flush the file buffers, and on some OS it messes up the parent's file buffers.

It should be noted that `sys-fork` is not safe when multiple threads are running. Because `fork(2)` copies the process' memory image which includes any mutex state, a mutex which is locked by another thread at the time of `sys-fork` remains locked in the child process, nevertheless the child process doesn't have the thread that unlock it! (This applies to the internal mutexes as well, so even you don't use Scheme mutex explicitly, this situation can always happen.)

If what you want is to spawn another program in a multi-threaded application, use `sys-fork-and-exec` explained below. If you absolutely need to run Scheme code in the child process, a typical technique is that you fork a manager process at the beginning of application, and whenever you need a new process you ask the manager process to fork one for you.

This procedure is not available on Windows native platforms.

`sys-exec` *command args :key directory iomap sigmask* [Function]

[POSIX+] Execute *command* with *args*, a list of arguments. The current process image is replaced by *command*, so this function never returns.

All elements of *args* must be strings. The first element of *args* is used as `argv[0]`, i.e. the program name.

The keyword argument *directory* must be a string of a directory name or `#f`. If it is a string, `sys-exec` change current working directory there before executing the program.

The *iomap* keyword argument, when provided, specifies how the open file descriptors are treated. It must be the following format:

```
((to-fd . from-port-or-fd) ...)
```

To-fd must be an integer, and *from-port-or-fd* must be an integer file descriptor or a port. Each element of the list makes the file descriptor of *from-port-or-fd* of the current process be mapped to the file descriptor *to-fd* in the executed process.

If *iomap* is provided, any file descriptors other than specified in the iomap list will be closed before `exec()`. Otherwise, all file descriptors in the current process remain open.

```
(sys-exec "ls" '("ls" "-l")) ⇒ ;; ls is executed.

(let ((out (open-output-file "ls.out")))
  (sys-exec "ls" '("ls" "-l") :iomap `((2 . 1) (1 . ,out)))
   ⇒
  ;; ls is executed, with its stderr redirected
  ;; to the current process's stdout, and its
  ;; stdout redirected to the file "ls.out".
```

The *sigmask* keyword argument can be an instance of `<sys-sigset>` or `#f` (See Section 6.25.7 [Signal], page 260, for the details of signal masks). If it is an instance of `<sys-sigset>`, the signal mask of calling thread is replaced by it just before `exec(2)` is called. It is useful, for example, to run an external program from a thread where all signals are blocked (which is the default; see Section 6.25.7.5 [Signals and threads], page 265). Without setting *sigmask*, the `exec`ed process inherits calling thread's signal mask and become a process that blocks all signals, which is not very convenient in most cases.

When `sys-exec` encounters an error, most of the time it raises an error condition. Once the file descriptors are permuted, however, it would be impractical to handle errors in reasonable way (you don't even know stderr is still available!), so Gauche simply exits on the error.

On Windows native platforms, only redirections of stdin, stdout and stderr are handled. Singal mask is ignored, for Windows doesn't have signals as the means of interprocess communication.

`sys-fork-and-exec` *command args :key directory iomap sigmask detached* [Function]

Like `sys-exec`, but executes `fork(2)` just before remapping I/O, altering signal mask and call `execvp(2)`. Returns child's process id. The meanings of arguments are the same as `sys-exec`.

It is strongly recommended to use this procedure instead of `sys-fork` and `sys-exec` combination when you need to spawn another program while other threads are running. No memory allocation nor lock acquisition is done between `fork(2)` and `execvp(2)`, so it's pretty safe in the multithreaded environment.

On Windows native platforms, this procedure returns a Windows handle object (`<win:handle>`) of the created process instead of an integer process ID. See below for Windows process handle specific API.

Like `sys-exec`, only redirections of stdin, stdout and stderr are handled on Windows native platforms.

When a true value is given to the *detached* keyword argument, the executed process is detached from the current process group and belongs to its own group. That is, it won't be affected to the signal sent to the process group the caller process currently belongs to. It is a part of the common idioms to start a daemon process.

On Unix platforms, besides the executed process gets its own session by `setsid(2)`, it performs extra `fork(2)` to make its parent be the `init` process (pid=1). (Note: It means the running process is actually a grandchild of the calling process, although that relationship isn't preserved. The returned pid is the running process's one, not the intermediate process that exits immediately.)

On Windows native platforms, this flag causes the new process to be created with the `CREATE_NEW_PROCESS_GROUP` creation flag.

Wait

`sys-wait` [Function]

[POSIX] Calls system's `wait(2)`. The process suspends its execution until one of the child terminates. Returns two exact integer values, the first one is the child's process id, and the second is a status code. The status code can be interpreted by the following functions.

`sys-waitpid` *pid :key nohang untraced* [Function]

[POSIX] This is an interface to `waitpid(3)`, an extended version of wait.

pid is an exact integer specifying which child(ren) to be waited. If it is a positive integer, it waits for that specific child. If it is zero, it waits for any member of this process group. If it is -1, it waits for any child process. If it is less than -1, it waits for any child process whose process group id is equal to the absolute value of *pid*.

If there's no child process to wait, or a specific *pid* is given but it's not a child process of the current process, an error (`<system-error>`, `ECHILD`) is signaled.

The calling process suspends until one of those child process is terminated, unless true is specified to the keyword argument *nohang*.

If true is specified to the keyword argument *untraced*, the status of stopped child process can be also returned.

The return values are two exact integers, the first one is the child process id, and the second is a status code. If *nohang* is true and no child process status is available, the first value is zero.

On Windows native platforms, this procedure may also accept a Windows process handle (`<win:handle>`) object as *pid* to wait the specific process. You can pass `-1` as *pid* to wait for any children, but you cannot wait for a specific process group.

`sys-wait-exited?` *status* [Function]
`sys-wait-exit-status` *status* [Function]

[POSIX] The argument is an exit status returned as a second value from `sys-wait` or `sys-waitpid`. `sys-wait-exited?` returns `#t` if the child process is terminated normally. `sys-wait-exit-status` returns the exit code the child process passed to `exit(2)`, or the return value of `main()`.

`sys-wait-signaled?` *status* [Function]
`sys-wait-termsig` *status* [Function]

[POSIX] The argument is an exit status returned as a second value from `sys-wait` or `sys-waitpid`. `sys-wait-signaled?` returns `#t` if the child process is terminated by an uncaught signal. `sys-wait-termsig` returns the signal number that terminated the child.

`sys-wait-stopped?` *status* [Function]
`sys-wait-stopsig` *status* [Function]

[POSIX] The argument is an exit status returned as a second value from `sys-waitpid`. `sys-wait-stopped?` returns `#t` if the child process is stopped. This status can be caught only by `sys-waitpid` with true *untraced* argument. `sys-wait-stopsig` returns the signum number that stopped the child.

On Windows native platforms, exit code is not structured as on Unix. You cannot distinguish a process being exited voluntarily or by forced termination. Gauche uses exit code `#xff09` to terminate other process with `sys-kill`, and the above `sys-wait-*` procedures are adjusted accordingly, so that `sys-wait-signaled?` can likely to be used to check whether if the child process is terminated by Gauche. (See Section 6.25.7 [Signal], page 260, for the details of signal support on Windows.) `Sys-wait-stopped?` never returns true on Windows native platforms (yet).

Windows specific utilities

The following procedures are to access Windows process handle. They are only available on Windows native platforms.

`sys-win-process?` *obj* [Function]

[Windows] Returns `#t` iff *obj* is a Windows process handle object.

`sys-win-process-pid` *handle* [Function]

[Windows] Returns an integer PID of the process represented by a Windows process handle *handle*. An error is signaled if *handle* is not a valid Windows process handle.

Note that the API to get a pid from a process handle is only provided on or after Windows XP SP1. If you call this procedure on Windows version before that, `-1` will be returned.

6.25.11 I/O multiplexing

The interface functions for `select(2)`. The higher level interface is provided on top of these primitives; see Section 9.28 [Simple dispatcher], page 440.

`<sys-fdset>` [Builtin Class]

Represents `fd_set`, a set of file descriptors. You can make an empty file descriptor set by make method:

```
(make <sys-fdset>)
```

`sys-fdset` *elt* . . . [Function]

Creates a new `<sys-fdset>` instance with file descriptors specified by *elt* . . . . Each *elt* can be an integer file descriptor, a port, or a `<sys-fdset>` instance. In the last case, the descriptors in the given fdset is copied to the new fdset.

`sys-fdset-ref` *fdset port-or-fd* [Function]
`sys-fdset-set!` *fdset port-or-fd flag* [Function]

Gets and sets specific file descriptor bit of *fdset*. *port-or-fd* may be a port or an integer file descriptor. If *port-or-fd* is a port that doesn't have associated file descriptor, `sys-fdset-ref` returns `#f`, and `sys-fdset-set!` doesn't modify *fdset*. *flag* must be a boolean value.

You can use generic setter of `sys-fdset-ref` as this:

```
(set! (sys-fdset-ref fdset port-or-fd) flag)
  ≡ (sys-fdset-set! fdset port-or-fd flag)
```

`sys-fdset-copy!` *dest-fdset src-fdset* [Function]

Copies the content of *src-fdset* into *dest-fdset*. Returns *dest-fdset*.

`sys-fdset-clear!` *fdset* [Function]

Empties and returns *fdset*.

`sys-fdset->list` *fdset* [Function]
`list->sys-fdset` *fds* [Function]

Converts an fdset to a list of integer file descriptors and vice versa. In fact, `list->sys-fdset` works just like `(lambda (fds) (apply sys-fdset fds))`, so it accepts ports and other fdsets as well as integer file descriptors.

`sys-fdset-max-fd` *fdset* [Function]

Returns the maximum file descriptor number in *fdset*.

`sys-select` *readfds writefds exceptfds :optional timeout* [Function]
`sys-select!` *readfds writefds exceptfds :optional timeout* [Function]

Waits for a set of file descriptors to change status. *readfds*, *writefds*, and *exceptfds* are `<fdset>` objects to represent a set of file descriptors to watch. File descriptors in *readfds* are watched to see if characters are ready to be read. File descriptors in *writefds* are watched if writing to them is ok. File descriptors in *exceptfds* are watched for exceptions. You can pass `#f` to one or more of those arguments if you don't care about watching the condition.

timeout specifies maximum time `sys-select` waits for the condition change. It can be a real number, for number of microseconds, or a list of two integers, the first is the number of seconds and the second is the number of microseconds. If you pass `#f`, `sys-select` waits indefinitely.

`sys-select` returns four values. The first value is a number of descriptors it detected status change. It may be zero if timeout expired. The second, third and fourth values are `<fdset>` object that contains a set of descriptors that changed status for reading, writing, and exception, respectively. If you passed `#f` to one or more of *readfds*, *writefds* and *exceptfds*, the corresponding return value is `#f`.

`sys-select!` variant works the same as `sys-select`, except it modifies the passed `<fdset>` arguments. `sys-select` creates new `<fdset>` objects and doesn't modify its arguments.

6.25.12 Garbage collection

The garbage collector runs implicitly whenever it is necessary, and you don't usually need to worry about it. However, in case if you do need to worry, here are a few procedures you can use.

`gc` [Function]

Trigger a full GC. It may be useful if you want to reduce interference of GC in certain parts of code by calling this immediately before that.

`gc-stat` [Function]

Returns a list of lists, each inner list contains a keyword and related statistics. Current statistics include `:total-heap-size`, `:free-bytes`, `:bytes-since-gc` and `:total-bytes`.

6.25.13 Miscellaneous system calls

`sys-pause` [Function]

[POSIX] Suspends the process until it receives a signal whose action is to either execute a signal-catching function or to terminate the process. This function only returns when the signal-catching function returns. The returned value is undefined.

Note that just calling `pause()` doesn't suffice the above semantics in Scheme-level. Internally this procedure calls `sigsuspend()` with the current signal mask.

`sys-alarm` *seconds* [Function]

[POSIX] Arranges a SIGALRM signal to be delivered after *seconds*. The previous settings of the alarm clock is canceled. Passing zero to *seconds* doesn't schedule new alarm. Returns the number of seconds remaining until previously scheduled alarm was due to be delivered (or zero if no alarm is active).

`sys-sleep` *seconds :optional* (*no-retry #f*) [Function]

[POSIX] Suspends the calling thread until the specified number of seconds elapses.

Note that libc's `sleep(3)` could return before the specified interval if the calling thread receives a signal; in that case, `sys-sleep` automatically restarts `sleep(3)` again with remaining time interval (after invoking Scheme signal handlers if there's any) by default. So you can count on the thread does sleep at least the specified amount of time.

If you do want `sys-sleep` to return prematurely upon receiving a signal, you can give a true value to an optional argument *no-retry*.

The reason that we retries by default is that Gauche's GC may use signals to synchronize between threads. If GC is invoked by one thread While another thread is sleeping on `sleep(3)`, it may return prematurely. It could happen often if other threads allocate a lot, which could make `sys-sleep` unreliable.

Returns zero if it sleeps well (which is always the case if *no-retry* is false), or the number of unslept seconds if it is woke up by a signal.

To be portable across POSIX implementation, keep *seconds* less than 65536.

Some systems may be using `alarm(2)` to implement `sleep(3)`, so you shouldn't mix `sys-sleep` and `sys-alarm`.

`sys-nanosleep` *nanoseconds :optional (no-retry #f)* [Function]

[POSIX] Suspends the calling thread until the specified number of nanoseconds elapses. The argument *nanoseconds* can be a `<time>` object (see Section 6.25.9 [Time], page 269), or a real number.

The system's `nanosleep(2)` could return before the specified interval if the calling thread receives a signal; in that case, `sys-nanosleep` automatically restarts `nanosleep(2)` again with remaining time interval (after invoking Scheme signal handlers if there's any) by default. So you can count on the thread does sleep at least the specified amount of time.

The reason that we retries by default is that Gauche's GC may use signals to synchronize between threads. If GC is invoked by one thread while another thread is sleeping on `nanosleep(2)`, it may return prematurely. It could happen often if other threads allocate a lot, which could make `sys-nanosleep` unreliable.

Returns `#f` if *nanoseconds* elapsed (which is always the case if `no-retry` is `#f`), or a `<time>` object that indicates the remaining time if `sys-nanosleep` is interrupted by a signal.

```
;wait for 0.5 sec
(sys-nanosleep 500000000)

;wait for 1.3 sec
(sys-nanosleep (make <time> :second 1 :nanosecond 300000000))
```

Note: On Windows native platforms, this function is emulated using `Sleep`. The argument is rounded up to millisecond resolution, and it won't be interrupted by a signal.

`sys-random` [Function]

`sys-srandom` *seed* [Function]

A pseudo random number generator. `sys-random` returns a random number between 0 and a positive integer *rand_max*, inclusive. This is a straightforward interface to `random(3)`. If the underlying system doesn't have `random(3)`, `lrand48(3)` is used.

`sys-srandom` sets the seed of the random number generator. It uses either `srandom(3)` or `srand48(3)`, depending on the system.

The intention of these functions are to provide an off-the-stock handy random number generator (RNG) for applications that doesn't sensitive to the quality and/or speed of RNG. For serious statistics analysis, use Mersenne Twister RNG in `math.mt-random` module (see Section 12.26 [Mersenne-Twister random number generator], page 704).

`RAND_MAX` [Variable]

Bound to a positive integer that `sys-random` may return.

sys-get-osfhandle *port-or-fd* [Function]

[Windows] This procedure is only available on Windows native platforms. Returns a Windows file handle associated to the given port or integer file descriptor. Throws an error if the given argument does not have associated file handle.

6.26 Development helper API

Gauche has some basic built-in APIs to help developers to analyze the program.

6.26.1 Debugging aid

debug-print *expr* [Macro]

This macro prints *expr* in a source form, then evaluates it, then prints out the result(s), and returns them.

The special reader syntax `#?=expr` is expanded into `(debug-print expr)`. See Section 3.4 [Debugging], page 27, for the details.

debug-print-width [Parameter]

This parameter specifies the maximum width of information to be printed by `debug-print`. If the information takes more columns than the value of this parameter, it is truncated.

To show all the information, set `#f` to this parameter.

debug-funcall (*proc arg ...*) [Macro]

This macro prints the value of *arg*s right before calling *proc* and the result(s) of the call afterwards.

The special reader syntax `#?,expr` is expanded into `(debug-funcall expr)`. See Section 3.4 [Debugging], page 27, for the details.

debug-source-info *obj* [Function]

Retrieves source information attached to *obj*. The source information is returned as a list of source file name and an integer line number. If no source information is available in *obj*, `#f` is returned.

source-code *closure* [Function]

Returns the source code of *closure*, if available. Otherwise, `#f` is returned.

Currently, only the code that's directly read from Scheme source is available; if the Scheme code is precompiled, the source code isn't saved. It may be changed in future.

source-location *closure* [Function]

Returns the location (a list of filename and line number) where *closure* is defined, if available. Otherwise, `#f` is returned.

```
gosh> (use rfc.http)
gosh> (source-location http-get)
("/usr/share/gauche-0.9/0.9.5/lib/rfc/http.scm" 443)
```

disasm *closure* [Function]

Disassemble the compiled body of *closure* and print it. It may not be very useful unless you're tracking a compiler bug, or trying to tune the program to its limit.

If you're reading the disassembler output, keep in mind that the compiled code vector may have some dead code; they are produced by the jump optimization, but the compiler doesn't bother to eliminate them.

`debug-label` *obj* [Function]

This returns a string that is quasi-unique to an object *obj*. "Quasi-unique" means the label is unique to the *obj*— the same (`eq?`) objs returns the same string, and if two objs return different string they aren't `eq?` to each other— *until next GC occurs*.

This is mostly for printing out anonymous objects that doesn't have any other good way to distinguish each other. Note that uniqueness isn't guaranteed across GCs, you shouldn't use the returned value as the key to identify the objects.

6.26.2 Profiler API

These are the functions to control Gauche's built-in profiler. See Section 3.6.1 [Using profiler], page 30, for the explanation of the profiler.

Note that the profiler isn't guaranteed to work correctly yet in multi-threaded program, since the interaction between `setitimer` and threads are platform-dependent.

`profiler-start` [Function]

Starts the sampling profiler. If the profiler is already started, nothing is done.

`profiler-stop` [Function]

Stop the sampling profiler, and save the sampled data into the internal structure. If there are already saved sampled data, the newly obtained data is added to it. If the profiler isn't running, nothing is done.

`profiler-reset` [Function]

Stop the profiler if it is running. Then discard the saved sampled data.

`profiler-show` *:key sort-by max-rows* [Function]

Show the saved sampled data.

The keyword argument *sort-by* may be one of the symbols `time`, `count`, or `time-per-call`, to specify how the result should be sorted. The default is `time`.

The keyword argument *max-rows* specifies the max number of rows to be shown. If it is `#f`, all the data is shown.

`with-profiler` *thunk* [Function]

A convenience procedure. Call *thunk* with the sampling profiler running, and show the result to the current output port afterwards. Returns value(s) thunk yields. The profiler is reset after the result is shown.

You can't nest this construct; the innermost `with-profiler` will reset the profiler, invalidates any outer `with-profiler`.

7 Object system

Gauche's object system design is largely inspired by STklos, whose design has come from Tiny-CLOS. It supports multiple inheritance, multimethods, and metaobject protocol.

The type system is integrated to the object system, that is, a string is an instance of the class `<string>`, and so on.

7.1 Introduction to the object system

This section briefly explains the basic structure of Gauche's object system. It is strongly influenced by CLOS (Common-Lisp Object System). If you have experience in CLOS or related systems such as TinyCLOS, STklos or Guile's object system, you may skip to the next section.

Three concepts play the central role in CLOS-like object systems: A *class*, a *generic function*, and a *method*.

A *class* specifies a structure of object. It also defines a datatype (strictly speaking, it's not the same thing as a datatype, but let's skip the complicated part for now).

For example, a point in 2D space can be represented by x and y coordinates. A point class can be defined using `define-class` macro. In the shortest form, it can be defined like this:

```
(define-class <2d-point> () (x y))
```

(You can find the code of definitions in the examples of this section in `examples/oointro.scm` of Gauche's source distribution.)

The symbol `<2d-point>` is the name of the class, and also the global variable `<2d-point>` is bound to a class object. Surrounding a class name by `<` and `>` is just a convention; you can pass any symbol to `define-class`.

The second argument of `define-class` is a list of direct superclasses, which specifies inheritance of the class. We'll come back to it later.

The third argument of `define-class` is a list of *slots*. A slot is a storage space, usually in each object, where you can store a value. It is something similar to what is called a field or an instance variable in other object-oriented languages; but slots can be configured more than just a per-object storage space.

Now we defined a 2D point class, so we can create an instance of a point. You can pass a class to a generic function `make` to create an instance. (Don't worry about what generic function is—think it as a special type of function, just for now).

```
(define a-point (make <2d-point>))

a-point  ⇒ #<<2d-point> 0x8117570>
```

If you are using `gosh` interactively, you can use a generic function `describe` to inspect the internal of an instance. A short alias, `d`, is defined to `describe` for the convenience. (See Section 9.13 [Interactive session], page 384, for the details).

```
gosh> (d a-point)
#<<2d-point> 0x8117570> is an instance of class <2d-point>
slots:
  x         : #<unbound>
  y         : #<unbound>
```

In order to access or modify the value of the slot, you can use `slot-ref` and `slot-set!`, respectively. These names are taken from STklos.

```
(slot-ref a-point 'x)  ;; access to the slot x of a-point
  ⇒ error, since slot 'x doesn't have a value yet
```

```
(slot-set! a-point 'x 10.0)  ;; set 10.0 to the slot x of a-point

(slot-ref a-point 'x)
  ⇒ 10.0
```

Gauche also provides a shorter name, `ref`, which can also be used in srfi-17's generalized `set!` syntax:

```
(ref a-point 'x) ⇒ 10.0

(set! (ref a-point 'y) 20.0)

(ref a-point 'y) ⇒ 20.0
```

Now you can see slot values are set.

```
gosh> (d a-point)
#<<2d-point> 0x8117570> is an instance of class <2d-point>
slots:
  x         : 10.0
  y         : 20.0
```

In practice, it is usually convenient if you can specify the default value for a slot, or give values for slots when you create an instance. Such information can be specified by *slot options*. Let's modify the definition of `<2d-point>` like this:

```
(define-class <2d-point> ()
  ((x :init-value 0.0 :init-keyword :x :accessor x-of)
   (y :init-value 0.0 :init-keyword :y :accessor y-of)))
```

Note that each slot specification is now a list, instead of just a symbol as in the previous example. The list's car now specifies the slot name, and its cdr gives various information. The value after `:init-value` defines the default value of the slot. The keyword after `:init-keyword` defines the keyword argument which can be passed to `make` to initialize the slot at creation time. The name after keyword `:accessor` is bound to a generic function that can be used to access/modify the slot, instead of using `slot-ref`/`slot-set!`.

Let's see some interactive session. You create an instance of the new `<2d-point>` class, and you can see the slots are initialized by the default values.

```
gosh> (define a-point (make <2d-point>))
a-point
gosh> (d a-point)
#<<2d-point> 0x8148680> is an instance of class <2d-point>
slots:
  x         : 0.0
  y         : 0.0
```

You create another instance, this time giving initialization values by keyword arguments.

```
gosh> (define b-point (make <2d-point> :x 50.0 :y -10.0))
b-point
gosh> (d b-point)
#<<2d-point> 0x8155b80> is an instance of class <2d-point>
slots:
  x         : 50.0
  y         : -10.0
```

Accessors are less verbose than `slot-ref`/`slot-set!`, thus convenient.

```
gosh> (x-of a-point)
0.0
```

```
gosh> (x-of b-point)
50.0
gosh> (set! (y-of a-point) 3.33)
#<undef>
gosh> (y-of a-point)
3.33
```

The full list of available slot options is described in Section 7.2.1 [Defining class], page 286. At a first glance, the declarations of such slot options may look verbose. The system might have provide a static way to define init-keywords or accessor names automatically; however, CLOS-like systems prefer flexibility. Using a mechanism called metaobject protocol, you can customize how these slot options are interpreted, and you can add your own slot options as well. See Section 7.5 [Metaobject protocol], page 301, for details.

We can also have `<2d-vector>` class in similar fashion.

```
(define-class <2d-vector> ()
  ((x :init-value 0.0 :init-keyword :x :accessor x-of)
   (y :init-value 0.0 :init-keyword :y :accessor y-of)))
```

Yes, we can use the same accessor name like `x-of`, and it is effectively overloaded.

If you are familiar with mainstream object-oriented languages, you may wonder where methods are. Here they are. The following form defines a method `move-by!` of three arguments, *pt*, *dx*, *dy*, where *pt* is an instance of `<2d-point>`.

```
(define-method move-by! ((pt <2d-point>) dx dy)
  (inc! (x-of pt) dx)
  (inc! (y-of pt) dy))
```

The second argument of `define-method` macro specifies a *method specializer list*. It indicates the first argument must be an instance of `<2d-point>`, and the second and third can be any type. The syntax to call a method is just like the one to call an ordinary function.

```
gosh> (move-by! b-point 1.4 2.5)
#<undef>
gosh> (d b-point)
#<<2d-point> 0x8155b80> is an instance of class <2d-point>
slots:
  x         : 51.4
  y         : -7.5
```

You can overload the method by different specializers; here you can move a point using a vector.

```
(define-method move-by! ((pt <2d-point>) (delta <2d-vector>))
  (move-by! pt (x-of delta) (y-of delta)))
```

Specialization isn't limited to a user-defined classes. You can also specialize a method using Gauche's built-in type.

```
(define-method move-by! ((pt <2d-point>) (c <complex>))
  (move-by! pt (real-part c) (imag-part c)))
```

And here's the example session:

```
gosh> (define d-vector (make <2d-vector> :x -9.0 :y 7.25))
d-vector
gosh> (move-by! b-point d-vector)
#<undef>
gosh> (d b-point)
#<<2d-point> 0x8155b80> is an instance of class <2d-point>
slots:
```

```
  x         : 42.4
  y         : -0.25
gosh> (move-by! b-point 3+2i)
#<undef>
gosh> (d b-point)
#<<2d-point> 0x8155b80> is an instance of class <2d-point>
slots:
  x         : 45.4
  y         : -2.25
```

You see that a method is dispatched not only by its primary receiver (`<2d-point>`), but also other arguments. In fact, the first argument is no more special than the rest. In CLOS-like system a method does not belong to a particular class.

So what is actually a method? Inspecting `move-by!` reveals that it is an instance of `<generic>`, a generic function. (Note that `describe` truncates the printed value in `methods` slot for the sake of readability).

```
gosh> move-by!
#<generic move-by! (3)>
gosh> (d move-by!)
#<generic move-by! (3)> is an instance of class <generic>
slots:
  name      : move-by!
  methods   : (#<method (move-by! <2d-point> <complex>)> #<method (move-
gosh> (ref move-by! 'methods)
(#<method (move-by! <2d-point> <complex>)>
 #<method (move-by! <2d-point> <2d-vector>)>
 #<method (move-by! <2d-point> <top> <top>)>)
```

I said a generic function is a special type of function. It is recognized by Gauche as an applicable object, but when applied, it selects appropriate method(s) according to its arguments and calls the selected method(s).

What the `define-method` macro actually does is (1) to create a generic function of the given name if it does not exist yet, (2) to create a method object with the given specializers and the body, and (3) to add the method object to the generic function.

The accessors are also generic functions, created implicitly by the `define-class` macro.

```
gosh> (d x-of)
#<generic x-of (2)> is an instance of class <generic>
slots:
  name      : x-of
  methods   : (#<method (x-of <2d-vector>)> #<method (x-of <2d-point>)>)
```

In the mainstream dynamic object-oriented languages, a class has many roles; it defines a structure and a type, creates a namespace for its slots and methods, and is responsible for method dispatch. In Gauche, namespace is managed by modules, and method dispatch is handled by generic functions.

The default printed representation of object is not very user-friendly. Gauche's `write` and `display` function call a generic function `write-object` when they encounter an instance they don't know how to print. You can define its method specialized to your class to customize how the instance is printed.

```
(define-method write-object ((pt <2d-point>) port)
  (format port "[[~a, ~a]]" (x-of pt) (y-of pt)))

(define-method write-object ((vec <2d-vector>) port)
```

```
  (format port "<<~a, ~a>>" (x-of vec) (y-of vec)))
```

And what you'll get is:

```
gosh> a-point
[[0.0, 3.33]]
gosh> d-vector
<<-9.0, 7.25>>
```

If you customize the printed representation to conform srfi-10 format, and define a corresponding read-time constructor, you can make your instances to be written-out and read-back just like built-in objects. See Section 6.22.7.3 [Read-time constructor], page 229, for the details.

Several built-in functions have similar way to extend their functionality for user-defined objects. For example, if you specialize a generic function `object-equal?`, you can compare the instances by `equal?`:

```
(define-method object-equal? ((a <2d-point>) (b <2d-point>))
  (and (equal? (x-of a) (x-of b))
       (equal? (y-of a) (y-of b))))

(equal? (make <2d-point> :x 1 :y 2) (make <2d-point> :x 1 :y 2))
  ⇒ #t

(equal? (make <2d-point> :x 1 :y 2) (make <2d-point> :x 2 :y 1))
  ⇒ #f

(equal? (make <2d-point> :x 1 :y 2) 'a)
  ⇒ #f

(equal? (list (make <2d-point> :x 1 :y 2)
              (make <2d-point> :x 3 :y 4))
        (list (make <2d-point> :x 1 :y 2)
              (make <2d-point> :x 3 :y 4)))
  ⇒ #t
```

Let's proceed to more interesting examples. Think of a class `<shape>`, which is an entity that can be drawn. As a base class, it keeps common attributes such as a color and line thickness in its slots.

```
(define-class <shape> ()
  ((color     :init-value '(0 0 0) :init-keyword :color)
   (thickness :init-value 2 init-keyword :thickness)))
```

When an instance is created, `make` calls a generic function `initialize`, which takes care of initializing slots such as processing init-keywords and init-values. You can customize the initialization behavior by specializing the `initialize` method. The `initialize` method is called with two arguments, one is a newly created instance, and another is a list of arguments passed to `make`.

We define a `initialize` method for `<shape>` class, so that the created shape will be automatically recorded in a global list. Note that we don't want to replace system's `initialize` behavior completely, since we still need the init-keywords to be handled.

```
(define *shapes* '())  ;; global shape list

(define-method initialize ((self <shape>) initargs)
  (next-method)  ;; let the system to handle slot initialization
  (push! *shapes* self)) ;; record myself to the global list
```

The trick is a special method, `next-method`. It can only be used inside a method body, and calls *less specific method* of the same generic function—typically, it means you call the same method of superclass. Most object-oriented languages have the concept of calling superclass's method. Because of multiple-argument dispatching and multiple inheritance, `next-method` is a little bit more complicated, but the basic idea is the same.

So, what's the superclass of `<shape>`? In fact, all Scheme-defined class inherits a class called `<object>`. And it is `<object>`'s initialize method which takes care of slot initialization. After calling `next-method` within your `initialize` method, you can assume all the slots are properly initialized. So it is generally the first thing in your `initialize` method to call `next-method`.

Let's inspect the above code. When you call `(make <shape> args ...)`, the system allocates memory for an instance of `<shape>`, and calls `initialize` generic function with the instance and `args ...`. It is dispatched to the `initialize` method you just defined. In it, you call `next-method`, which in turn calls `<object>` class's `initialize` method. It initializes the instance with init-values and init-keywords. After it returns, you register the new `<shape>` instance to the global shape list `*shapes*`.

The `<shape>` class represents just an abstract concept of shape. Now we define some concrete drawable shapes, by *subclassing* the `<shape>` class.

```
(define-class <point-shape> (<shape>)
  ((point  :init-form (make <2d-point>) :init-keyword :point)))

(define-class <polyline-shape> (<shape>)
  ((points :init-value '() :init-keyword :points)
   (closed :init-value #f  :init-keyword :closed)))
```

Note the second argument passed to `define-class`. It indicates that `<point-shape>` and `<polyline-shape>` inherit slots of `<shape>` class, and also instances of those subclasses can be accepted wherever an instance of `<shape>` class is accepted.

The `<point-shape>` adds one slot, `point`, which contains an instance of `<2d-point>` defined in the beginning of this section. The `<polyline-shape>` class stores a list of points, and a flag, which specifies whether the end point of the polyline is connected to its starting point or not.

Inheritance is a powerful mechanism that should be used with care, or it easily result a code which is untractable ("Object-oriented programming offers a sustainable way to write spaghetti code.", as Paul Graham says in his article "The Hundred-Year Language"). The rule of thumb is to make a subclass when you need a subtype. The inheritance of slots is just something that comes with, but it shouldn't be the main reason to do subclassing. You can always "include" the substructure, as is done in `<point-shape>` class.

There appeared a new slot option in `<point-shape>` class. The `:init-form` slot option specifies the default value of the slot when init-keyword is not given to `make` method. However, unlike `:init-value`, with which the value is evaluated at the time the class is defined, the value with `:init-form` is evaluated when the system actually needs the value. So, in the `<point-shape>` instance, the default `<2d-point>` instance is only created if the `<point-shape>` instance is created without having `:point` init-keyword argument.

A shape may be drawn in different formats for different devices. For now, we just consider a PostScript output. To make the `draw` method polymorphic, we define a postscript output device class, `<ps-device>`.

```
(define-class <ps-device> () ())
```

Then we can write a `draw` method, specialized for both `<shape>` and `<ps-device>`.

```
(define-method draw ((self <shape>) (device <ps-device>))
  (format #t "gsave\n")
  (draw-path self device)
```

```
    (apply format #t "~a ~a ~a setrgbcolor\n" (ref self 'color))
    (format #t "~a setlinewidth\n" (ref self 'thickness))
    (format #t "stroke\n")
    (format #t "grestore\n"))
```

In this code, the *device* argument isn't used within the method body. It is just used for method dispatching. If we eventually have different output devices, we can add a `draw` method that is specialized for such devices.

The above `draw` method does the common work, but actual drawing must be done in specialized way for each subclasses.

```
(define-method draw-path ((self <point-shape>) (device <ps-device>))
  (apply format #t "newpath ~a ~a 1 0 360 arc closepath\n"
         (point->list (ref self 'point))))

(define-method draw-path ((self <polyline-shape>) (device <ps-device>))
  (let ((pts (ref self 'points)))
    (when (>= (length pts) 2)
      (format #t "newpath\n")
      (apply format #t "~a ~a moveto\n" (point->list (car pts)))
      (for-each (lambda (pt)
                  (apply format #t "~a ~a lineto\n" (point->list pt)))
                (cdr pts))
      (when (ref self 'closed)
        (apply format #t "~a ~a lineto\n" (point->list (car pts))))
      (format #t "closepath\n"))))

;; utility method
(define-method point->list ((pt <2d-point>))
  (list (x-of pt) (y-of pt)))
```

Finally, we do a little hack. Let `draw` method work on the list of shapes, so that we can draw multiple shapes within a page in batch.

```
(define-method draw ((shapes <list>) (device <ps-device>))
  (format #t "%%\n")
  (for-each (cut draw <> device) shapes)
  (format #t "showpage\n"))
```

Then we can write some simple figures . . . .

```
(use srfi-1)      ;; for iota
(use math.const)  ;; for constant pi

(define (shape-sample)

  ;; creates 5 corner points of pentagon
  (define (make-corners scale)
    (map (lambda (i)
           (let ((pt (make <2d-point>)))
             (move-by! pt (make-polar scale (* i 2/5 pi)))
             (move-by! pt 200 200)
             pt))
         (iota 5)))

  (set! *shapes* '())  ;; clear the shape list
```

```
(let* ((corners (make-corners 100)))
  ;; a pentagon in green
  (make <polyline-shape>
    :color '(0 1 0) :closed #t
    :points corners)
  ;; a star-shape in red
  (make <polyline-shape>
    :color '(1 0 0) :closed #t
    :points (list (list-ref corners 0)
                  (list-ref corners 2)
                  (list-ref corners 4)
                  (list-ref corners 1)
                  (list-ref corners 3)))
  ;; put dots in each corner of the star
  (for-each (cut make <point-shape> :point <>)
            (make-corners 90))
  ;; draw the shapes
  (draw *shapes* (make <ps-device>)))
)
```

The function `shape-sample` writes out a PostScript code of simple drawing to the current output port. You can write it out to file by the following expression, and then view the result by PostScript viewer such as GhostScript.

```
(with-output-to-file "oointro.ps" shape-sample)
```

7.2 Class

In this section, a class in Gauche is explained in detail.

7.2.1 Defining class

To define a class, use a macro `define-class`.

`define-class` *name supers* (*slot-spec* . . .) *option* . . . [Macro]

Creates a class object according to the arguments, and globally bind it to a variable *name*. This macro should be used at toplevel.

Supers is a list of direct superclasses from which this class inherits. You can use multiple inheritance. All Scheme-defined classes implicitly inherits `<object>`. It is implicitly added to the right of *supers* list, so you don't need to specify it. See Section 7.2.2 [Inheritance], page 288, for the details about inheritance.

Slot-spec is a specification of a "slot", sometimes known as a "field" or an "instance variable" (but you can specify "class variable" in *slot-spec* as well). The simplest form of *slot-spec* is just a symbol, which names the slot. Or you can give a list, whose first element is a symbol and whose rest is an interleaved list of keywords and values. The list form not only defines a name of the slot but specifies behavior of the slot. It is explained below.

Finally, *option* . . . is an interleaved list of keywords and values, specifies how class object should be created. This macro recognizes one keyword, `:metaclass`, whose corresponding value is used for metaclass (class that instantiates another class). Other options are passed to the `make` method to create the class object. See Section 7.5.1 [Class instantiation], page 301, for the usage of metaclass.

If a slot specification is a list, it should be in the following form:

```
(slot-name :option1 value1 :option2 value2 ...)
```

Each keyword (`option1` etc.) gives a *slot option*. By default, the following slot options are recognized. You can add more slot options by defining metaclass.

`:allocation`
: Specifies an allocation type of this slot, which specifies how the value for this slot is stored. The following keyword values are recognized by the standard class. A programmer can define his own metaclass to extend the class to recognize other allocation types.

  `:instance`
  : A slot is allocated for each instance, so that every instance can have distinct value. This realizes so-called "instance variable" behavior. If `:allocation` slot option is omitted, this is the default.

  `:class`
  : A slot is allocated in this class object, so that every instance will share the same value for this slot. This realizes so-called "class variable" behavior. The slot value is also shared by all subclasses (unless a subclass definition shadows the slot).

  `:each-subclass`
  : Similar to `class` allocation, but a slot is allocated for each class; that is, it is shared by every instance of the class, but not shared by the instances of its subclasses.

  `:virtual`
  : No storage is allocated for this type of slot. Accessing the slot calls procedures given in `:slot-ref` and `:slot-set!` options described below. In other words, you can make a procedural slot. If a slot's allocation is specified as virtual, at least `:slot-ref` option has to be specified as well, or `define-class` raises an error.

  `:builtin`
  : This type of allocation only appears in built-in classes, and you can't specify it in Scheme-defined class.

`:init-keyword`
: A keyword value given to this slot option can be used to pass an initial value to `make` method when an instance is created.

`:init-value`
: Gives an initial value of the slot, if the slot is not initialized by the keyword argument at the creation time. The value is evaluated when `define-class` is evaluated.

`:init-form`
: Like `init-value`, but the value given is wrapped in a thunk, and evaluated each time when the value is required. If both `init-value` and `init-form` are given, `init-form` is ignored. Actually, `:init-form` *expr* is converted to `:init-thunk (lambda ()` *expr*`)` by `define-class` macro.

`:initform`
: A synonym of `init-form`. This is kept for compatibility to STk, and shouldn't be used in the new code.

`:init-thunk`
: Gives a thunk, which will be evaluated to obtain an initial value of the slot, if the slot is not initialized by the keyword argument at the creation time. To give a value to `:init-form` is equivalent to give `(lambda () value)` to `:init-thunk`.

`:getter`
: Takes a symbol, and a getter method is created and bound to the generic function of that name. The getter method takes an instance of the class and returns the value of the slot.

:setter Takes a symbol, and a setter method is created and bound to the generic function of that name. The setter method takes an instance of the class and a value, and sets the value to the slot of the instance.

:accessor
Takes a symbol, and create two methods; a getter method and a setter method. A getter method is bound to the generic function of the given name, and a setter method is added as the *setter* of that generic function (see Section 4.4 [Assignments], page 47, for generic setters).

:slot-ref
Specifies a value that evaluates to a procedure which takes one argument, an instance. This slot option must be specified if the allocation of the slot is virtual. Whenever a program tries to get the value of the slot, either using slot-ref or the getter method, the specified procedure is called, and its result is returned as the value of the slot. The procedure can return an undef value (the return value of undefined) to indicate the slot doesn't have a value. If the slot allocation is not virtual this slot option is ignored.

:slot-set!
Specifies a value that evaluates to a procedure which takes two arguments, an instance and a value. Whenever a program tries to set the value of the slot, either using slot-set! or the setter method, the specified procedure is called with the value to be set. If the slot allocation is not virtual this slot option is ignored. If this option isn't given to a virtual slot, the slot becomes read-only.

:slot-bound?
Specifies a value that evaluates to a procedure which takes one argument, an instance. This slot option is only meaningful when the slot allocation is virtual. Whenever a program tries to determine whether the slot has a value, this procedure is called. It should return a true value if the slot has a value, or #f otherwise. If this slot option is omitted for a virtual slot, the system calls the procedure given to slot-ref instead, and see whether its return value is #<undef> or not.

7.2.2 Inheritance

Inheritance has two roles. First, you can *extend* the existing class by adding more slots. Second, you can *specialize* the methods related to the existing class so that those methods will do a little more specific task than the original methods.

Let's define some terms. When a class <T> inherits a class <S>, we call <T> a *subclass* of <S>, and <S> a *superclass* of <T>. This relation is transitive: <T>'s subclasses are also <S>'s subclasses, and <S>'s superclasses are also <T>'s superclasses. Specifically, if <T> directly inherits <S>, that is, <S> appeared in the superclass list when <T> is defined, then <S> is a *direct superclass* of <T>, and <T> is a *direct subclass* of <S>.

When a class is defined, it and its superclasses are ordered from subclasses to superclasses, and a list of classes is created in such order. It is called *class precedence list*, or CPL. Every class has its own CPL. A CPL of a class always begins with the class itself, and ends with <top>.

You can query a class's CPL by a procedure class-precedence-list:

```
gosh> (class-precedence-list <boolean>)
(#<class <boolean>> #<class <top>>)
gosh> (class-precedence-list <string>)
(#<class <string>> #<class <sequence>> #<class <collection>> #<class <top>>)
```

As you see, all classes inherits a class named <top>. Some built-in classes have several abstract classes in its CPL between itself and <top>; the above example shows <string> class inherits <sequence> and <collection>. That means a string can behave both as a sequence and a collection.

```
gosh> (is-a? "abc" <string>)
#t
gosh> (is-a? "abc" <sequence>)
#t
gosh> (is-a? "abc" <collection>)
#t
```

How about inheritance of Scheme-defined classes? If there's only single inheritance, its CPL is straightforward: you can just follow the class's super, its super's super, its super's super's super, . . ., until you reach <top>. See the example:

```
gosh> (define-class <a> () ())
<a>
gosh> (define-class <b> (<a>) ())
<b>
gosh> (class-precedence-list <b>)
(#<class <b>> #<class <a>> #<class <object>> #<class <top>>)
```

Scheme-defined class always inherits <object>. It is automatically inserted by the system.

When multiple inheritance is involved, a story becomes a bit complicated. We have to merge multiple CPLs of the superclasses into one CPL. It is called *linearization*, and there are several known linearization strategies. By default, Gauche uses an algorithm called *C3 linearization*, which is consistent with the local precedence order, monotonicity, and the extended precedence graph. We don't go into the details here; as a general rule, the order of superclasses in a class's CPL is always consistent to the order of direct superclasses of the class, the order of CPL of each superclasses, and the order of direct superclasses of each superclass, and so on. See [Dylan], page 829, for the precise description.

If a class inherits superclasses in a way that its CPL can't be constructed with satisfying consistencies, an error is reported.

Here's a simple example of multiple inheritance.

```
(define-class <grid-layout> () ())

(define-class <horizontal-grid> (<grid-layout>) ())

(define-class <vertical-grid> (<grid-layout>) ())

(define-class <hv-grid> (<horizontal-grid> <vertical-grid>) ())

(map class-name (class-precedence-list <hv-grid>))
 ⇒ (<hv-grid> <horizontal-grid> <vertical-grid>
     <grid-layout> <object> <top>)
```

Note that the order of direct superclasses of <hv-grid> (<horizontal-grid> and <vertical-grid>) is kept.

The following is a little twisted example:

```
(define-class <pane> () ())

(define-class <scrolling-mixin> () ())

(define-class <scrollable-pane> (<pane> <scrolling-mixin>) ())
```

```
(define-class <editing-mixin> () ())

(define-class <editable-pane> (<pane> <editing-mixin>) ())

(define-class <editable-scrollable-pane>
   (<scrollable-pane> <editable-pane>) ())

(map class-name (class-precedence-list <editable-scrollable-pane>))
 ⇒ (<editable-scrollable-pane> <scrollable-pane>
     <editable-pane> <pane> <scrolling-mixin> <editing-mixin>
     <object> <top>)
```

Once the class precedence order is determined, the slots of defined class is calculated as follows: the slot definitions are collected in the direction from superclasss to subclass in CPL. If a subclass has a slot definition of the same name of the one in superclass, then the slot definition of the subclass is taken and superclass's is discarded. Suppose a class `<S>` defines slots `a`, `b`, and `c`, a class `<T>` defines slots `c`, `d`, and `e`, and a class `<U>` defines slots `b` and `e`. When `<U>`'s CPL is `(<U> <T> <S> <object> <top>)`, then `<U>`'s slots is calculated as the chart below; that is, `<U>` gets five slots, of which `b` and `e`'s definitions come from `<U>`'s definitions, `c` and `d`'s come from `<T>`, and `a`'s comes from `<S>`.

```
    CPL      | slot definitions
             |  () indicates shadowed slot
  -----------+-------------------
    <top>    |
    <object> |
    <S>      | a  (b) (c)
    <T>      |         c   d  (e)
    <U>      |     b           e
  -----------+--------------------
  <U>'s slots| a   b   c   d   e
```

You can get a list of slot definitions of a class object using `class-slots` function.

Note that the behavior described above is mere a default behavior. You can customize how the CPL is computed, or how slot definitions are inherited, by defining metaclass. For example, you can write a metaclass that allows you to merge slot options of the same slot names, instead of the one shadowing the other. Or you can write a metaclass that forbids a subclass shadows the superclass's slot.

7.2.3 Class object

What is a class? In Gauche, a class is just an object that implements a specific feature: to instantiate an object. Because of that, you can introspect the class by just looking into the slot values. There are some procedures provided for the convenience of such introspection. Note that if those procedures return a list, it belongs to the class and you shouldn't modify it.

`class-name` *class* [Function]

Returns the name of *class*.

```
(class-name <string>) ⇒ <string>
```

`class-precedence-list` *class* [Function]

Returns the class precedence list of *class*.

```
(class-precedence-list <string>)
  ⇒ (#<class <string>>
      #<class <sequence>>
```

```
            #<class <collection>>
            #<class <top>>)
```

`class-direct-supers` *class* [Function]

Returns a list of direct superclasses of *class*. A direct superclass is a class from which *class* inherits directly.

```
(class-direct-supers <string>)
  ⇒ (#<class <sequence>>)
```

`class-direct-subclasses` *class* [Function]

Returns a list of direct subclasses of *class*. A direct subclass is a class that directly inherits *class*. If `<T>` is a direct subclass of `<S>`, then `<S>` is a direct superclass of `<T>`.

`class-slots` *class* [Function]

Returns a list of *slot definitions* of *class*. A slot definition is a list whose car is the name of the slot and whose cdr is a keyword-value list that specifies slot options. You can further inspect a slot definition to know what characteristics the slot has. See Section 7.2.4 [Slot definition object], page 291, for the details.

The standard way to get a list of slot names of a given class is `(map slot-definition-name (class-slots class))`.

`class-slot-definition` *class slot-name* [Function]

Returns a slot definition of a slot specified by *slot-name* in a class *class*. If *class* doesn't have a named slot, `#f` is returned.

`class-direct-slots` *class* [Function]

Returns a list of slot definitions that are directly defined in this class (i.e. not inherited from superclasses). This information is used to calculate slot inheritance during class initialization.

`class-direct-methods` *class* [Function]

Returns a list of methods that has *class* in its specializer.

`class-slot-accessor` *class slot-name* [Function]

Returns a slot accessor object of the slot specified by *slot-name* in *class*. A slot accessor object is an internal object that encapsulates the information how to access, modify, and initialize the given slot.

You don't usually need to deal with slot accessor objects unless you are defining some special slots using metaobject protocol.

7.2.4 Slot definition object

A slot definition object, returned by `class-slots`, `class-direct-slots` and `class-slot-definition`, keeps information about a slot. Currently Gauche uses a list to represent the slot definition, as STklos and TinyCLOS do. However, it is not guaranteed that Gauche keeps such a structure in future; you should use the following dedicated accessor methods to obtain information of a slot definition object.

`slot-definition-name` *slot-def* [Function]

Returns the name of a slot given by a slot definition object *slot-def*.

`slot-definition-options` *slot-def* [Function]

Returns a keyword-value list of slot options of *slot-def*.

`slot-definition-allocation` *slot-def* [Function]

Returns the value of `:allocation` option of *slot-def*.

`slot-definition-getter` *slot-def* [Function]
`slot-definition-setter` *slot-def* [Function]
`slot-definition-accessor` *slot-def* [Function]

Returns the value of `:getter`, `:setter` and `:accessor` slot options of *slot-def*, respectively.

`slot-definition-option` *slot-def option :optional default* [Function]

Returns the value of slot option *option* of *slot-def*. If there's no such an option, *default* is returned if given, or an error is signaled otherwise.

7.2.5 Class redefinition

If the specified class name is bound to a class when `define-class` is used, it is regarded as *redefinition* of the original class.

Redefinition of a class means the following operations:

- A new class object is created based on the new definition, and bound to the variable given to `define-class`.
- Methods defined on the original class (i.e. methods that have the original class in their specializers) are changed so that they are defined on the new class.
- The direct-subclasses link of the direct superclasses of the original class is modified so that they will point to the new class.
- All the subclasses of the original class are redefined recursively so that they reflect the changes of the class. Each class remembers its initialization arguments, and each redefined subclass gets the same initialization arguments as the original subclass.
- The original class is marked *redefined*.

Note that the original class and the new class are different objects. The original class object remembers which variable in which module it is originally bound, and replaces the binding to a new class. If you keep the direct reference to the original class somewhere else, it still refers to the original class; you might want to take extra care. You can customize class redefinition behavior by defining the `class-redefinition` method; see Section 7.5 [Metaobject protocol], page 301, for the details.

If there are instances of the original class, such instances are automatically *updated* when it is about to be accessed or modified via `class-of`, `is-a?`, `slot-ref`, `slot-set!`, `ref`, a getter method, or a setter method.

Updating an instance means that the class of the instance is changed (from the old class to the new class). By default, the values of the slots that are common in the original class and the new class are carried over, and the slots added by the new class are initialized according to the slot specification of the new class, and the values of the slots that are removed from the original class are discarded. You can customize this behavior by writing the `change-class` method. See Section 7.3.3 [Changing classes], page 297, for the details.

Notes on thread safety

Class redefinition process is non-local operation with full of side-effects. It is difficult to guarantee that two threads safely run class redefinition protocol simultaneously. So Gauche uses a process-wide lock to limit only one thread to enter the class redefinition protocol at a time.

If a thread tries to redefine a class while another thread is in the redefinition protocol, the thread is blocked, even if it is redefining a class different from the one that are being redefined; because redefinition affects all the subclasses, and all the methods and generic functions that are related to the class and subclasses, it is not trivial to determine two classes are completely independent or not.

If a thread tries to access an instance whose class is being redefined by another thread, also the thread is blocked until the redefinition is finished.

Note that the instance update protocol isn't serialized. If two threads try to access an instance whose class has been redefined, both trigger the instance update protocol, which would cause an undesired race condition. It is the application's responsibility to ensure such a case won't happen. It is natural since the instance access isn't serialized by the system anyway. However, an extra care is required to have mutex within an instance; just accessing the mutex in it may trigger the instance update protocol.

Notes on compatibility

Class redefinition protocols subtlety differ among CLOS-like Scheme systems. Gauche's is very similar to STklos's, except that STklos 0.56 doesn't replace bindings of redefined subclasses, and also it doesn't remember initialization arguments so the redefined subclass may lose some of the information that the original subclass has. Guile's object system swaps identities of the original class and the redefined class at the end of class redefinition protocol, so the reference to the original class object will turn to the redefined class. As far as the author knows, class redefinition is not thread-safe in both STklos 0.56 and Guile 1.6.4.

7.2.6 Class definition examples

Let's see some examples. Suppose you are defining a graphical toolkit. A `<window>` is a rectangle region on the screen, so it has width and height. It can be organized hierarchically, i.e. a window can be placed within another window; so it has a pointer to the parent window. And we specify the window's position, x, y, by the coordinate relative to its parent window. Finally, we create a "root" window that covers entire screen. It also serves the default parent window. So far, what we get is something like this:

```
;; The first version
(define-class <window> ()
  (;; Pointer to the parent window.
   (parent      :init-keyword :parent :init-form *root-window*)
   ;; Sizes of the window
   (width       :init-keyword :width  :init-value 1)
   (height      :init-keyword :height :init-value 1)
   ;; Position of the window relative to the parent.
   (x           :init-keyword :x :init-value 0)
   (y           :init-keyword :y :init-value 0)
   ))

(define *screen-width* 1280)
(define *screen-height* 1024)

(define *root-window*
  (make <window> :parent #f :width *screen-width* :height *screen-height*))
```

Note the usage of `:init-value` and `:init-form`. When the `<window>` class is defined, we haven't bound `*root-window*` yet, so we can't use `:init-value` here.

```
gosh> *root-window*
#<<window> 0x80db1d0>
gosh> (define window-a (make <window> :width 100 :height 100))
window-a
gosh> (d window-a)
#<<window> 0x80db1b0> is an instance of class <window>
slots:
  parent    : #<<window> 0x80db1d0>
  width     : 100
```

```
  height    : 100
  x         : 0
  y         : 0
gosh> (define window-b
        (make <window> :parent window-a :width 50 :height 20 :x 10 :y 5))
window-b
gosh> (d window-b)
#<<window> 0x80db140> is an instance of class <window>
slots:
  parent    : #<<window> 0x80db1b0>
  width     : 50
  height    : 20
  x         : 10
  y         : 5
```

If you're like me, you don't want to expose a global variable such as `*root-window*` for users of your toolkit. One way to encapsulate it (to certain extent) is to keep the pointer to the root window in a class variable. Add the following slot option to the definition of `<window>`, and the slot `root-window` of the `<window>` class refers to the same storage space.

```
(define-class <window> ()
  (...
   ...
   (root-window :allocation :class)
   ...))
```

You can use `slot-ref` and `slot-set!` on an instance of `<window>`, or use `class-slot-ref` and `class-slot-set!` on the `<window>` class itself, to get/set the value of the `root-window` slot.

The users of the toolkit may want to get the absolute position of the window (the coordinates in the root window) instead of the relative position. You may provide virtual slots that returns the absolute positions, like the following:

```
(define-class <window> ()
  (...
   ...
   (root-x :allocation :virtual
           :slot-ref  (lambda (o)
                        (if (ref o 'parent)
                            (+ (ref (ref o 'parent) 'root-x)
                               (ref o 'x))
                            (ref o 'x)))
           :slot-set! (lambda (o v)
                        (set! (ref o 'x)
                              (if (ref o 'parent)
                                  (- v (ref (ref o 'parent) 'root-x))
                                  v)))
           )
    ...))
```

Whether providing such interface via methods or virtual slots is somewhat a matter of taste. Using virtual slots has an advantage of being able to hide the change of implementation, i.e. you can change to keep `root-x` in a real slot and make `x` a virtual slot later without breaking the code using `<window>`. (In the mainstream object-oriented languages, such kind of "hiding implementation" is usually achieved by hiding instance variables and exposing methods. In

Gauche and other CLOS-like systems, slots are always visible to the users, so the situation is a bit different.

7.3 Instance

In this section, we explain how to create and use an instance.

7.3.1 Creating instance

Using class object, you can create an instance of the class by a generic function `make`. A specialized method for standard `<class>` is defined:

`make` [Generic Function]
`make` (*class* <*class*>) *arg* . . . [Method]

Creates an instance of *class* and returns it. *Arg* . . . is typically a keyword-value list to initialize the instance.

Conceptually, the default `make` method is defined as follows:

```
(define-method make ((class <class>) . initargs)
  (let ((obj (allocate-instance class initargs)))
    (initialize obj initargs)
    obj))
```

That is, first it allocates memory for *class*'s instance, then initialize it with the `initialize` method.

`allocate-instance` [Generic Function]
`allocate-instance` (*class* <*class*>) *initargs* [Method]

Returns a newly-allocated uninitialized instance of *class*.

`initialize` [Generic Function]
`initialize` (*obj* <*object*>) *initargs* [Method]

The default initialize method for `<object>` works as follows:

- For each initializable slot of the class
  - If (the slot has the :init-keyword slot option AND the keyword appears in *initargs*): Then the corresponding value is used to initialize the slot
  - Else if the slot has :init-value slot option: Then the value given to the slot option is used to initialize the slot
  - Else if the slot has :init-thunk slot option: Then the thunk is called, and the returned value is used to initialize the slot.
  - Else: The slot is left unbound.

Among the default slot allocation classes, only instance-allocated slots are initializable and are handled by the above sequence. Class-allocated slots (e.g. its slot allocation is either `:class` or `:each-subclass`) are initialized when the class object is created, if `:init-value` or `:init-form` slot option is given. Virtual slots aren't initialized at all.

An user-defined allocation class can be configured either initializable or not initializable; see Section 7.5 [Metaobject protocol], page 301, for the details.

If you specialize `initialize` method, make sure to call `next-method` so that the slots are properly initialized by the default sequence, before accessing any slot of the newly created instance.

Typically you specialize `initialize` method for your class to customize how the instance is initialized.

It is not common to specialize `allocate-instance` method. However, knowing that how `make` works, you can specialize `make` itself to avoid allocation of instance in some circumstances (e.g. using pre-allocated instances).

7.3.2 Accessing instance

Standard accessors

`slot-ref` *obj slot* [Function]

Returns a value of the slot *slot* of object *obj*.

If the specified slot is not bound to any value, a generic function `slot-unbound` is called with three arguments, *obj*'s class, *obj*, and *slot*. The default behavior of `slot-unbound` is to signal an error.

If the object doesn't have the specified slot, a generic function `slot-missing` is called with three arguments, *obj*'s class, *obj*, and *slot*. The default behavior of `slot-missing` is to signal an error.

`slot-set!` *obj slot value* [Function]

Alters the value of the slot *slot* of object *obj* to the value *value*.

If the object doesn't have the specified slot, a generic function `slot-missing` is called with four arguments, *obj*'s class, *obj*, *slot*, *value*.

`slot-bound?` *obj slot* [Function]

Returns true if object *obj*'s slot *slot* is bound, otherwise returns false.

If the object doesn't have the specified slot, a generic function `slot-missing` is called with three arguments, *obj*'s class, *obj*, *slot*.

`slot-exists?` *obj slot* [Function]

Returns true if *obj* has the slot named *slot*.

`slot-push!` *obj slot value* [Function]

This function implements the common idiom. It can be defined like the following code (but it may be optimized in the future versions).

```
(define (slot-push! obj slot value)
  (slot-set! obj slot (cons value (slot-ref obj slot))))
```

`slot-pop!` *obj slot :optional fallback* [Function]

Reverse operation of `slot-push!`. If the value of *slot* of *obj* is a pair, removes its car and returns the removed item.

When the value of *slot* is not a pair, or the *slot* is unbound, *fallback* is returned if it is provided, otherwise an error is signaled.

`ref` (*obj* <*object*>) (*slot* <*symbol*>) [Method]

`(setter ref)` (*obj* <*object*>) (*slot* <*symbol*>) *value* [Method]

These methods just calls `slot-ref` and `slot-set!`, respectively. They are slightly less efficient than directly calling `slot-ref` and `slot-set!`, but more compact in the program code.

Fallback methods

`slot-unbound` [Generic Function]

`slot-unbound` (*class* <*class*>) *obj slot* [Method]

This generic function is called when an unbound slot value is retrieved. The return value of this generic function will be returned to the caller that tried to get the value.

The default method just signals an error.

`slot-missing` [Generic Function]
`slot-missing` (*class* <*class*>) *obj slot :optional value* [Method]

This generic function is called when a non-existent slot value is retrieved or set. The return value of this generic function will be returned to the caller that tried to get the value.

The default method just signals an error.

Special accessors

`current-class-of` *obj* [Function]

Returns a class metaobject of *obj*. If *obj*'s class has been redefined, but *obj* is not updated for the change, then this procedure returns the original class of *obj* without updating *obj*.

You need this procedure in rare occasions, such as within `change-class` method, in which you don't want to trigger updating *obj* (which would cause infinite loop).

`class-slot-ref` *class slot-name* [Function]
`class-slot-set!` *class slot-name obj* [Function]
`class-slot-bound?` *class slot-name obj* [Function]

When slot's `:allocation` option is either `:class` or `:each-subclass`, these procedures allow you to get/set the value of the slot without having an instance.

`slot-ref-using-class` (*class* <*class*>) (*obj* <*object*>) *slot-name* [Method]
`slot-set-using-class!` (*class* <*class*>) (*obj* <*object*>) *slot-name value* [Method]
`slot-bound-using-class?` (*class* <*class*>) (*obj* <*object*>) *slot-name* [Method]

Generic function version of `slot-ref`, `slot-set!` and `slot-bound?`. *Class* must be the class of *obj*.

Besides being generic, these functions are different from their procedural versions that they don't trigger class redefinition when *obj*'s class has been redefined (i.e. in which case, *class* should be the original class of *obj*).

Note: Unlike CLOS, `slot-ref` etc. don't call the generic function version in it, so you can't customize the behavior of `slot-ref` by specializing `slot-ref-using-class`. So the primary purpose of those generic functions are to be used within `change-class` method; especially, `slot-ref` etc. can't be used during *obj*'s being redefined, since they trigger class redefinition again (see Section 7.3.3 [Changing classes], page 297, for details).

7.3.3 Changing classes

Class change protocol

An unique feature of CLOS-family object system is that you can change classes of an existing instance. The two classes doesn't need to be related; you can change a sewing machine into an umbrella, if you like.

`change-class` [Generic Function]
`change-class` (*obj* <*object*>) (*new-class* <*class*>) [Method]

Changes an object *obj*'s class to *new-class*. The default method just calls `change-object-class` procedure.

`change-object-class` *obj orig-class new-class* [Function]

Changes an object *obj*'s class from *orig-class* to *new-class*. This isn't a generic function—changing object's class needs some secret magic, and this procedure encapsulates it.

The precise steps of changing class are as follow:

1. A new instance of *new-class* is allocated by `allocate-instance`.

2. For each slot of *new-class*:
   1. If the slot also exists in *old-class*, and is bound in *obj*, the value is retrieved from *obj* and set to the new instance. (The slot is *carried over*).
   2. Otherwise, the slot of the new instance is initialized by standard slot initialization protocol, as described in Section 7.3.1 [Creating instance], page 295.
3. Finally, the content of the new instance is *transplanted* to the *obj*—that is, *obj* becomes the instance of *new-class* without changing its identity.

Note that `initialize` method of *new-class* isn't called on *obj*. If you desire, you can call it by your own `change-class` method.

`Change-object-class` returns *obj*.

Usually a user is not supposed to call `change-object-class` directly. Instead, she can define a specialized `change-class`. For example, if she wants to carry over the slot `x` of old class to the slot `y` of new class, she may write something like this:

```
(define-method change-class ((obj <old-class>) <new-class>)
  (let ((old-val (slot-ref obj 'x)))
    (next-method)              ;; calls default change-class
    (slot-set! obj 'y old-val)  ;; here, obj's class is already <new-class>.
    obj))
```

Customizing instance update

Updating an instance for a redefined class is also handled as class change. When an object is accessed via normal slot accessor/modifier, its class is checked whether it has been redefined. And if it has indeed been redefined, `change-class` is called with the redefined class as *new-class*; that is, updating an instance is regarded as changing object's class from the original one to the redefined one.

By specializing `change-class`, you can customize the way an instance is updated for a redefined class. However, you need a special care to write `change-class` for class redefinition.

First, the redefinition changes global binding of the class object. So you need to keep the reference to the old class before redefining the class, and use the old class to specialize `change-class` method:

```
;; save old <myclass>
(define <old-myclass> <myclass>)

;; redefine <myclass>
(define-class <myclass> ()
  ...)

;; define customized change-class method
(define-method change-class ((obj <old-myclass>) <myclass>)
  ...
  (next-method)
  ...)
```

Next, note that the above `change-class` method may be triggered implicitly when you access to *obj* via `slot-ref`, `slot-set!`, `class-of`, etc. If you use such procedures like `slot-ref` on *obj* again within `change-class`, it would trigger the instance update protocol recursively, which would cause an infinite loop. You can only use the methods that doesn't trigger instance update, that is, `slot-ref-using-class`, `slot-set-using-class!`, `slot-bound-using-class?` and `current-class-of`.

If you want to carry over a slot whose value is calculated procedurally, such as a virtual slot, then `slot-ref` etc. might be called implicitly on *obj* during calculating the slot value. Actually `change-object-class` has a special protection to detect such a recursion. If that happens, `change-object-class` gives up to retrieve the slot value and just initializes the slot of the new instance as if the old slot were unbound.

Customizing instance update is highly tricky business, although very powerful. You can find some nontrivial cases in the test program of Gauche source code; take a look at `test/object.scm`.

7.4 Generic function and method

Defining methods

`define-generic` *name :key class* [Macro]

Creates a generic function and bind it to *name*.

You don't usually need to use this, since the `define-method` macro implicitly creates a generic function if it doesn't exist yet.

You can pass a subclass of `<generic>` to the *class* keyword argument so that the created generic function will be the instance of the passed class, instead of the default `<generic>` class. It is useful when you defined a subclass of `<generic>` to customize generic function application behavior.

`define-method` *name* [*qualifier* . . .] *specs body* [Macro]

Defines a method whose name is *name*. If there's already a generic function object globally bound to *name*, the created method is added to the generic function. If *name* is unbound, or bound to an object except a generic function, then a new generic function is created, bound to *name*, then a new method is added to it.

The name can be followed by optional *qualifiers*, each of which is a keyword. Currently, only the following qualifier is valid.

`:locked` Declares that you won't redefine the method with the same specifiers. Attempt to redefine it will raise an error. (You can still define methods with different specifiers.)

Most methods concerning basic operations on built-in objects are locked, for redefining them would case Gauche's infrastracture unstable. It also allows Gauche to perform certain optimizations.

Specs specifies the arguments and their types for this method. It's like the argument list of lambda form, except you can specify the type of each argument.

```
specs : ( arg ... )
      | ( arg ... . symbol )
      | ( arg ... extended-spec ...)
      | symbol

arg   : ( symbol class )
      | symbol
```

Class specifies the class that the argument has to belong to. If *arg* is just a symbol, it is equivalent to `(arg <top>)`. You can't specify the type for the "rest" argument, for it is always bound to a list.

You can use extended argument specifications such as `:optional`, `:key` and `:rest` as well. (See Section 4.3 [Making procedures], page 42, for the explanation of extended argument specifications). Those extended arguments are treated as if a single "rest" argument in terms of dispatching; they aren't used for method dispatch, and you can't specify classes for these optional and keyword arguments.

The list of classes of the argument list is called *method specializer list*, based on which the generic function will select appropriate methods(s). Here are some examples of *specs* and the corresponding specializer list (note that the rest argument isn't considered as a part of specializer list; we know it's always a list.) The `optional` item indicates whether the method takes rest arguments or not.

```
specs:        ((self <myclass>) (index <integer>) value)
specializers: (<myclass> <integer> <top>)
optional:     #f

specs:        (obj (attr <string>))
specializers: (<top> <string>)
optional:     #f

specs:        ((self <myclass>) obj . options)
specializers: (<myclass> <top>)
optional:     #t

specs:        ((self <myclass>) obj :optional (a 0) (b 1) :key (c 2))
specializers: (<myclass> <top>)
optional:     #t

specs:        args
specializers: ()
optional:     #t
```

If you define a method on *name* whose specializer list, and whether it takes rest arguments, match with one in the generic function's methods, then the existing method is replaced by the newly defined one, unless the original method is locked.

Note: If you're running Gauche with keyword-symbol integrated mode (see Section 6.8.1 [Keyword and symbol integration], page 141), there's an ambiguity if you specify a keyword as the sole *specs* (to receive entire arguments in a single variable). Gauche parses keywords following *name* as qualifiers, so avoid using a keyword as such a variable.

Applying generic function

When a generic function is applied, first it selects methods whose specializer list matches the given arguments. For example, suppose a generic function `foo` has three methods, whose specializer lists are `(<string> <top>)`, `(<string> <string>)`, and `(<top> <top>)`, respectively. When `foo` is applied like `(foo "abc" 3)`, the first and the third method will be selected.

Then the selected methods are sorted from the most *specific* method to the least specific method. It is calculated as follows:

- Suppose we have a method `a` that has specializers `(A1 A2 ...)`, and a method `b` that has `(B1 B2 ...)`.
- Find the minimum *n* where the classes `An` and `Bn` differ. Then the class of *n*-th argument is taken, and its class precedence list is checked. If `An` comes before `Bn` in the CPL, then method `a` is more specific than `b`. Otherwise, `b` is more specific than `a`.
- If all the specializers of `a` and `b` are the same, except that one has an improper tail (`"rest"` argument) and another doesn't, then the method that doesn't have an improper tail is more specific than the one that has.

Once methods are sorted, the body of the first method is called with the actual argument.

Within the method body, a special local variable `next-method` is bound implicitly.

`next-method` [Next method]
`next-method` *args* . . . [Next method]

This variable is bound within a method body to a special object that encapsulates the next method in the sorted method list.

Calling without arguments invokes the next method with the same arguments as this method is called with. Passing *args* . . . explicitly invokes the next method with the passed arguments.

If `next-method` is called in the least specific method, i.e. there's no `"next method"`, an error is signaled.

7.5 Metaobject protocol

In CLOS-like object systems, the object system is built on top of itself—that is, things such as the structure of the class, how a class is created, how an instance is created and initialized, and how a method is dispatched and called, are all defined in terms of the object system. For example, a class is just an instance of the class `<class>` that defines a generic structure and behavior of standard classes. If you subclass `<class>`, then you can create your own set of classes that behaves differently than the default behavior; in effect, you are creating your own object system.

Metaobject protocols are the definitions of APIs concerning about how the object systems are built—building-block classes, and the names and orders of generic functions to be called during operations of the object system. Subclassing these classes and specializing these methods are the means of customizing object system behaviors.

7.5.1 Class instantiation

Every class is an instance of a group of special classes. A class that can be a class of another class is called *metaclass*. In Gauche, only the `<class>` class or its subclasses can be a metaclass.

Expansion of `define-class`

The `define-class` macro is basically a wrapper of the code that creates an instance of `<class>` (or specified metaclass) and bind it to the given name. Suppose you have the following `define-class` form.

```
(define-class name (supers)
  slot-specs
  options ...)
```

It is expanded into a form like this (you can see the exact form by looking at the definition of `define-class` macro in `src/libobj.scm` of the source code tree.

```
(define name
  (let ((tmp1 (make metaclass
                 :name 'name :supers (list supers)
                 :slots (map process-slot-definitions
                             slot-specs)
                 :defined-modules (list (current-module))
                 options ...)))
    ... check class redefinition ...
    ... registering accessor methods ...
    tmp1))
```

The created class's class, i.e. *metaclass*, is determined by the following rules.

1. If `:metaclass` option is given to the `define-class` macro, its value is used. The value must be the `<class>` class or its descendants.
2. Otherwise, the metaclasses of the classes in the class precedence list is examined.
   - If all the metaclasses are `<class>`, then the created class's metaclass is also `<class>`.
   - If all the metaclasses are either `<class>` or another metaclass `A`, then the created class' metaclass is `A`.
   - If the set of metaclasses contains more than one metaclass (`A`, `B`, `C` ...) other than `<class>`, then the created class' metaclass is a metaclass that inherits all of those metaclasses `A`, `B`, `C` ....

The class's name, superclasses, and slot definitions are passed as the initialization arguments to the `make` generic function, with other arguments passed to `define-class`. The initialization argument `defined-modules` is passed to remember which module the class is defined, for the redefinition of this class.

The slot specifications *slot-specs* are processed by internal method *process-slot-definitions* (which can't be directly called) to be turned into slot definitions. Specifically, an `:init-form` slot option is turned into an `:init-thunk` option, and `:getter`, `:setter` and `:accessor` slot options are quoted.

After the class (an instance of *metaclass*) is created, the global binding of *name* is checked. If it is bound to a class, then the class redefinition protocol is invoked (see Section 7.2.5 [Class redefinition], page 292).

Then, the methods given to `:getter`, `:setter` and `:accessor` slot options in *slot-spec* are collected and registered to the corresponding generic functions.

Class structure

`<class>` [Class]

The base class of all metaclasses, `<class>`, has the following slots. Note that these slots are for internal management, and users can't change those information freely once the class is initialized.

It is recommended to obtain information about a class by procedures described in Section 7.2.3 [Class object], page 290, instead of directly accessing those slots.

`name` [Instance Variable of `<class>`]

The name of the class; the symbol given to `define-class` macro. `class-name` returns this value.

`cpl` [Instance Variable of `<class>`]

Class precedence list. `class-precedence-list` returns this value.

`direct-supers` [Instance Variable of `<class>`]

The list of direct superclasses. `class-direct-supers` returns this value.

`accessors` [Instance Variable of `<class>`]

An assoc list of slot accessors—it encapsulates how each slot should be accessed.

`slots` [Instance Variable of `<class>`]

A list of slot definitions. `class-slots` returns this value. See Section 7.2.4 [Slot definition object], page 291, for the details of slot definitions.

`direct-slots` [Instance Variable of `<class>`]

A list of slot definitions that is directly specified in this class definition (i.e. not inherited). `class-direct-slots` returns this value.

`num-instance-slots` [Instance Variable of `<class>`]
The number of instance allocated slots.

`direct-subclasses` [Instance Variable of `<class>`]
A list of classes that directly inherits this class. `class-direct-subclasses` returns this value.

`direct-methods` [Instance Variable of `<class>`]
A list of methods that has this class in its specializer list. `class-direct-methods` returns this value.

`initargs` [Instance Variable of `<class>`]
The initialization argument list when this class is created. The information is used to initialize redefined class (see Section 7.2.5 [Class redefinition], page 292).

`defined-modules` [Instance Variable of `<class>`]
A list of modules where this class has a global binding.

`redefined` [Instance Variable of `<class>`]
If this class has been redefined, this slot contains a reference to the new class. Otherwise, this slot has `#f`.

`category` [Instance Variable of `<class>`]
The value of this slot indicates how this class is created. Scheme defined class has a symbol `scheme`. Other values are for internal use.

The initialize method for `<class>`

`initialize` (*class* `<`*class*`>`) *:rest initargs* [Method]
The `define-class` macro expands into a call of `(make <class> ...)`, which allocates a class metaobject and calls `initialize` method. This method takes care of computing inheritance order (class precedence list) and calculate slots, and set up various internal slots. Then, at the very end of this method, it *freezes* the essential class slots; they became immutable.

Calculation of inheritance and slots are handle by generic functions. If you define a metaclass, you can define methods for them to customize how those calculations are done. Class inheritance is calculated by `compute-cpl` defined below. Slot calculation is a bit involved, and explained in the next subsection (see Section 7.5.2 [Customizing slot access], page 303).

If your metaclass needs to initialize auxiliary slots, you can define your own `initialize` method, in which you call `next-method` first to set up the core part of the `<class>` structure, then you sets up metaclass-specific part. One caveat is that, after `next-method` handles initialization of the core `<class>` part, you can no longer modify essential class slots. If you need to tweak those slots, you can override `class-post-initialize` method, which is called right before the core class slots are frozen.

`compute-cpl` *class* [Generic function]

`class-post-initialize` *class initargs* [Generic function]

7.5.2 Customizing slot access

`compute-slots` *class* [Generic Function]
`compute-get-n-set` *class slot-definition* [Generic Function]
These two generic functions are responsible to determine what slots a class has, and how each slot is accessed.

In the `initialize` method of a class, `compute-slots` is called after the class's `direct-supers`, `cpl` and `direct-slots` are set. It must decide what slots the class should have, and what slot options each slot should have, based on those three piece of information. The returned value should have the following form, and it is used as the value of the `slots` slot of the class.

```
<slots> : (<slot-definition> ...)
<slot-definition> : (<slot-name> . <slot-options>)
<slot-name> : symbol
<slot-options> : keyword-value alternating list.
```

After the `slots` slot of the class is set by the returned value from `compute-slots`, `compute-get-n-set` is called for each slot to calculate how to access and modify the slot. The class and the slot definition are the arguments. It must return either one of the followings:

an integer *n*

This slot becomes *n*-th instance slot. This is the only way to allocate a slot per instance.

The base method of *compute-get-n-set* keeps track of the current number of allocated instance slots in the class's `num-instance-slots` slot. It is not recommended for other specialized methods to use or change the value of this slot, unless you know a very good reason to override the object system behavior in deep down. Usually it is suffice to call `next-method` to let the base method reserve an instance slot for you.

See the examples below for modifying instance slot access behaviors.

a list (*get-proc set-proc bound?-proc initializable*)

The *get-proc*, *set-proc* and *bound?-proc* elements are procedures invoked when this slot of an instance is accessed (either via `slot-ref`/`slot-set!`/`slot-bound?`, or an accessor method specified by `:getter`/`:setter` slot options). The value other than *get-proc* may be `#f`, and can be omitted if all the values after it is also `#f`. That is, the simplest form of this type of return value is a list of one element, *get-proc*.

- When this slot is about to be read, *get-proc* is called with an argument, the instance. The returned value of *get-proc* is the value of the slot.

  The procedure may return `#<undef>` to indicate the slot is unbound. It triggers the `slot-unbound` generic function. (That is, this type of slot cannot have `#<undef>` as its value.)

- When this slot is about to be written, *set-proc* is called with two arguments, the instance and the new value. It is called purely for the side effect; the procedure may change the value of other slot of the instance, for example.

  If this element is `#f` or omitted, the slot becomes read-only; any attempt to write to the slot will raise an error.

- When `slot-bound?` is called to check whether the slot of an instance is bound, *bound?-proc* is called with an argument, the instance. It should return a boolean value which will be the result of `slot-bound?`.

  If this element is `#f` or omitted, `slot-bound?` will call *get-proc* and returns true if it returns `#<undef>`.

- The last element, *initializable*, is a flag that indicates whether this slot should be initialized when `:init-value` or `:init-form`.

A `<slot-accessor>` object

Access to this slot is redirected through the returned slot-accessor object. See below for more on `<slot-accessor>`.

The value returned by `compute-get-n-set` is immediately passed to `compute-slot-accessor` to create a *slot accessor* object, which encapsulates how to access and modify the slot.

After all slot definitions are processed by `compute-get-n-set` and `compute-slot-accessor`, an assoc list of slot names and `<slot-accessor>` objects are stored in the class's `accessors` slot.

`compute-slot-accessor` [Generic Function]
`compute-slot-accessor` (*class* <*class*>) *slot access-specifier* [Method]

Access-specifier is a value returned from `compute-get-n-set`. The base method creates an instance of `<slot-accessor>` that encapsulates how to access the given slot.

Created slot accessor objects are stored (as an assoc list using slot names as keys) in the class's `accessors` slot. Standard slot accessors and mutators, such as `slot-ref`, `slot-set!`, `slot-bound?`, and the slot accessor methods specified in `:getter`, `:setter` and `:accessor` slot options, all go through slot accessor object eventually. Specifically, those functions and methods first looks up the slot accessor object of the desired slot, then calls `slot-ref-using-accessor` etc.

`compute-slots` (*class* <*class*>) [Method]

The standard method walks CPL of *class* and gathers all direct slots. If slots with the same name are found, the one of a class closer to *class* in CPL takes precedence.

`compute-get-n-set` (*class* <*class*>) *slot* [Method]

The standard processes the slot definition with the following slot allocations: `:instance`, `:class`, `each-subclass` and `:virtual`.

`slot-ref-using-accessor` *obj slot-accessor* [Function]
`slot-set-using-accessor!` *obj slot-accessor value* [Function]
`slot-bound-using-accessor?` *obj slot-accessor* [Function]
`slot-initialize-using-accessor!` *obj slot-accessor initargs* [Function]

The low-level slot accessing mechanism. Every function or method that needs to read or write to a slot eventually comes down to one of these functions.

Ordinary programs need not call these functions directly. If you ever need to call them, you have to be careful not to grab the reference to *slot-accessor* too long; if *obj*'s class is changed or redefined, *slot-accessor* can no longer be used.

Here we show a couple of small examples to illustrate how slot access protocol can be customized. You can also look at `gauche.mop.*` modules (in the source tree, look under `lib/gauche/mop/`) for more examples.

The first example implements the same functionality of `:virtual` slot allocation. We add `:procedural` slot allocation, which adds `:ref`, `:set!` and `:bound?` slot options.

```
(define-class <procedural-slot-meta> (<class>) ())

(define-method compute-get-n-set ((class <procedural-slot-meta>) slot)
  (if (eqv? (slot-definition-allocation slot) :procedural)
    (let ([get-proc   (slot-definition-option slot :ref)]
          [set-proc   (slot-definition-option slot :set!)]
          [bound-proc (slot-definition-option slot :bound?)])
      (list get-proc set-proc bound-proc))
    (next-method)))
```

A specialized `compute-get-n-set` is defined on a metaclass `<procedural-slot-meta>`. It checks the slot allocation, handles it if it is `:procedural`, and delegates other slot allocation cases to `next-method`. This is a typical way to add new slot allocation by layering.

To use this :procedural slot, give <procedural-slot-meta> to a :metaclass argument of define-class:

```
(define-class <temp> ()
  ((temp-c :init-keyword :temp-c :init-value 0)
   (temp-f :allocation :procedural
           :ref   (lambda (o) (+ (*. (ref o 'temp-c) 9/5) 32))
           :set!  (lambda (o v)
                    (set! (ref o 'temp-c) (*. (- v 32) 5/9)))
           :bound? (lambda (o) (slot-bound? o 'temp-c))))
  :metaclass <procedural-slot-meta>)
```

An instance of <temp> keeps a temperature in both Celsius and Fahrenheit. Here's an example interaction.

```
gosh> (define T (make <temp>))
T
gosh> (d T)
#<<temp> 0xb6b5c0> is an instance of class <temp>
slots:
  temp-c    : 0
  temp-f    : 32.0
gosh> (set! (ref T 'temp-c) 100)
#<undef>
gosh> (d T)
#<<temp> 0xb6b5c0> is an instance of class <temp>
slots:
  temp-c    : 100
  temp-f    : 212.0
gosh> (set! (ref T 'temp-f) 450)
#<undef>
gosh> (d T)
#<<temp> 0xb6b5c0> is an instance of class <temp>
slots:
  temp-c    : 232.22222222222223
  temp-f    : 450.0
```

Our next example is a simpler version of gauche.mop.validator. We add a slot option :filter, which takes a procedure that is applied to a value to be set to the slot.

```
(define-class <filter-meta> (<class>) ())

(define-method compute-get-n-set ((class <filter-meta>) slot)
  (cond [(slot-definition-option slot :filter #f)
         => (lambda (f)
              (let1 acc (compute-slot-accessor class slot (next-method))
                (list (lambda (o) (slot-ref-using-accessor o acc))
                      (lambda (o v) (slot-set-using-accessor! o acc (f v)))
                      (lambda (o) (slot-bound-using-accessor? o acc))
                      #t)))]
        [else (next-method)]))
```

The trick here is to call next-method and compute-slot-accessor to calculate the slot accessor and wrap it. See how this metaclass works:

```
(define-class <foo> ()
  ((v :init-value 0 :filter x->number))
```

```
  :metaclass <filter-meta>)

gosh> (define foo (make <foo>))
foo
gosh> (ref foo'v)
0
gosh> (set! (ref foo'v) "123")
#<undef>
gosh> (ref foo'v)
123
```

7.5.3 Method instantiation

`make` (*class* <*method*>) *:rest initargs* [Method]

7.5.4 Customizing method application

`apply-generic` *gf args* [Generic Function]

`sort-applicable-methods` *gf methods args* [Generic Function]

`method-more-specific?` *method1 method2 classes* [Generic Function]

`apply-methods` *gf methods args* [Generic Function]

`apply-method` *gf method build-next args* [Generic Function]

8 Library modules - Overview

In the following chapters, we explain library modules bundled with Gauche's distribution. These modules should generally be loaded and imported (usually using `use` - See Section 4.13.4 [Using modules], page 73, for details), unless otherwise noted.

Some modules are described as "autoloaded". That means you don't need to `load` or `use` the module explicitly; at the first time the bindings are used in the program, the module is automatically loaded and imported. See Section 6.23.4 [Autoload], page 242, for the details of autoloading.

As the number of bundled libraries grows, it becomes harder to find the one you need. If you feel lost, check out the section Section 8.1 [Finding libraries you need], page 308, in which we categorize libraries by their purposes.

The following four chapters describe bundled modules, grouped by their names.

- Chapter 9 [Library modules - Gauche extensions], page 315, contains a description of `gauche.*` modules, which are more or less considered the core features of Gauche but separated since less frequently used. (Some modules are rather ad-hoc, but here for historical reasons).
- Chapter 10 [Library modules - R7RS standard libraries], page 499, explains how Gauche integrates R7RS into existing Gauche structures. If you want to write R7RS-compliant portable programs, you definitely want to check the first two sections of this chapter. What follows is the description of R7RS modules. Since Gauche supports most of R7RS core procedures in either built-in or existing modules, most R7RS modules are for the compatibility.
- Chapter 11 [Library modules - SRFIs], page 590, describes the modules which provide SRFI functionalities. They have the names beginning with `srfi-`. Note that some of SRFI features are built in Gauche core and not listed here. See Section 2.1 [Standard conformance], page 5, for the entire list of supported SRFIs.
- Chapter 12 [Library modules - Utilities], page 638, describes other modules —including database interface, filesystem utilities, network protocol utilities, and more.

There are a few procedures that help your program to check the existence of certain modules or libraries at run-time. See Section 6.23.5 [Operations on libraries], page 243, for the details.

8.1 Finding libraries you need

Each module is named more or less after what it implements rather than what it is implemented *for*. If the module solves one problem, both are the same. However, sometimes there are multiple ways to solve a problem, or one implementation of an algorithm can solve multiple different problems; thus it is difficult to name the modules in problem-oriented (or purpose-oriented) way.

Because of this, it may not be straightforward for a newcomer to Gauche to find an appropriate Gauche module to solve her problem, since there may be multiple algorithms to do the job, and each algorithm can be implemented in different modules.

The modules are also designed in layers; some low-level modules provide direct interface to the system calls, while some higher-level ones provide more abstract, easy-to-use interface, possibly built on top of more than one low-level modules. Which one should you use? Generally you want to use the highest level, for the very purpose of libraries are to provide easy, abstract interface. However there are times that you have to break the abstraction and to go down to tweak the machinery in the basement; then you need to use low-level modules directly.

The purpose of this section is to group the libraries by their purposes. Each category lists relevant modules with brief descriptions.

8.1.1 Library directory - data containers

Generic container operations

Some data containers have similar properties; for example, lists, vectors and hash tables can be seen as a collection of data. So it is handy to have generic operators, such as applying a procedure to all the elements.

Gauche provides such mechanism to a certain degree, mainly using its object system.

- *Collection* - Generic functions applicable for unordered set of values. See Section 9.5 [Collection framework], page 344.
- *Sequence* - Generic functions applicable for ordered set of values. See Section 9.29 [Sequence framework], page 441.
- *Dictionary* - Generic functions to handle dictionary, that is, a mapping from keys to values. See Section 9.9 [Dictionary framework], page 366.
- *Relation* - Generic functions to handle relations (in a sense of Codd's definition). See Section 12.70 [Relation framework], page 807.
- *Comprehension* - This is a collection of macros very handy to construct and traverse collections/sequences in concise code. See Section 11.12 [Eager comprehensions], page 610.

Container implementations

- *List* - the universal data structure. You want to check Section 6.6 [Pairs and lists], page 125, and Section 10.3.1 [R7RS lists], page 512,
- *Vector* - a one-dimensional array of arbitrary Scheme values. See Section 6.14 [Vectors], page 174, and Section 11.30 [Vector library], page 632. If you need a wide range of index, but the actual data is sparse, you might want to look at Section 12.15.1 [Sparse vectors], page 667.
- *Uniform vector* - a special kind of vectors that can hold limited types of values (e.g. integers representable in 8bits). It tends to be used in performance sensitive applications, such as graphics. See Section 9.36 [Uniform vectors], page 476.
- *Array* - multi-dimensional arrays that can hold arbitrary Scheme values. See Section 9.1 [Arrays], page 315.
- *Uniform array* - multi-dimensional arrays that can hold limited types of values. This is also supported by Section 9.1 [Arrays], page 315.
- *String* - a sequence of characters. See Section 6.12 [Strings], page 153, and Section 11.5 [String library], page 591. Gauche handles multibyte strings— see Section 2.2 [Multibyte strings], page 11, for the details.
- *Character set* - a set of characters. See Section 6.11 [Character set], page 147, and Section 11.6 [Character-set library], page 600.
- *Hash table* - hash tables. See Section 6.15 [Hashtables], page 177. For very large hash tables (millions of entries), Section 12.15.3 [Sparse tables], page 671, may provide better memory footprint.
- *Balanced tree* - If you need to order keys in a dictionary, you can use treemaps. See Section 6.16 [Treemaps], page 182.
- *Immutable map* - Sometimes immutable dictionary is handy. Internally it implements a functional balanced tree. See Section 12.11 [Immutable map], page 654.
- *Queue* - Both fast and thread-safe queues are provided in Section 12.12 [Queue], page 656. Thread-safe queues can also be used as synchronized messaging channel.
- *Heap* - See Section 12.9 [Heap], page 652.
- *Ring buffer* - Space-efficient ring buffer. See Section 12.14 [Ring buffer], page 665.

- *Cache* - Various cache algorithm implementations. See Section 12.8 [Cache], page 649.
- *Record* - a simple data structure. Although Gauche's object system can be used to define arbitrary data structures, you might want to look at Section 9.26 [Record types], page 433, and Section 12.69 [SLIB-compatible record type], page 806, for they are more portable and potentially more efficient.
- *Stream* - you can implement cool lazy algorithms with it. See Section 12.71 [Stream library], page 809.
- *Trie* - Another tree structure for efficient common-prefix search. See Section 12.16 [Trie], page 672.
- *Database interface* - dbm interface can be used as a persistent hash table; see Section 12.18 [Generic DBM interface], page 682. For generic RDBMS interface, see Section 12.17 [Database independent access layer], page 676.

8.1.2 Library directory - string and character

Basic string operations are covered in Section 6.12 [Strings], page 153, and Section 11.5 [String library], page 591. A string is also a sequence of characters, so you can apply methods in Section 9.5 [Collection framework], page 344, and Section 9.29 [Sequence framework], page 441.

Character and character set operations are covered in Section 6.10 [Characters], page 143, Section 6.11 [Character set], page 147, and Section 11.6 [Character-set library], page 600.

If you scan or build strings sequentially, do not use index access. String ports (see Section 6.22.5 [String ports], page 224) provides more efficient, and elegant way.

You can use regular expressions to search and extract character sequences from strings; see Section 6.13 [Regular expressions], page 162.

If you need to deal with low-level (i.e. byte-level) representation of strings, Section 9.36 [Uniform vectors], page 476, has some tools to convert strings and byte vectors back and forth.

Are you dealing with a structure higher than a mere sequence of characters? Then take a look at `text.*` modules. Section 12.56 [Parsing input stream], page 786, has some basic scanners. Section 12.60 [Transliterate characters], page 791, implements a feature similar to Unix's `tr(1)`. You can take `diff` of two texts; see Section 12.52 [Calculate difference of text streams], page 779. And if you want to construct large text from string fragments, do not use `string-append`—see Section 12.61 [Lazy text construction], page 793.

Last but not least, Gauche has support of various character encoding schemes. See Section 9.4 [Character code conversion], page 339, for the basic utilities. Most higher-level functions such as `open-input-file` can take `:encoding` keyword argument to perform character conversion implicitly. Also see Section 2.3 [Multibyte scripts], page 11, if you write Scheme program in non-ASCII characters. If you want to process Gauche source code which may contain "encoding" magic comment, see Section 6.22.6 [Coding-aware ports], page 226. Gauche also has GNU gettext compatible module (Section 12.54 [Localized messages], page 783) if you need localization.

8.1.3 Library directory - data exchange

Most useful programs need to communicate with outside world (other programs or humans). That involves reading the external data into your program understanding whatever format the data is in, and also writing the data in the format the others can understand.

Lots of network-related external formats are defined in RFC, and there are corresponding `rfc.*` module that handle some of them. See Section 12.29 [RFC822 message parsing], page 712, for example, to handle the pervasive RFC2822 message format. Or, JSON can be handled by Section 12.37 [JSON parsing and construction], page 728.

When you exchange table-formatted data, one of the easiest way may be the plain text, one row per line, and columns are separated by some specific characters (e.g. comma). See Section 12.51 [CSV tables], page 776, for basic parser/writer for them.

Oh, and nowadays every business user wants XML, right? You know they are just S-expressions with extra redundancy and pointy parentheses. So why don't you read XML as if they're S-exprs, process them with familiar cars and cdrs and maps, then write them out with extra redundancy and pointy parens? Module `sxml.ssax` (Section 12.46 [Functional XML parser], page 747) implements SAX XML parser, with which you can parse XML and process them on the fly, or convert it to SXML, S-expression XML. You can query SXML using SXPath, an XPath counterparts of S-expression (Section 12.47 [SXML query language], page 757). You can output all kinds of XML and HTML using the SXML serializer (Section 12.49 [Serializing XML and HTML from SXML], page 771).

(But you know most web services nowadays also talks JSON, and that's much lighter and handier than XML. See Section 12.37 [JSON parsing and construction], page 728).

It is planned that various file format handling routines would be available as `file.*` modules, though we have none ready yet. If you plan to write one, please go ahead and let us know!

8.1.4 Library directory - files

Files and directories. Roughly speaking, there are two places you want to look at.

Section 6.25.4 [Filesystems], page 250, in the core, has routines close to the underlying OS provides. If you have experience with Unix system programming you'll find familiar function names there. The `fcntl` functionality is splitted to `gauche.fcntl` (Section 9.10 [Low-level file operations], page 370), FYI.

Also you definitely want to look at `file.util` (Section 12.24 [Filesystem utilities], page 692), which implements higher-level routines on top of system-level ones.

8.1.5 Library directory - processes and threads

Process-related routines also come in two levels.

The `gauche.process` module provides high-level routines (Section 9.25 [High-level process interface], page 421); you can pipe the data into and out of child processes easily, for example.

Gauche core provides the primitive `fork` and `exec` interface as well as the convenient `system` call (see Section 6.25.10 [Process management], page 271). Use them when you want a precise control over what you're doing.

Gauche has preemptive threads on most Unix platforms including OSX. Check out Section 9.33 [Threads], page 457, for the basic thread support, including primitive mutexes. The `data.queue` module (see Section 12.12 [Queue], page 656) provides thread-safe queue that can also be handy for synchronization. Thread pool is available in `control.thread-pool` (see Section 12.6 [Thread pools], page 646).

8.1.6 Library directory - networking

We have multi-layer abstraction here. At the bottom, we have APIs corresponding to socket-level system calls. In the middle, a convenience library that automates host name lookups, connection and shutdown, etc. On top of them we have several modules that handles specific protocols (e.g. http).

The `gauche.net` module (Section 9.20 [Networking], page 398) provides the bottom and middle layer. For the top layer, look for `rfc.*` modules, e.g. `rfc.http` (Section 12.34 [HTTP], page 721). More protocol support is coming (there are `rfc.ftp` and `rfc.imap4` written by users, which are waiting for being integrated into Gauche—maybe in next release).

There's a plan of even higher level of libraries, under the name `net.*`, which will abstract more than one network protocols. The planned ones include sending emails, or universal resource access by uri. Code contributions are welcome.

8.1.7 Library directory - input and output

8.1.8 Library directory - time

8.1.9 Library directory - bits and bytes

Binary I/O

As the bottom level, Gauche includes primitive byte I/O (`read-byte`, `write-byte`) as well as block I/O (`read-uvector`, `read-uvector!`, `write-uvector`) in its core. (See Section 6.22.7.1 [Reading data], page 227, Section 6.22.8 [Output], page 231, and Section 9.36.4 [Uvector block I/O], page 488).

As the middle level, the module `binary.io` (Section 12.1 [Binary I/O], page 638) has routines to retrieve specific datatype with optional endian specification.

And as the top level, the module `binary.pack` (Section 12.2 [Packing binary data], page 641) allows packing and unpacking structured binary data, a la Perl's `pack/unpack`.

Bit manipulation

Gauche core provides basic bitshift and mask operations (see Section 6.3.6 [Basic bitwise operations], page 121). SRFI-151 has comprehensive bitwise operations (see Section 11.34 [Bitwise operations], page 633).

8.2 Naming convention of libraries

The following table summarizes naming categories of the modules, including external ones and planned ones.

`binary.*` Utilities to treat binary data.

`compat.*` Provides compatibility layers.

`data.*` Implementations of various data structures.

`dbi.*, dbd.*`
Database independent interface layer and drivers.

`dbm.*` DBM interface

`gauche.*` Stuffs more or less considered as Gauche core features.

`gl.*` OpenGL binding and related libraries (external package).

`gtk.*` GTk+ binding and related libraries (external package).

`file.*` Manipulating files and directories.

`lang.*` Language-related libraries, artificial and/or natural (planned).

`math.*` Mathematics.

`os.*` Features for specific OSes.

`rfc.*` Implementations of net protocols defined in RFC's.

`srfi-*` SRFI implementations.

`sxml.*` SXML libraries.

`text.*` Libraries dealing with text data.

`util.*` Generic implementations of various algorithms.

`www.*` Implementations of various protocols and formats mainly used in WWW.

8.3 Obsolete and superseded modules

During the course of development of Gauche, some modules have been renamed, merged, or dissolved into the core. Also, some SRFI libraries become standard and given a new name, or superseded with a newer SRFI library.

We list such modules here for the reference. New code shouldn't use these modules, although they are kept in the distribution so that legacy code can keep running.

Obsolete modules

`text.unicode` [Module]

Renamed to `gauche.unicode`. See Section 9.35 [Unicode utilities], page 471.

`util.list` [Module]

Dissolved into the core. No longer needed.

`util.queue` [Module]

Renamed to `data.queue`. See Section 12.12 [Queue], page 656.

`util.rbtree` [Module]

Incorporated into the core as built-in object `<tree-map>`. See Section 6.16 [Treemaps], page 182.

The following procedures are aliases of the ones with replacing `rbtree` for `tree-map`, e.g. `rbtree-get` is the same as `tree-map-get`.

```
make-rbtree       rbtree?           rbtree-get        rbtree-put!
rbtree-delete!    rbtree-exists?    rbtree-empty?     rbtree-update!
rbtree-push!      rbtree-pop!       rbtree-num-entries rbtree->alist
alist->rbtree     rbtree-keys       rbtree-values     rbtree-copy
rbtree-fold       rbtree-fold-right
```

The following procedures are similar to `tree-map-min`, `tree-map-max`, `tree-map-pop-min!` and `tree-map-pop-max!`, respectively, except that the `rbtree-*` version takes an optional default argument and returns it when the tree is empty, and raise an error if no default argument is provided and tree is empty. (The tree-map version just returns `#f` for the empty tree.)

```
rbtree-min            rbtree-max
rbtree-extract-min!   rbtree-extract-max!
```

The following procedure doesn't have corresponding API in tree-map. It checks internal consistency of the given tree-map.

```
rbtree-check
```

`util.sparse` [Module]

Renamed to `data.sparse`. See Section 12.15 [Sparse data containers], page 666.

`util.trie` [Module]

Renamed to `data.trie`. See Section 12.16 [Trie], page 672.

Superseded modules

`srfi-1` [Module]

SRFI-1 (List library) has become a part of R7RS large, as `scheme.list`. See Section 10.3.1 [R7RS lists], page 512.

`srfi-14` [Module]

SRFI-14 (Character-set library) has become a part of R7RS large, as `scheme.charset`. See Section 10.3.6 [R7RS character sets], page 533.

`srfi-43` [Module]

Vector library (Legacy) - this module is effectively superseded by R7RS and `srfi-133`. See Section 6.14 [Vectors], page 174, and see Section 11.30 [Vector library], page 632.

`srfi-60` [Module]

Integers as bits - this module is superseded by `srfi-151`. See Section 11.34 [Bitwise operations], page 633.

`srfi-69` [Module]

Basic hash tables - this module is superseded by R7RS `scheme.hash-table`. See Section 10.3.7 [R7RS hash tables], page 536.

`srfi-111` [Module]

SRFI-111 (Boxes) has become a part of R7RS `scheme.box` module. See Section 10.3.12 [R7RS boxes], page 547.

`srfi-113` [Module]

SRFI-113 (Sets and bags) has become a part of R7RS `scheme.set`. See Section 10.3.5 [R7RS sets], page 525.

`srfi-114` [Module]

Comparators - R7RS favored srfi-128 over this srfi to make `scheme.comparator` (Section 10.3.15 [R7RS comparators], page 551), so adoption of this srfi may not be as wide.

Note that, in Gauche, a native comparator object can be used for srfi-114 procedures, and this module provides some useful additional utilities. It's ok to use this module if portability isn't a big issue.

`srfi-117` [Module]

SRFI-117 has become R7RS's `scheme.list-queue`. See Section 10.3.13 [R7RS list queues], page 547.

`srfi-127` [Module]

SRFI-127 has become R7RS's `scheme.lseq`. See Section 10.3.10 [R7RS lazy sequences], page 544.

`srfi-132` [Module]

SRFI-132 has become R7RS's `scheme.sort`. See Section 10.3.4 [R7RS sort], page 522.

`srfi-133` [Module]

SRFI-133 has become R7RS's `scheme.vector`. See Section 10.3.2 [R7RS vectors], page 517.

9 Library modules - Gauche extensions

9.1 gauche.array - Arrays

gauche.array [Module]

This module provides multi-dimensional array data type and operations. The primitive API follows SRFI-25. Besides a generic srfi-25 array that can store any Scheme objects, this module also provides array classes that stores numeric objects efficiently, backed up by homogeneous numeric vectors (see Section 9.36 [Uniform vectors], page 476). An external representation of arrays, using SRFI-10 mechanism, is also provided.

Each element of an *N*-dimensional array can be accessed by *N* integer indices, `[ i_0 i_1 ... i_N-1 ]`. An array has associated *shape* that knows lower-bound *s_k* and upper-bound *e_k* of index of each dimension, where *s_k* <= *e_k*, and the index *i_k* must satisfy *s_k* <= *i_k* < *e_k*. (Note: it is allowed to have *s_k* == *e_k*, but such array can't store any data. It is also allowed to have zero-dimensional array, that can store a single data.). The shape itself is a [*D* x 2] array, where *D* is the dimension of the array which the shape represents.

You can pass index(es) to array access primitives in a few ways; each index can be passed as individual argument, or can be 'packed' in a vector or one-dimensional array. In the latter case, such a vector or an array is called an `"index object"`. Using a vector is efficient in Gauche when you iterate over the elements by changing the vector elements, for it won't involve memory allocation.

Arrays can be compared by the `equal?` procedure. `Equal?` returns `#t` if two arrays have the same shape and their corresponding elements are the same in the sense of `equal?`.

Internally, an array consists of a backing storage and a mapping procedure. A backing storage is an object of aggregate type that can be accessed by an integer index. A mapping procedure takes multi-dimensional indices (or index object) and returns a scalar index into the backing storage.

`<array-base>` [Class]

{`gauche.array`} An abstract base class of array types, that implements generic operations on the array. To create an array instance, you should use one of the following concrete array classes.

`<array>` [Class]
`<u8array>` [Class]
`<s8array>` [Class]
`<u16array>` [Class]
`<s16array>` [Class]
`<u32array>` [Class]
`<s32array>` [Class]
`<u64array>` [Class]
`<s64array>` [Class]
`<f16array>` [Class]
`<f32array>` [Class]
`<f64array>` [Class]

{`gauche.array`} Concrete array classes. The `<array>` class implements srfi-25 compatible array, i.e. an array that can store any Scheme objects. The `<u8array>` class through `<f64array>` classes uses a `<u8vector>` through `<f64vector>` as a backing storage, and can only store a limited range of integers or inexact real numbers, but they are space efficient.

`#,(<array> shape obj ...)` [Reader Syntax]

An array is written out in this format. (Substitute `<array>` for `<u8array>` if the array is `<u8array>`, etc.) *shape* is a list of even number of integers, and each $2n$-th integer and $2n+1$-th integer specifies the inclusive lower-bound and exclusive upper-bound of n-th dimension, respectively. The following *obj* ... are the values in the array listed in row-major order.

When read back, this syntax is read as an array with the same shape and content, so it is `equal?` to the original array.

```
; an array such that:
;   8 3 4
;   1 5 9
;   6 7 2
#,(<array> (0 3 0 3) 8 3 4 1 5 9 6 7 2)

; a 4x4 identity matrix
#,(<array> (0 4 0 4) 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1)
```

`array?` *obj* [Function]

[SRFI-25] {`gauche.array`} Returns `#t` if *obj* is an array, `#f` otherwise. It is equivalent to `(is-a? obj <array-base>)`.

`make-array` *shape :optional init* [Function]

[SRFI-25] {`gauche.array`} Creates an array of shape *shape*. *Shape* must be a [D x 2] array, and for each k ($0 <= k < D$), the [k 0] element must be less than or equal to the [k 1] element. If *init* is given, all the elements are initialized by it. Otherwise, the initial value of the elements are undefined.

```
(make-array (shape 0 2 0 2 0 2) 5)
 ⇒ #,(<array> (0 2 0 2 0 2) 5 5 5 5 5 5 5 5)
```

`make-u8array` *shape :optional init* [Function]
`make-s8array` *shape :optional init* [Function]

...

`make-f32array` *shape :optional init* [Function]
`make-f64array` *shape :optional init* [Function]

{`gauche.array`} Like `make-array`, but creates and returns an uniform numeric array.

`array-copy` *array* [Function]

{`gauche.array`} Returns a copy of *array*, with the same class, shape and content.

`shape` *bound* ... [Function]

[SRFI-25] {`gauche.array`} Takes even number of exact integer arguments, and returns a two-dimensional array that is suitable for representing the shape of an array.

```
(shape 0 2 1 3 3 5)
 ⇒ #,(<array> (0 3 0 2) 0 2 1 3 3 5)

(shape)
 ⇒ #,(<array> (0 0 0 2))
```

`array` *shape init* ... [Function]

[SRFI-25] {`gauche.array`} Creates an array of shape *shape*, initializing its elements by *init* ....

```
(array (shape 0 2 1 3) 'a 'b 'c 'd)
 ⇒ #,(<array> (0 2 1 3) a b c d)
```

`u8array` *shape init ...* [Function]
`s8array` *shape init ...* [Function]
...

`f32array` *shape init ...* [Function]
`f64array` *shape init ...* [Function]

{`gauche.array`} Like `array`, but creates and returns an uniform numeric array initialized by *init* ....

```
(u8array (shape 0 2 0 2) 1 2 3 4)
 ⇒ #,(<u8array> (0 2 0 2) 1 2 3 4)
```

`array-rank` *array* [Function]

[SRFI-25] {`gauche.array`} Returns the number of dimensions of an array *array*.

```
(array-rank (make-array (shape 0 2 0 2 0 2))) ⇒ 3
(array-rank (make-array (shape))) ⇒ 0
```

`array-shape` *array* [Function]

{`gauche.array`} Returns a shape array of *array*.

`array-start` *array dim* [Function]
`array-end` *array dim* [Function]
`array-length` *array dim* [Function]

[SRFI-25+] {`gauche.array`} `Array-start` returns the inclusive lower bound of index of *dim*-th dimension of an array *array*. `Array-end` returns the exclusive upper bound. And `array-length` returns the difference between two. `Array-start` and `array-end` are defined in SRFI-25.

```
(define a (make-array (shape 1 5 0 2)))

(array-start a 0)  ⇒ 1
(array-end a 0)    ⇒ 5
(array-length a 0) ⇒ 4
(array-start a 1)  ⇒ 0
(array-end a 1)    ⇒ 2
(array-length a 1) ⇒ 2
```

`array-size` *array* [Function]

{`gauche.array`} Returns the total number of elements in the array *array*.

```
(array-size (make-array (shape 5 9 1 3))) ⇒ 8
(array-size (make-array (shape))) ⇒ 1
(array-size (make-array (shape 0 0 0 2))) ⇒ 0
```

`array-ref` *array k ...* [Function]
`array-ref` *array index* [Function]

[SRFI-25] {`gauche.array`} Gets the element of array `array`. In the first form, the element is specified by indices *k* .... In the second form, the element is specified by an index object *index*, which must be a vector or an one-dimensional array.

`array-set!` *array k ... value* [Function]
`array-set!` *array index value* [Function]

[SRFI-25] {`gauche.array`} Sets the element of array *array* to *value*. In the first form, the element is specified by indices *k* .... In the second form, the element is specified by an index object *index*, which must be a vector or an one-dimensional array.

`share-array` *array shape proc* [Function]

[SRFI-25] {`gauche.array`} Creates and returns a new array of shape *shape*, that shares the backing storage with the given array *array*. The procedure *proc* maps the indices of the new array to the indices to the original array, i.e. *proc* must be a *n*-ary procedure that returns *m* values, where *n* is the dimension of the new array and *m* is the one of the original array. Furthermore, *proc* must be an affine function; each mapping has to be a linear combination of input arguments plus optional constant. (`Share-array` optimizes the mapping function based on the affinity assumption, so *proc* won't be called every time the new array is accessed).

`array-for-each-index` *array proc :optional index* [Function]

{`gauche.array`} Calls *proc* with every index of *array*. If no *index* argument is provided, *proc* is called as `(proc i j k ...)`, in which (*i*,*j*,*k*,. . .) walks over the index. It begins from the least index value of each dimension, and latter dimension is incremented faster.

```
gosh> (define a (array (shape 0 2 0 2) 1 2 3 4))
a
gosh> a
#,(<array> (0 2 0 2) 1 2 3 4)
gosh> (array-for-each-index a (^(i j) (print i","j)))
0,0
0,1
1,0
1,1
```

This form of passing indexes is simple but not very efficient, though. For better performance, you can pass an index object to an optional argument *index*, which is modified for each index and passed to *proc*. The index object must be mutable, and either a vector, an one-dimensional array, an s8vector, an s16vector or an s32vector. The length of the index object must match the rank of the array. Using index object is efficient since the loop won't allocate. Don't forget that the index object is destructively modified within the loop.

```
gosh> (array-for-each-index a (cut format #t "~s\n" <>) (vector 0 0))
#(0 0)
#(0 1)
#(1 0)
#(1 1)

gosh> (array-for-each-index a (cut format #t "~s\n" <>) (s8vector 0 0))
#s8(0 0)
#s8(0 1)
#s8(1 0)
#s8(1 1)
```

The procedure returns an unspecified value.

`shape-for-each` *shape proc :optional index* [Function]

{`gauche.array`} Calls *proc* with all possible indexes represented by the shape *shape*. The optional *index* argument works the same way as `array-for-each-index`. Returns an unspecified value.

```
gosh> (shape-for-each (shape 0 2 0 2) (^(i j) (print i","j)))
0,0
0,1
1,0
1,1
```

`tabulate-array` *shape proc :optional index* [Function]

{`gauche.array`} Calls *proc* over each index represented by the shape *shape*, and creates an array from the result of *proc*. The optional index object can be used in the same way as `array-for-each-index`. The following example creates an identity matrix of the given shape:

```
(tabulate-array (shape 0 3 0 3) (^(i j) (if (= i j) 1 0)))
  ⇒ #,(<array> (0 3 0 3) 1 0 0 0 1 0 0 0 1)
```

`array-retabulate!` *array proc :optional index* [Function]
`array-retabulate!` *array shape proc :optional index* [Function]

{`gauche.array`} Calls *proc* over each index of the given *array*, and modifies the array's element by the returned value of *proc*. The optional index object can be used in the same way as `array-for-each-index`. The second form takes a shape; it must match the *array*'s shape. It is redundant, but may allow some optimization in future in case *shape* is a literal. Returns an unspecified value.

`array-map` *proc array0 array1 . . .* [Function]
`array-map` *shape proc array0 array1 . . .* [Function]

{`gauche.array`} The arguments *array0*, *array1*, . . . must be arrays with the same shape. For each set of corresponding elements of the input arrays, *proc* is called, and a new array of the same shape is created by the returned values. The second form takes a shape argument, which must match the shape of input array(s). It is redundant, but may allow some optimization in future in case *shape* is a literal.

```
(array-map - (array (shape 0 2 0 2) 1 2 3 4))
  ⇒ #,(<array> (0 2 0 2) -1 -2 -3 -4)
```

`array-map!` *array proc array0 array1 . . .* [Function]
`array-map!` *array shape proc array0 array1 . . .* [Function]

{`gauche.array`} Like `array-map`, but the results of *proc* are stored by the given *array*, whose shape must match the shape of input array(s). Returns unspecified value.

`array->vector` *array* [Function]
`array->list` *array* [Function]

{`gauche.array`} Returns a fresh vector or a fresh list of all elements in *array*.

```
(array->vector
 (tabulate-array (shape 1 3 1 4)
                 (^(i j) (+ (* 10 i) j))))
 ⇒ #(11 12 13 21 22 23)
```

`array-concatenate` *a b :optional dimension* [Function]

{`gauche.array`} Concatenates arrays at the specified dimension. The sizes of the specified dimension of two arrays must match, although the shapes can be different. Arrays can be of any ranks, but two ranks must match.

```
;;  [a b]              [a b]
;;  [c d] (+)       => [c d]
;;            [e f]    [e f]
(array-concatenate
 (array (shape 0 2 0 2) 'a 'b 'c 'd)
 (array (shape 0 1 0 2) 'e 'f))
 ⇒ #,(<array> (0 3 0 2) a b c d e f)

;;  [a b]     [e]    [a b e]
;;  [c d] (+) [f] => [c d f]
```

```
(array-concatenate
 (array (shape 0 2 0 2) 'a 'b 'c 'd)
 (array (shape 0 2 0 1) 'e 'f)
 1)
 ⇒ #,(<array> (0 2 0 3) a b e c d f)

;; The index range can differ, as far as the sizes match
(array-concatenate
 (array (shape 0 2 0 2) 'a 'b 'c 'd)
 (array (shape 1 3 0 1) 'e 'f) 1)
 ⇒ #,(<array> (0 2 0 3) a b e c d f)
```

`array-transpose` *array :optional dim1 dim2* [Function]

{`gauche.array`} The given array must have a rank greater than or equal to 2. Transpose the array's *dim1*-th dimension and *dim2*-th dimension. The default is 0 and 1.

`array-rotate-90` *array :optional dim1 dim2* [Function]

{`gauche.array`} The given array must have a rank greater than or equal to 2. We regard the array as a matrix with *dim1*-th dimension as rows and *dim2*-th dimension as columns, and returns a fresh array whose content is filled by *rotating array* 90 degree clockwise. The defaults of *dim1* and *dim2* are 0 and 1, respectively.

```
;; [1 2 3]      [4 1]
;; [4 5 6]  =>  [5 2]
;;              [6 3]
(array-rotate-90 (array (shape 0 2 0 3) 1 2 3 4 5 6))
 ⇒ #,(<array> (0 3 0 2) 4 1 5 2 6 3)
```

If *array* has a rank greater than 2, the array is treated as a matrix of subarrays.

`array-flip` *array :optional dimension* [Function]
`array-flip!` *array :optional dimension* [Function]

{`gauche.array`} Flips the content of the array across the *dimension*-th dimension. (default is 0). `array-flip!` modifies the content of *array* and return it. `array-flip` doesn't modify *array* but creates a fresh array with the flipped content and returns it.

```
;; [1 2 3]  =>  [4 5 6]
;; [4 5 6]      [1 2 3]
(array-flip (array (shape 0 2 0 3) 1 2 3 4 5 6))
 ⇒ #,(<array> (0 2 0 3) 4 5 6 1 2 3)

;; [1 2 3]  =>  [3 2 1]
;; [4 5 6]      [6 5 4]
(array-flip (array (shape 0 2 0 3) 1 2 3 4 5 6) 1)
 ⇒ #,(<array> (0 2 0 3) 3 2 1 6 5 4)
```

`identity-array` *dimension :optional class* [Function]

{`gauche.array`} Returns a fresh identity array of rank 2, with the given dimension. You can pass one of array classes to *class* to make the result the instance of the class; the default class is `<array>`.

```
(identity-array 3)
 ⇒ #,(<array> (0 3 0 3) 1 0 0 0 1 0 0 0 1)

(identity-array 3 <f32array>)
 ⇒ #,(<f32array> (0 3 0 3) 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0)
```

`array-inverse` *array* [Function]

{`gauche.array`} Regards the *array* as a matrix, and returns its inverse matrix; *array* must be 2-dimensional, and must have square shape. If *array* doesn't satisfy these conditions, an error is thrown.

If *array* isn't a regular matrix, `#f` is returned.

`determinant` *array* [Function]
`determinant!` *array* [Function]

{`gauche.array`} Regards the *array* as a matrix, and calculates its determinant; *array* must be 2-dimensional, and must have square shape. If *array* doesn't satisfy these conditions, an error is thrown.

`determinant!` destructively modifies the given array during calculation. It is faster than `determinant`, which copies *array* before calculation to preserve it.

`array-mul` *a b* [Function]

{`gauche.array`} Arrays *a* and *b* must be rank 2. Regarding them as matrices, multiply them together. The number of rows of *a* and the number of columns of *b* must match.

```
;;             [6 5]
;; [1 2 3] x [4 3] => [20 14]
;; [4 5 6]   [2 1]    [56 41]

(array-mul (array (shape 0 2 0 3) 1 2 3 4 5 6)
           (array (shape 0 3 0 2) 6 5 4 3 2 1))
 ⇒ #,(<array> (0 2 0 2) 20 14 56 41)
```

`array-expt` *array pow* [Function]

{`gauche.array`} Raises *array* to the power of *pow*; *array* must be a square matrix, and *pow* must be a nonnegative exact integer.

`array-div-left` *a b* [Function]
`array-div-right` *a b* [Function]

{`gauche.array`} Inverse of `array-mul`; `array-div-left` returns a matrix `M` such that `(array-mul B M)` equals to `A`, and `array-div-right` returns a matrix `M` such that `(array-mul M B)` equals to `A`. *A* and *B* must be a 2-dimensional square matrix. If `B` isn't regular, an error is thrown.

`array-add-elements` *array array-or-scalar ...* [Function]
`array-add-elements!` *array array-or-scalar ...* [Function]
`array-sub-elements` *array array-or-scalar ...* [Function]
`array-sub-elements!` *array array-or-scalar ...* [Function]
`array-mul-elements` *array array-or-scalar ...* [Function]
`array-mul-elements!` *array array-or-scalar ...* [Function]
`array-div-elements` *array array-or-scalar ...* [Function]
`array-div-elements!` *array array-or-scalar ...* [Function]

{`gauche.array`} Element-wise arithmetics. The second argument and after must be an array of the same shape of the first argument, or a number; if it is a number, it is interpreted as an array of the same shape of the first argument, and each element of which is the given number.

Returns an array of the same shape of the first argument, where each element is the result of addition, subtraction, multiplication or division of the corresponding elements of the arguments.

The linear-update version (procedures whose name ends with `!`) may reuse the storage of the first array to calculate the result. The first array must be mutable. The caller must still use the returned value instead of counting on the side effects.

```
(array-add-elements (array (shape 0 2 0 2) 1 2 3 4)
                    (array (shape 0 2 0 2) 5 6 7 8)
                    10)
 ⇒ #,(<array> (0 2 0 2) 16 18 20 22)

(array-div-elements (array (shape 0 2 0 2) 1 3 5 7)
                    100
                    (array (shape 0 2 0 2) 2 4 6 8))
 ⇒ #,(<array> (0 2 0 2) 1/200 3/400 1/120 7/800)
```

If only one argument is passed, these procedures returns the argument itself.

You can mix different types of arrays as long as their shapes are the same. The result is the same type as the first argument.

```
(array-mul-elements (make-u8array (shape 0 2 0 2) 3)
                    (array (shape 0 2 0 2) 1 3 5 7))
 ⇒ #,(<u8array> (0 2 0 2) 3 9 15 21)
```

`array-negate-elements` *array* [Function]
`array-negate-elements!` *array* [Function]

Returns an array with the same type of the shape of *array*, but each element is a negation of the corresponding elements in the original array.

```
(array-negate-elements (array (shape 0 2 0 2) 1 2 3 4))
  ⇒ #,(<array> (0 2 0 2) -1 -2 -3 -4)
```

`array-reciprocate-elements` *array* [Function]
`array-reciprocate-elements!` *array* [Function]

Returns an array with the same type of the shape of *array*, but each element is a reciprocal of the corresponding elements in the original array.

```
(array-reciprocate-elements (array (shape 0 2 0 2) 1 2 3 4))
  ⇒ #,(<array> (0 2 0 2) 1 1/2 1/3 1/4)
```

9.2 `gauche.base` - Importing gauche built-ins

`gauche.base` [Module]

This module exports Gauche built-in procedures and syntaxes, so that they can be imported to other modules that don't inherit `gauche` module.

All the bindings available in the `gauche` module are exported, except `import`, which is renamed to `gauche:import` to avoid conflict with R7RS `import`.

The module extends `gauche.keyword`, so also exports all the keywords—the bindings from `gauche.keyword`— so that the code imports `gauche.base` can access to self-bound keywords without inheriting the keyword module.

Typical Gauche code doesn't need this module, for built-ins are available by default through inheritance. A newly created module *inherits* the `gauche` module by default. (See Section 4.13.5 [Module inheritance], page 75, for the details.)

Sometimes you need a module that doesn't inherit the `gauche` module, yet you want to use Gauche built-in features. Particularly, R7RS libraries and programs require any bindings to be explicitly imported, so R7RS's `import` and `define-library` sets up the module not to inherit the `gauche` module. In R7RS code, you need `(import (gauche base))` to use Gauche's built-in features.

Another use case is to eliminate some built-in bindings, yet keep the rest of bindings accessible, in your module. For example, the following setup creates `almost-gauche` module that has almost all default bindings except `string-scan` and `string-split`:

```
(define-module almost-gauche
  (use scheme.r5rs)
  (use gauche.base :except (string-scan string-split)
                   :rename ((gauche:import import)))
  (extend)
  )
(select-module almost-gauche)

;; your code here
```

Note the empty `extend`; it empties the module's inheritance. (The `:rename` option of `gauche.base` is just to get the original name of `import` back in `almost-gauche` module; if you don't use `import` directly, you won't need it.)

9.3 gauche.cgen - Generating C code

Significant part of Gauche is written in Gauche or S-expression based DSL. During the building process, they are converted into C sources and then compiled by C compiler. The `gauche.cgen` module and its submodules expose the functionality Gauche build process is using to the general use.

Required features for a C code generator differ greatly among applications, and too much scaffolding could be a constraint for the module users. So, instead of providing a single solid framework, we provide a set of loosely coupled modules so that you can combine necessary features freely. In fact, some of Gauche build process only use `gauche.cgen.unit` and `gauche.cgen.literal` (see `src/builtin-syms.scm`, for example).

`gauche.cgen` [Module]

This is a convenience module that extends `gauche.cgen.unit`, `gauche.cgen.literal`, `gauche.cgen.type` and `gauche.cgen.cise` together.

Usually you can just use `gauche.cgen` and don't need to think about individual submodules. The following subsections are organized by submodules only for the convenience of explanation.

9.3.1 Generating C source files

One of the tricky issues about generating C source is that you have to put several fragments of code in different parts of the source file, even you want to say just one thing—that is, sometimes you have to put declaration before the actual definition, plus some setup code that needs to be run at initialization time.

Creating a frame

`<cgen-unit>` [Class]

{`gauche.cgen`} A *cgen-unit* is a unit of C source generation. It corresponds to one .c file, and optionally one .h file. During the processing, a "current unit" is kept in a parameter `cgen-current-unit`, and most cgen APIs implicitly work to it.

The following slot are for public use. They are used to tailor the output. Usually you set those slots at initialization time. The effect is undefined if you change them in the middle of the code generation process.

`name` [Instance Variable of `<cgen-unit>`]

A string to name this unit. This is used for the default name of the generated files (*`name`*`.c` and *`name`*`.h`) and the suffix of the default name of initialization function. Other cgen modules may use this to generate names. Avoid using characters that are not valid for C identifiers.

You can override those default names by setting the other slots.

`c-file` [Instance Variable of `<cgen-unit>`]
`h-file` [Instance Variable of `<cgen-unit>`]

The name of the C source file and header file, in strings. If they are `#f` (by default), the value of name slot is used as the file name, with extension `.c` or `.h` is attached, respectively.

To get the file names to be generated, use `cgen-unit-c-file` and `cgen-unit-h-file` generic functions, instead of reading these slots.

`preamble` [Instance Variable of `<cgen-unit>`]

A list of strings to be inserted at the top of the generated sources. The default value is `("/* Generated by gauche.cgen */")`. Each string appears in its own line.

`init-prologue` [Instance Variable of `<cgen-unit>`]
`init-epilogue` [Instance Variable of `<cgen-init>`]

A string to start or to end the initialization function, respectively. The default value of `init-prologue` is `"void Scm_Init_NAME(void) {"` where `NAME` is the value of the `name` slot. The default value of `init-epilogue` is just `"}"`. Each string appears in its own line.

To get the default initialization function name, use `cgen-unit-init-name` generic function.

To customize initialization function name, arguments and/or return type, set `init-prologue`.

The content of initialization function is filled by the code fragments registered by `cgen-init`.

`cgen-current-unit` [Parameter]

A parameter to keep the current cgen-unit.

A typical flow of generating C code is as follows:

1. Create a `<cgen-unit>` and make it the current unit.
2. Call code insertion APIs with code fragments. Fragments are accumulated in the current unit.
3. Call *emit* method on the unit, which generates a C file and optionally a header file.

`cgen-emit-c` *cgen-unit* [Generic Function]
`cgen-emit-h` *cgen-unit* [Generic Function]

{`gauche.cgen`} Write the accumulated code fragments in *cgen-unit* to a C source file and C header file. The name of the files are determined by calling `cgen-unit-c-file` and `cgen-unit-h-file`, respectively. If the files already exist, its content is overwritten; you can't gradually write to the files. So, usually these procedures are called at the last step of the code generation.

We'll explain the details of how each file is organized under "Filling the content" section below.

`cgen-unit-c-file` *cgen-unit* [Generic Function]
`cgen-unit-h-file` *cgen-unit* [Generic Function]

{`gauche.cgen`} Returns a string that names C source and header file for *cgen-unit*, respectively. The default method first looks at `c-file` or `h-file` slot of the *cgen-unit*, and if it is `#f`, use the value of `name` slot and appends an extension `.c` or `.h`.

`cgen-unit-init-name` *cgen-unit* [Generic Function]

{`gauche.cgen`} Returns a string that names the initialization function generated to C. It is used to create the default `init-prologue` value.

Filling the content

There are four parts to which you can add C code fragment. Within each part, code fragments are rendered in the same order as added.

extern This part is put into the header file, if exists.

decl Placed at the beginning of the C source, after the standard prologue.

body Placed in the C source, following the 'decl' part.

init Placed inside the initialization function, which appears at the end of the C source.

The following procedures are the simple way to put a source code fragments in an appropriate part:

`cgen-extern` *code* . . . [Function]
`cgen-decl` *code* . . . [Function]
`cgen-body` *code* . . . [Function]
`cgen-init` *code* . . . [Function]

{`gauche.cgen`} Put code fragments *code* . . . to the appropriate parts. Each fragment must be a string.

This is a minimal example to show the typical usage. After running this code you'll get `my-cfile.c` and `my-cfile.h` in the current directory.

```
(use gauche.parameter)
(use gauche.cgen)

(define *unit* (make <cgen-unit> :name "my-cfile"))

(parameterize ([cgen-current-unit *unit*])
  (cgen-decl "#include <stdio.h>")
  (cgen-init "printf(stderr, \"initialization function\\n\");")
  (cgen-body "void foo(int n) { printf(stderr, \"got %d\\n\", n); }")
  (cgen-extern "void foo(int n);")
  )

(cgen-emit-c *unit*)
(cgen-emit-h *unit*)
```

These are handy escaping procedures; they are useful even if you don't use other parts of the `cgen` modules.

`cgen-safe-name` *string* [Function]
`cgen-safe-name-friendly` *string* [Function]
`cgen-safe-string` *string* [Function]
`cgen-safe-comment` *string* [Function]

{`gauche.cgen`} Escapes characters invalid in C identifiers, C string literals or C comments.

With `cgen-safe-name`, characters other than ASCII alphabets and digits are converted to a form `_XX`, where `XX` is hexadecimal notation of the character code. (Note that the character `_` is also converted.) So the returned string can be used safely as a C identifier. The mapping is injective, that is, if the source strings differ, the result string always differ.

On the other hand, `cgen-safe-name-friendly` converts the input string into more readable C identifier. `->` becomes `_TO` (e.g. `char->integer` becomes `char_TOinteger`), other `-` and `_` become `_`, `?` becomes `P` (e.g. `char?` becomes `charP`), `!` becomes `X` (e.g. `set!` becomes `setX`), `<` and `>` become `_LT` and `_GT` respectively. Other special characters except `_` are converted to `_XX` as in `cgen-safe-name`. The mapping is not injective; e.g. both `read-line` and `read_line` map to `read_line`. Use this only when you think some human needs to read the generated C code (which is not recommended, by the way.)

If you want to write out a Scheme string as a C string literal, you can use `cgen-safe-string`. It escapes control characters and non-ascii characters. If the Scheme string contains a character beyond ASCII, it is encoded in Gauche's native encoding. (NB: It also escapes `?`, to avoid accidenal formation of C trigraphs).

Much simpler is `cgen-safe-comment`, which just converts `/*` and `*/` into `/ *` and `* /` (a space between those two characters), so that it won't terminate the comment inadvertently. (Technically, escaping only `*/` suffice, but some simple-minded C parser might be confused by `/*` in the comments). The conversion isn't injective as well.

```
(cgen-safe-name "char-alphabetic?")
  ⇒ "char_2dalphabetic_3f"
(cgen-safe-name-friendly "char-alphabetic?")
  ⇒ "char_alphabeticP"
(cgen-safe-string "char-alphabetic?")
  ⇒ "\"char-alphabetic\\077\""

(cgen-safe-comment "*/*"
  ⇒ "* / *"
```

If you want to conditionalize a fragment by C preprocessor `#ifdef`s, use the following macro:

cgen-with-cpp-condition *cpp-expr body . . .* [Macro]

{gauche.cgen} Code fragments submitted in *body* . . . are protected by `#if` *cpp-expr* and `#endif`.

If *cpp-expr* is a string, it is emitted literally:

```
(cgen-with-cpp-condition "defined(FOO)"
  (cgen-init "foo();"))

;; will generate:
#if defined(FOO)
foo();
#endif /* defined(FOO) */
```

You can also construct *cpp-expr* by S-expr.

```
<cpp-expr> : <string>
           | (defined <cpp-expr>)
           | (not <cpp-expr>)
           | (<n-ary-op> <cpp-expr> <cpp-expr> ...)
           | (<binary-op> <cpp-expr> <cpp-expr>)

<n-ary-op> : and | or | + | * | - | /

<binary-op> : > | >= | == | < | <= | !=
            | logand | logior | lognot | >> | <<
```

Example:

```
(cgen-with-cpp-condition '(and (defined FOO)
```

```
                                (defined BAR))
    (cgen-init "foo();"))

  ;; will generate:
  #if ((defined FOO)&&(defined BAR))
  foo();
  #endif /* ((defined FOO)&&(defined BAR)) */
```

You can nest `cgen-with-cpp-condition`.

Submitting code fragments for more than one parts

When you try to abstract code generation process, calling individual procedures for each parts (e.g. `cgen-body` or `cgen-init`) becomes tedious, since such higher-level constructs are likely to require generating code fragments to various parts. Instead, you can create a customized class that handles submission of fragments to appropriate parts.

`<cgen-node>` [Class]

{`gauche.cgen`} A base class to represent a set of code fragments.

The state of C preprocessor condition (set by `with-cgen-cpp-condition`) is captured when an instance of the subclass of this class is created, so generating appropriate `#if`s and `#endif`s are automatically handled.

You subclass `<cgen-node>`, then define method(s) to one or more of the following generic functions:

`cgen-emit-xtrn` *cgen-node* [Generic Function]
`cgen-emit-decl` *cgen-node* [Generic Function]
`cgen-emit-body` *cgen-node* [Generic Function]
`cgen-emit-init` *cgen-node* [Generic Function]

{`gauche.cgen`} These generic functions are called during writing out the C source within `cgen-emit-c` and `cgen-emit-h`. Inside these methods, anything written out to the current output port goes into the output file.

While generating `.h` file by `cgen-emit-h`, `cgen-emit-xtrn` method for all submitted nodes are called in order of submission.

While generating `.c` file by `cgen-emit-c`, `cgen-emit-decl` method for all submitted nodes are called first, then `cgen-emit-body` method, then `cgen-emit-init` method.

If you don't specialize any one of these method, it doesn't generate code in that part.

Once you define your subclass and create an instance, you can submit it to the current cgen unit by this procedure:

`cgen-add!` *cgen-node* [Function]

{`gauche.cgen`} Submit *cgen-node* to the current cgen unit. If the current unit is not set, *cgen-node* is simply ignored.

In fact, the procedures `cgen-extern`, `cgen-decl`, `cgen-body` and `cgen-init` are just a convenience wrapper to create an internal subclass specialized to generate code fragment only to the designated part.

9.3.2 Generating Scheme literals

Sometimes you want to refer to a Scheme constant value in C code. It is trivial if the value is a simple thing like Scheme boolean (`SCM_TRUE`, `SCM_FALSE`), characters (`SCM_MAKE_CHAR(code)`), small integers (`SCM_MAKE_INT(value)`), etc. You can directly write it in C code. However, once you step outside of these simple values, it gets tedious quickly, involving static data declarations and/or runtime initialization code.

For example, to get a Scheme value of a list of symbols (a b c), you have to (1) create ScmStrings for the names of the symbols, (2) pass them to Scm_Intern to get Scheme symbols, then (3) call Scm_Conses (or a convenience macro SCM_LIST3) to build a list.

With gauche.cgen, those code can be generated automatically.

NOTE: If you use cgen-literal, make sure you call (cgen-decl "#include <gauche.h>") to include gauche.h before the first call of cgen-literal, which may insert declarations that needs gauche.h.

cgen-literal *obj* [Function]

{gauche.cgen} Returns an <cgen-literal> object for a Scheme object *obj*, and submit necessary declarations and initialization code to the current cgen unit.

For the above example, you can just call (cgen-literal '(a b c)) and the C code to set up the Scheme literal of the list of three symbols will be generated.

The result of cgen-literal is an instance of <cgen-literal>; the detail of the class isn't for public use, but you can use it to refer the created literal in C code.

cgen-cexpr *cgen-literal* [Generic Function]

{gauche.cgen} Returns a C code expression fragment of type ScmObj, which represents the Scheme literal value.

The following example creates a C function printabc that prints the literal value (a b c), created by cgen-literal.

```
(define *unit* (make <cgen-unit> :name "foo"))
(parameterize ((cgen-current-unit *unit*))
  (let1 lit (cgen-literal '(a b c))
    (cgen-body
     (format "void printabc() { Scm_Printf(SCM_CUROUT, \"%S\", ~a); }"
             (cgen-c-name lit)))))
(cgen-emit-c *unit*)
```

If you examine the generated file foo.c, you'll get a general idea of how it is handled.

One advantage of cgen-literal is that it tries to share the same literal whenever possible. If you call (cgen-literal '(a b c)) twice in the same cgen unit, you'll get one instance of cgen-literal. If you call (cgen-literal '(b c)) then, it will share the tail of the original list (a b c). So you can just use cgen-literal whenever you need to have Scheme literal values, without worrying about generating excessive amount of duplicated code.

Certain Scheme objects cannot be generated as a literal; for example, an opened port can't, since it carries lots of runtime information.

(There's a machinery to allow programmers to extend the cgen-literal behavior for new types. The API isn't fixed yet, though.)

9.3.3 Conversions between Scheme and C

In the C world, any Scheme object is uniformly of type ScmObj. But it is often the case that you need to narrow down to the specific type and convert it to a C value. Gauche maintains a database of how to typecheck and map Scheme value to C value and vice versa.

Note that the mapping isn't one-to-one: Scheme <integer> can be mapped to C's short, long, unsigned int, or even just ScmObj if the C routine wants to cover bignums. So each mapping has its own name. For historical reasons, each mapping is called *stub type*. The names of stub types look like Scheme type but its semantics differ from Scheme type. Remember: Each stub type represents a specific mapping between a Scheme type and a C type.

Each stub type has a *C-predicate*, a *boxer* and an *unboxer*, each of them is a Scheme string for the name of a C function or C macro. A C-predicate takes `ScmObj` object and returns C boolean value that if the given object has a valid type and range for the stub type. A boxer takes C object and converts it to a Scheme object; it usually involves wrapping or *boxing* the C value in a tagged pointer or object, hence the name. An unboxer does the opposite: takes a Scheme object and convert it to a C value. The Scheme object must be checked by the C-predicate before being passed to the unboxer.

The following table shows the predefined stub types. Note that the most of aggregate types has one to one mappings. The difficult ones are numeric types and strings. Scheme numbers can represent much wider range of numbers than C, so you have to narrow down according to the capability of C routine. Scheme strings have byte size and character length, and the body may not be NULL-terminated; so the `<string>` stub type maps Scheme string to `ScmString*`. For the convenience, you can use `<const-cstring>`, which creates NUL-terminated C string; beware that it may incur some copying cost.

```
Stub type    Scheme       C            Notes
-----------------------------------------------------------------
<fixnum>     <integer>    int          Integers within fixnum range
<integer>    <integer>    ScmObj       Any exact integers
<real>       <real>       double       Value converted to double
<number>     <number>     ScmObj       Any numbers

<int>        <integer>    int          Integers representable in C
<int8>       <integer>    int
<int16>      <integer>    int
<int32>      <integer>    int
<short>      <integer>    short
<long>       <integer>    long
<uint>       <integer>    uint         Integers representable in C
<uint8>      <integer>    uint
<uint16>     <integer>    uint
<uint32>     <integer>    uint
<ushort>     <integer>    ushort
<ulong>      <integer>    ulong
<float>      <real>       float        Unboxed value casted to float
<double>     <real>       double       Alias of <real>

<boolean>    <boolean>    int          Boolean value
<char>       <char>       ScmChar      Note: not a C char

<void>       -            void         (Used only as a return type.
                                         Scheme function returns #<undef>)

<string>     <string>     ScmString*   Note: not a C string

<const-cstring> <string>  const char*  For arguments, string is unboxed
                                       by Scm_GetStringConst.
                                       For return values, C string is boxed
                                       by SCM_MAKE_STR_COPYING.
```

```
<const-cstring-safe> <string> const char*  Like <const-cstring>,
                                            but when converting from Scheme,
                                            reject a string with NUL chars in it.

<pair>        <pair>        ScmPair*
<list>        <list>        ScmObj
<string>      <string>      ScmString*
<symbol>      <symbol>      ScmSymbol*
<keyword>     <keyword>     ScmKeyword*
<vector>      <vector>      ScmVector*
<uvector>     <uvector>     ScmUVector*
<s8vector>    <s8vector>    ScmS8Vector*
<u8vector>    <u8vector>    ScmU8Vector*
<s16vector>   <s16vector>   ScmS16Vector*
<u16vector>   <u16vector>   ScmU16Vector*
<s32vector>   <s32vector>   ScmS32Vector*
<u32vector>   <u32vector>   ScmU32Vector*
<s64vector>   <s64vector>   ScmS64Vector*
<u64vector>   <u64vector>   ScmU64Vector*
<f16vector>   <f16vector>   ScmF16Vector*
<f32vector>   <f32vector>   ScmF32Vector*
<f64vector>   <f64vector>   ScmF64Vector*
<c32vector>   <c32vector>   ScmC32Vector*
<c64vector>   <c64vector>   ScmC64Vector*
<c128vector>  <c128vector>  ScmC128Vector*

<hash-table>  <hash-table>  ScmHashTable*
<tree-map>    <tree-map>    ScmTreeMap*

<char-set>    <char-set>    ScmCharSet*
<regexp>      <regexp>      ScmRegexp*
<regmatch>    <regmatch>    ScmRegMatch*
<port>        <port>        ScmPort*
<input-port>   <input-port> ScmPort*
<output-port> <output-port> ScmPort*
<procedure>   <procedure>   ScmProcedure*
<closure>     <closure>     ScmClosure*
<promise>     <promise>     ScmPromise*

<class>       <class>       ScmClass*
<method>      <method>      ScmMethod*
<module>      <module>      ScmModule*
<thread>      <thread>      ScmVM*
<mutex>       <mutex>       ScmMutex*
<condition-variable> <condition-variable> ScmConditionVariable*
```

A stub type can have a *maybe* variation, denoted by `?` suffix; e.g. `<string>?`. It is a union type of the base type and boolean false (for `<string>?`, it can be either `<string>` or `#f`.) In the C world, boolean false is mapped to `NULL` pointer. It is convenient to pass a C value that allowed to be NULL back and forth—if you pass `#f` from the Scheme world it comes out `NULL` to the C world, and vice versa. The maybe variation is only meaningful when the C type is a pointer type.

`<cgen-type>` [Class]

{`gauche.cgen`} An instance of this class represents a stub type. It can be looked up by name such as `<const-cstring>` by `cgen-type-from-name`.

`cgen-type-from-name` *name* [Function]

{`gauche.cgen`} Returns an instance of `<cgen-type>` that has *name*. If the name is unknown, `#f` is returned.

`cgen-box-expr` *cgen-type c-expr* [Function]
`cgen-unbox-expr` *cgen-type c-expr* [Function]
`cgen-pred-expr` *cgen-type c-expr* [Function]

{`gauche.cgen`} *c-expr* is a string denotes a C expression. Returns a string of C expression that boxes, unboxes, or typechecks the *c-expr* according to the *cgen-type*.

```
;; suppose foo() returns char*
(cgen-box-expr
 (cgen-type-from-name '<const-cstring>)
 "foo()")
 ⇒ "SCM_MAKE_STR_COPYING(foo())"
```

9.3.4 CiSE - C in S expression

Some low-level routines in Gauche are implemented in C, but they're written in S-expression. We call it "C in S expression", or *CiSE*.

The advantage of using S-expression is its readability, obviously. Another advantage is that it allows us to write macros as S-expr to S-expr translation, just like the legacy Scheme macros. That's a powerful feature—effectively you can extend C language to suit your needs.

The `gauche.cgen.cise` module provides a set of tools to convert CiSE code into C code to be passed to the C compiler. It also has some support to overcome C quirks, such as preparing forward declarations.

Currently, we don't do rigorous check for CiSE; you can pass a CiSE expression to the translator that yields invalid C code, which will cause the C compiler to emit errors. The translator inserts line directives by default so the C compiler error message points to the location of original (CiSE) source instead of generated code; however, sometimes you need to look at the generated code to figure out what went wrong. We hope this will be improved in future.

In Gauche source code, CiSE is extensively used in precompiled Scheme files and recognized by the precompiler (`precomp`). However, `gauche.cgen.cise` is an independent module only relies on `gauche.cgen` basic features, so you can plug it to your own C code generating programs.

9.3.4.1 CiSE overview

Before diving into the details, it's easier to grasp some basic concepts.

A *CiSE fragment* is an S-expression that follows CiSE syntax (see Section 9.3.4.2 [CiSE syntax], page 332). A CiSE fragment can be translated to a C code fragment by `cise-render`. Note that some translation may not be local, e.g. it may want to emit forward declarations before other C code fragments. So, the full translation requires buffering—you process all the CiSE fragments and save output, emit forward declarations, then emit the saved C code fragments. We have a wrapper procedure, `cise-translate`, to take care of it, but for your purpose you may want to roll your own wrapper.

A *CiSE macro* is a Scheme code that translates a CiSE fragment to another CiSE fragment. There are number of predefined CiSE macros. You can add your own CiSE macros by utilities such as `define-cise-stmt` and `define-cise-expr`.

A *CiSE ambient* is a bundle of information that affects fragment translation. It contains CiSE macro definitions, and also it keeps track of forward declarations.

9.3.4.2 CiSE syntax

In this section, we lists basic CiSE syntax. They are just data from the viewpoint of Gauche—so you can build and manipulate them like any S-expression (quasiquote comes pretty handy).

CiSE types

C types can be written either as a symbol (e.g. `int`) or a list (e.g. `(const char *)`. When used in definition, it is preceded by `::`. The following example shows types are used in local variable definitions:

```
(let* ([a :: int 0]
       [b :: (const char *) "abc"])
  ...)
```

For the convenience and readability, you can write the variable name, separating double-colon and type name concatenated. You can also concatenate point suffixes (`char*` instead of `char *` in the following example):

```
(let* ([a::int 0]
       [b::(const char*) "abc"])
  ...)
```

CiSE translater first breaks up these concatenated forms, then deal with types.

At this moment, CiSE does *not* check if type is valid C type. It just pass along whatever given.

There are a few special type notations for more complex types. These can appear in middle of the type; for example, you can write `(const .struct x (a::int b::double) *)` to produce `const struct x {int a; double b;} *`.

`.array` *elt-type* (*dim* . . .) [CiSE Type]

`.struct` [*tag*] [(*field-spec* . . .)] [CiSE Type]
`.union` [*tag*] [(*field-spec* . . .)] [CiSE Type]

`.function` (*arg-spec* . . .) *ret-type* [CiSE Type]

CiSE statements

`begin` *stmt* . . . [CiSE Statement]

Code grouping with `{` and `}`

`let*` ((*name* [`::` *type*] [*init-expr*]) . . .) *stmt* . . . [CiSE Statement]

Declare and optionally assign initial values to local variables.

type should be a CiSE type. If *type* is omitted, the default type is `ScmObj`. Note that array initialization is not supported yet.

`if` *test-expr then-stmt* [*else-stmt*] [CiSE Statement]
`when` *test-expr stmt* . . . [CiSE Statement]
`unless` *test-expr stmt* . . . [CiSE Statement]
`cond` (*cond1 stmt1* . . .) . . . [`(else` *else-stmt* . . .)] [CiSE Statement]

Conditional statements.

`case` *expr* ((*val1* . . .) *stmt1* . . .) . . . [`(else` *else-stmt* . . .)] [CiSE Statement]
`case/fallthrough` *expr* ((*val1* . . .) *stmt1* . . .) . . . [`(else` *else-stmt* . . .)] [CiSE Statement]

Switch-case statement. `case` does not fall through between 'case' blocks while `case/fallthrough` does.

`for` (*start-expr test-expr update-expr*) *stmt* ... [CiSE Statement]
`for` () *stmt* ... [CiSE Statement]
`loop` *stmt* ... [CiSE Statement]
`while` *test-expr body* ... [CiSE Statement]

Loop statements.

`for-each` (`lambda` (*var*) *stmt* ...) *expr* [CiSE Statement]
`dolist` [*var expr*] *stmt* ... [CiSE Statement]

expr must yield a list. Traverse the list, binding each element to *var* and executing *stmt* .... The lambda form is a fake; you don't really create a closure.

`pair-for-each` (`lambda` (*var*) *stmt* ...) *expr* [CiSE Statement]

Like `for-each`, but *var* is bound to each 'spine' cell instead of each element of the list.

`dopairs` [*var expr*] *stmt* ... [CiSE Statement]

`dotimes` (*var expr*) *stmt* ... [CiSE Statement]

expr must yield an integer, *n*. Repeat *stmt* ... by binding *var* from 0 to (*n*-1).

`return` [*expr*] [CiSE Statement]
`break` [CiSE Statement]
`continue` [CiSE Statement]

Return, break and continue statements.

`label` *name* [CiSE Statement]
`goto` *name* [CiSE Statement]

Label and goto statements. We always add a null statement after the label so that we can place `(label name)` at the end of a compound statement.

`.if expr stmt [stmt]` [CiSE Statement]
`.when expr stmt ...` [CiSE Statement]
`.unless expr stmt ...` [CiSE Statement]
`.cond clause ...` [CiSE Statement]
`.define name[(arg ...)] [expr]` [CiSE Statement]
`.undef name` [CiSE Statement]
`.include path` [CiSE Statement]

Preprocessor directives.

expr could be a string, a symbol, a number or one of the following forms:

- `(defined c)`
- `(not c)`
- `(and c)`
- `(or c)`
- `(op c ...)` where *op* is either `+` or `*`.
- `(op c c ...)` where *op* is either `-` or `/`.
- `(op c c)` where *op* is either `>`, `>=`, `==`, `<`, `<=`, `!=`, `logand`, `logior`, `lognot`, `<<` or `>>`.

Note that defining a macro function without value

```
#define foo(abc)
```

is not supported because it's ambiguous with

```
#define foo abc()
```

when written in CiSE syntax. `(.define foo (abc))` always generates the latter.

`.include` could take a symbol. This is used for including system header files, e.g. `(.include <stdint.h>)`.

`define-cfn` *name* (*arg* [`::` *type*] ...) [*ret-type* [*qualifier* ...]] *stmt* ... [CiSE Statement]

Defines a C function.

If *type* or *ret-type* is omitted, the default type is `ScmObj`.

Supported qualifiers are `:static` and `:inline`, corresponding to C's `static` and `inline` keywords. If `:static` is specified, forward declaration is automatically generated.

`define-cvar` *name* [`::` *type*] [*qualifier* ...] [<*init-expr*>] [CiSE Statement]

Defines a global C variable. Supported qualifier is `:static`. Note that array initialization is not supported yet.

`define-ctype` *name* [`::` *type*] [CiSE Statement]

Defines a new type using `typedef`

`declare-cfn` *name* (*arg* [`::` *type*] ...) [*ret-type*] [CiSE Statement]
`declare-cvar` *name* [`::` *type*] [CiSE Statement]

Declares an external C function or variable.

`.static-decls` [CiSE Statement]

Produce declarations of static functions before function bodies.

`.raw-c-code` *body* ... [CiSE Statement]

CiSE expressions

`+` *expr* ... [CiSE Expression]
`-` *expr* ... [CiSE Expression]
`*` *expr* ... [CiSE Expression]
`/` *expr* ... [CiSE Expression]
`%` *expr1 expr2* [CiSE Expression]

Arithmetic operations.

`and` *expr* ... [CiSE Expression]
`or` *expr* ... [CiSE Expression]
`not` *expr* [CiSE Expression]

Boolean operations.

`logand` *expr1 expr2* ... [CiSE Expression]
`logior` *expr1 expr2* ... [CiSE Expression]
`logxor` *expr1 expr2* ... [CiSE Expression]
`lognot` *expr* [CiSE Expression]
`<<` *expr1 expr2* [CiSE Expression]
`>>` *expr1 expr2* [CiSE Expression]

Bitwise operations.

`*` *expr* [CiSE Expression]
`->` *expr1 expr2* ... [CiSE Expression]
`ref` *expr1 expr2* ... [CiSE Expression]
`aref` *expr1 expr2* ... [CiSE Expression]
`&` *expr* [CiSE Expression]

Dereference, reference and address operations. `ref` is C's `.`. `aref` is array reference.

`pre++` *expr* [CiSE Expression]
`post++` *expr* [CiSE Expression]
`pre--` *expr* [CiSE Expression]
`post--` *expr* [CiSE Expression]

Pre/Post increment or decrement.

`<` *expr1 expr2* [CiSE Expression]
`<=` *expr1 expr2* [CiSE Expression]
`>` *expr1 expr2* [CiSE Expression]
`>=` *expr1 expr2* [CiSE Expression]
`==` *expr1 expr2* [CiSE Expression]
`!=` *expr1 expr2* [CiSE Expression]

Comparison.

`set!` *lvalue1 expr1 lvalue2 expr2 . . .* [CiSE Expression]
`=` *lvalue1 expr1 lvalue2 expr2 . . .* [CiSE Expression]
`+=` *lvalue expr* [CiSE Expression]
`-=` *lvalue expr* [CiSE Expression]
`*=` *lvalue expr* [CiSE Expression]
`/=` *lvalue expr* [CiSE Expression]
`%=` *lvalue expr* [CiSE Expression]
`<<=` *lvalue expr* [CiSE Expression]
`>>=` *lvalue expr* [CiSE Expression]
`logand=` *lvalue expr* [CiSE Expression]
`logior=` *lvalue expr* [CiSE Expression]
`logxor=` *lvalue expr* [CiSE Expression]

Assignment expressions.

`cast` *type expr* [CiSE Expression]

Type casting.

`?:` *test-expr then-expr else-expr* [CiSE Expression]

Conditional expression.

`.type` *type* [CiSE Expression]

Useful to place a type name, e.g. an argument of sizeof operator.

9.3.4.3 CiSE procedures

`cise-ambient` [Parameter]

{`gauche.cgen`}

`cise-default-ambient` [Function]

{`gauche.cgen`}

`cise-ambient-copy` *ambient* [Function]

{`gauche.cgen`}

`cise-ambient-decl-strings` *ambient* [Function]

{`gauche.cgen`}

`cise-emit-source-line` [Parameter]

{`gauche.cgen`}

`cise-render` *cise-fragment :optional port context* [Function]

{`gauche.cgen`}

`cise-render-to-string` *cise-fragment :optional context* [Function]

{`gauche.cgen`}

`cise-render-rec` *cise-fragment stmt/expr env* [Function]

{`gauche.cgen`}

`cise-translate` *inp outp :key environment* [Function]
{`gauche.cgen`}

`cise-register-macro!` *name expander :optional ambient* [Function]
{`gauche.cgen`}

`cise-lookup-macro` *name :optional ambient* [Function]
{`gauche.cgen`}

`define-cise-stmt` *name* [*env*] *clause* . . . [*:where definition* . . .] [Macro]
`define-cise-expr` *name* [*env*] *clause* . . . [*:where definition* . . .] [Macro]
`define-cise-toplevel` *name* [*env*] *clause* . . . [*:where definition* . . .] [Macro]
{`gauche.cgen`}

`define-cise-macro` (*name form env*) *body* . . . [Macro]
`define-cise-macro` *name name2* [Macro]
{`gauche.cgen`}

9.3.5 Stub generation

`define-type` *NAME C-TYPE* [*DESC C-PREDICATE UNBOXER BOXER*] [Stub Form]

Register a new type to be recognized. This is rather a declaration than definition; no C code will be generated directly by this form.

`define-cproc` *name* (*args* . . .) [*ret-type*] [*flag* . . .] [*qualifier* . . .] *body* . . . [Stub Form]

Create Scheme procedure.

args specifies arguments:

- `arg ... [:rest var]` : Each *arg* is variable name or *var::type*, specifies required argument. If `:rest` is given, list of excessive arguments are passed to *var*.
- `arg ... :optional spec ... [:rest rest-var]` : Optional arguments. *spec* is *var* or (*var default*). If no *default* is given, *var* receives `SCM_UNBOUND`—if *var* isn't a type of `ScmObj` it will raise an error.
- `ARG ... :key spec ... [:allow-other-keys [:rest rest-var]]` : Keyword arguments. *spec* is *var* or (*var default*). If no *default* is given, *var* receives `SCM_UNBOUND`—if *var* isn't a type of `ScmObj` it will raise an error.
- `arg ... :optarray (var cnt max) [:rest rest-var]` : A special syntax to receive optional arguments as a C array. *var* is a C variable of type `ScmObj*`. *cnt* is a C variable of type `int`, which receives the number of optional argument in the ScmObj array. *max* specifies the maximum number of optional arguments that can be passed in the array form. If more than *max* args are given, a list of excessive arguments are passed to the *rest-var* if it is specified

ret-type specifies the return type of function. It could be either `:: typespec` or `::typespec` where *typespec* is a valid stub type, or `(type ...)` when multiple values are returned. When omitted, the procedure is assumed to return `<top>`.

flag is a keyword to modify some aspects of the procedure. Supported flags are as follows:

- `:fast-flonum` - indicates that the procedure accepts flonum arguments and it won't retain the reference to them. The VM can pass flonums on VM registers to the procedure with this flag. (This improves floating-point number handling, but it's behavior is highly VM-specific; ordinary stub writers shouldn't need to care about this flag at all.)

- `:constant` - indicates that this procedure returns a constant value if all args are compile-time constants. The compiler may replace the call to this proc with the value, if it determines all arguments are known at the compile time. The resulting value should be serializable to the precompiled file.

  NB: Since this procedure may be called at compile time, a subr that may return a different value for batch/cross compilation shouldn't have this flag.

qualifier is a list to adds auxiliary information to the procedure. Currently the following qualifiers are officially supported.

- `(setter setter-name)` : specify setter. *setter-name* should be a cproc name defined in the same stub file
- `(setter (args ...) body ...)` : specify setter anonymously.
- `(catch (decl c-stmt ...) ...)` : when writing a stub for C++ function that may throw an exception, use this spec to ensure the exception will be caught and converted to Gauche error condition.
- `(inliner insn-name)` : only used in Gauche core procedures that can be inlined into an VM instruction.

stmt is a cise expression. Inside the expression, a cise macro `(result expr ...)` can be used to assign the value(s) to return from the cproc. As a special case, if *stmt* is a single symbol, it names a C function to be called with the same argument (mod unboxing) as the cproc.

`define-cgeneric` *name c-name property-clause ...* [Stub Form]

Defines generic function. *c-name* specifies a C variable name that keeps the generic function structure. One or more of the following clauses can appear in *property-clause ...*:

- `(extern)` : makes *c-name* visible from other file (i.e. do not define the structure as `static`).
- `(fallback "fallback")` : specifies the fallback function.
- `(setter . setter-spec)` : specifies the setter.

`define-cmethod` *name (arg ...) body ...* [Stub Form]

`define-cclass` *scheme-name [qualifier ...] c-type-name c-class-name cpa (slot-spec ...) property ...* [Stub Form]

Generates C stub for static class definition, slot accessors and initialization. Corresponding C struct has to be defined elsewhere.

The following qualifiers are supported:

- `:base` generates a base class definition (inheritable from Scheme code).
- `:built-in` generates a built-in class definition (not inheritable from Scheme code). This is the default if neither `:base` nor `:built-in` are specified.
- `:private` - the class declaration and standard macro definitions are also generated (which needs to be in the separate header file if you want the C-level structure to be used from other C code. If the extension is small enough to be contained in one C file, this option is convenient.)

cpa lists ancestor classes in precedence order. They need to be C identifiers of Scheme class `Scm_*Class`, for the time being. `Scm_TopClass` is added at the end automatically.

slot-spec is defined as `(slot-name [qualifier ...])` or `slot-name`. The following qualifiers are supported:

- `:type cgen-type`
- `:c-name c-name` specifies the C field name if the autogenerated name from *slot-name* is not accurate.

- `:c-spec` *c-spec*
- `:getter` *proc-spec* specifies how to create the slot getter. *proc-spec* could be
  - `#f` to omit the getter
  - `#t` to generate a default one with type conversion according to *type*
  - A string is interpreted as the C code to implement the getter
  - `(c` *c-name*`)` specifies the C function name that implements the getter, which is implemented elsewhere.
- `:setter` *proc-spec* specifies how to create the slot setter. The syntax is the same as `:getter`.

The following *property* are supported:

- `(allocator` *proc-spec*`)`
- `(printer` *proc-spec*`)`
- `(comparer` *proc-spec*`)`
- `(direct-supers` *string* `...)`

`define-cptr` *scheme-name* [*qualifier* ...] *c-type c-name c-pred c-boxer c-unboxer* [(*flags* *flag* ...)] [(*print* *print-proc*)] [(*cleanup* *cleanup-proc*)] [Stub Form]

Defines a new foreign pointer class based on `<foreign-pointer>`. It is suitable when the C structure is mostly passed around using pointers; most typically, when the foreign library allocates the structure and returns the pointer to the Scheme world.

scheme-name is a Scheme variable name. This will be bound to a newly-created subclass of `<foreign-pointer>` to represent this C-ptr type.

c-type is the type of the C pointer we wrap.

c-name is the C variable name (of type `ScmClass *`). In initialization code, an instance of a class (the same one bound to *scm-name* in the Scheme world) will be stored in this C variable.

c-pred is a macro name to determine if a `ScmObj` is of this type. *c-boxer* is a macro name to wrap C pointer and return a `ScmObj` *c-unboxer* is a macro name to extract C pointer from a `ScmObj`

The only supported qualifier is `:private`, which will generate *c-pred*, *c-boxer* and *c-unboxer* definitions automatically. Otherwise those definitions must be provided elsewhere.

The two supported flags are

- `:keep-identity` (which is `SCM_FOREIGN_POINTER_KEEP_IDENTITY` in the C world) keeps a weak hash table that maps the wrapped C pointer to the wrapping `ScmObj`, so `Scm_MakeForeignPointer` (i.e. *c-boxer* when `:private` is used) returns eq? object if the same C pointer is given.

  This incurs some overhead, but cleanup procedure can safely free the foreign object without worrying if there's other ScmObj that's pointing to the same C pointer.

  Do not use this flag if the C pointer is also allocated by `GC_malloc`. The used hash table is only weak for its value, so the C pointer wouldn't be GCed.
- `:map-null` (which is `SCM_FOREIGN_POINTER_MAP_NULL` in the C world) makes `Scm_MakeForeignPointer` (i.e. *c-boxer* when `:private` is used) return `SCM_FALSE` when the C pointer is `NULL`.

`define-symbol` *scheme-name* [*c-name*] [Stub Form]

Defines a Scheme symbol. No Scheme binding is created. When *c-name* is given, the named C variable points to the created `ScmSymbol`.

`define-variable` *scheme-name initializer* [Stub Form]
Defines a Scheme variable.

`define-constant` *scheme-name initializer* [Stub Form]
Defines a Scheme constant.

`define-enum` *name* [Stub Form]
A `define-constant` specialized for enum values. This is useful for exporting C enums to Scheme.

`define-enum-conditionally` *name* [Stub Form]
Abbreviation of `(if "defined(name)" (define-enum name))`

`define-cise-stmt` *name clause* ... [Stub Form]
`define-cise-expr` *name clause* ... [Stub Form]
`define-cfn` ... [Stub Form]
`declare-cfn` ... [Stub Form]
`define-cvar` ... [Stub Form]
`declare-cvar` ... [Stub Form]
`define-ctype` ... [Stub Form]
`.define` ... [Stub Form]
`.if` ... [Stub Form]
`.include` ... [Stub Form]
`.undef` ... [Stub Form]
`.unless` ... [Stub Form]
`.when` ... [Stub Form]
Cise macro definitions (see Section 9.3.4 [C in S expression], page 331).

`initcode` *c-code* [Stub Form]
Insert *c-code* literally in the initialization function

`declcode` *stmt* ... [Stub Form]
Inserts declaration code. *stmt* is usually `.include` or other preprocessor statements but it could also be a string which is treated as C fragments.

`begin` *form* ... [Stub Form]
Treat each *form* as if they are toplevel stub forms.

`if` *test then-stmt* [*else-stmt*] [Stub Form]
`when` *test stmt* [Stub Form]
Deprecated. Please use `.if` and `.when` instead.

`include` *file* [Stub Form]
Include and evaluate another stub file.

9.4 `gauche.charconv` - Character Code Conversion

`gauche.charconv` [Module]
This module defines a set of functions that converts character encoding schemes (CES) of the given data stream.

This module is implicitly loaded when `:encoding` keyword argument is given to the file stream creating functions (such as `open-input-file` and `call-with-output-file`).

As of release 0.5.6, Gauche natively supports conversions between typical Japanese character encodings: ISO2022JP, ISO2022JP-3, EUC-JP (EUC-JISX0213), Shift_JISX0213, UTF-8 (Unicode 3.2). Conversions between other encodings are handled by `iconv(3)`. See Section 9.4.1 [Supported character encoding schemes], page 340, for details.

9.4.1 Supported character encoding schemes

A CES is represented by its name as a string or a symbol. Case is ignored. There may be several aliases defined for a single encoding.

A CES name "none" is special. When Gauche's native encoding is `none`, Gauche just treats a string as a byte sequence, and it's up to the application to interpret the sequence in an appropriate encoding. So, conversion to and from CES "none" does nothing.

You can check whether the specific conversion is supported on your system or not, by the following function.

`ces-conversion-supported?` *from-ces to-ces* [Function]

{`gauche.charconv`} Returns `#t` if conversion from the character encoding scheme (CES) *from-ces* to *to-ces* is supported in this system.

Note that this procedure may return true even if system only supports partial conversion between *from-ces* and *to-ces*. In such case, actual conversion might lose information by coercing characters in *from-ces* which are not supported in *to-ces*. (For example, conversion from Unicode to EUC-JP is "supported", although Unicode has characters that are not in EUC-JP).

Also note that this procedure always returns `#t` if *from-ces* and/or *to-ces* is "none", for conversion to/from CES "none" always succeeds (in fact, it does nothing).

```
;; see if you can convert the internal encoding to EUC-JP
(ces-conversion-supported? (gauche-character-encoding) "euc-jp")
```

Also there are two useful procedures to deal with CES names.

`ces-equivalent?` *ces-a ces-b :optional unknown-value* [Function]

{`gauche.charconv`} Returns true if two CESes *ces-a* and *ces-b* are equivalent to the knowledge of the system. Returns false if they are not. If the system doesn't know about equivalency, *unknown-value* is returned, whose default is `#f`.

CES "none" works like a wild card; it is "equivalent" to any CES. (Thus, `ces-equivalent?` is not transitive. The intended use of `ces-equivalent?` is to compare two given CES names and see if conversion is required or not).

```
(ces-equivalent? 'eucjp "EUC-JP")          ⇒ #t
(ces-equivalent? 'shift_jis "EUC-JP")      ⇒ #f
(ces-equivalent? "NoSuchEncoding" 'utf-8 '?) ⇒ ?
```

`ces-upper-compatible?` *ces-a ces-b :optional unknown-value* [Function]

{`gauche.charconv`} Returns true if a string encoded in CES *ces-b* can also be regarded as a string encoded in *ces-a* without conversion, to the knowledge of the system. Returns false if not. Returns *unknown-value* if the system can't determine which is the case.

Like `ces-equivalent?`, CES "none" works like a wildcard. It is upper-compatible to any CES, and any CES is upper-compatible to "none".

```
(ces-upper-compatible? "eucjp" "ASCII")             ⇒ #t
(ces-upper-compatible? "eucjp" "utf-8")             ⇒ #f
(ces-upper-compatible? "utf-8" "NoSuchEncoding" '?) ⇒ ?
```

Conversion between common japanese CESes (EUC_JP, Shift JIS, UTF-8 and ISO2022-JP) of the character set JIS X 0201 and JIS X 0213 is handled by Gauche's built-in algorithm (see below for details). When other CES name is given, Gauche uses `iconv(3)` if it is linked.

When Gauche's conversion routine encounters a character that can't be mapped, it replaces the character for "geta mark" (U+3013) if it's a multibyte character in the input encoding, or for '?' if it's a singlebyte character in the input encoding. If that happens in iconv, handling of such character depends on iconv implementation (glibc implementation returns an error).

If the conversion routine encounters an input sequence that is illegal in the input CES, an error is signaled.

Details of Gauche's native conversion algorithm: Between EUC_JP, Shift JIS and ISO2022JP, Gauche uses arithmetic conversion whenever possible. This even maps the undefined codepoint properly. Between Unicode (UTF-8) and EUC_JP, Gauche uses lookup tables. Between Unicode and Shift JIS or ISO2022JP, Gauche converts the input CES to EUC_JP, then convert it to the output CES. If the same CES is specified for input and output, Gauche's conversion routine just copies input characters to output characters, without checking the validity of the encodings.

`EUC_JP, EUCJP, EUCJ, EUC_JISX0213`
: Covers ASCII, JIS X 0201 kana, JIS X 0212 and JIS X 0213 character sets. JIS X 0212 character set is supported merely because it uses the code region JIS X 0213 doesn't use, and JIS X 0212 characters are not converted properly to Shift JIS and UTF-8. Use JIS X 0213.

`SHIFT_JIS, SHIFTJIS, SJIS`
: Covers Shift_JISX0213, except that 0x5c and 0x7e is mapped to ASCII character set (REVERSE SOLIDUS and TILDE), instead of JIS X 0201 Roman (YEN SIGN and OVERLINE).

`UTF-8, UTF8`
: Unicode 3.2. Note that some JIS X 0213 characters are mapped to Extension B (U+20000 and up). Some JIS X 0213 characters are mapped to two unicode characters (one base character plus a combining character).

`ISO2022JP, CSISO2022JP, ISO2022JP-1, ISO2022JP-2, ISO2022JP-3`
: These encodings differ a bit (except ISO2022JP and CSISO2022JP, which are synonyms), but Gauche handles them same. If one of these CES is specified as input, Gauche recognizes escape sequences of any of CES. ISO2022JP-2 defines several non-Japanese escape sequences, and they are recognized by Gauche, but mapped to substitution character ('?' or geta mark).

  For output, Gauche assumes ISO2022JP first, and uses ISO2022JP-1 escape sequence to put JIS X 0212 character, or uses ISO2022JP-3 escape sequence to put JIS X 0213 plane 2 character. Thus, if the string contains only JIS X 0208 characters, the output is compatible to ISO2022JP. Precisely speaking, JIS X 0213 specifies some characters in JIS X 0208 codepoint that shouldn't be mixed with JIS X 0208 characters; Gauche output those characters as JIS X 0208 for compatibility. (This is the same policy as Emacs-Mule's iso2022jp-3-compatible mode).

9.4.2 Autodetecting the encoding scheme

There are cases that you don't know the CES of the input, but you know it is one of several possible encodings. The charconv module has a mechanism to guess the input encoding. There can be multiple algorithms, and each algorithm has the name (wildcard CES). Right now, there's only one algorithm implemented:

`"*JP"`
: To guess the character encoding from japanese text, among either ISO2022-JP(-1,2,3), EUCJP, SHIFT_JIS or UTF-8.

The wildcard CES can be used in place of CES name for some conversion functions.

`ces-guess-from-string` *string scheme* [Function]

{`gauche.charconv`} Guesses the CES of *string* by the character guessing scheme *scheme* (e.g. `"*JP"`). Returns CES name that can be used by other charconv functions. It may return `#f` if the guessing scheme finds no possible encoding in *string*. Note that if there may be more than one possible encoding in *string*, the guessing scheme returns one of them, usually in favor of the native CES.

9.4.3 Conversion ports

`open-input-conversion-port` *source from-code :key to-code buffer-size owner?* [Function]

{`gauche.charconv`} Takes an input port *source*, which feeds characters encoded in *from-code*, and returns another input port, from which you can read characters encoded in *to-code*.

If *to-code* is omitted, the native CES is assumed.

buffer-size is used to allocate internal buffer size for conversion. The default size is about 1 kilobytes and it's suitable for typical cases.

If you don't know the *source*'s CES, you can specify CES guessing scheme, such as `"*JP"`, in place of *from-code*. The conversion port tries to guess the encoding, by prefetching the data from *source* up to the buffer size. It signals an error if the code guessing routine finds no appropriate CES. If the guessing routine finds ambiguous input, however, it silently assume one of possible CES's, in favor of the native CES. Hence it is possible that the guessing is wrong if the buffer size is too small. The default size is usually enough for most text documents, but it may fail if the large text contains mostly ASCII characters and multibyte characters appear only at the very end of the document. To be sure for the worst case, you have to specify the buffer size large enough to hold entire text.

By default, `open-input-conversion-port` leaves *source* open. If you specify true value to *owner?*, the function closes *source* after it reads EOF from the port.

For example, the following code copies a file `unknown.txt` to a file `eucjp.txt`, converting unknown japanese CES to EUC-JP.

```
(call-with-output-file "eucjp.txt"
  (lambda (out)
    (copy-port (open-input-conversion-port
                 (open-input-file "unknown.txt")
                 "*jp"              ;guess code
                 :to-code "eucjp"
                 :owner? #t)        ;close unknown.txt afterwards
               out)))
```

`open-output-conversion-port` *sink to-code :key from-code buffer-size owner?* [Function]

{`gauche.charconv`} Creates and returns an output port that converts given characters from *from-code* to *to-code* and feed to an output port *sink*. If *from-code* is omitted, the native CES is assumed. You can't specify a character guessing scheme (such as "*JP") to neither *from-code* nor *to-code*.

buffer-size specifies the size of internal conversion buffer. The characters put to the returned port may stay in the buffer, until the port is explicitly flushed (by `flush`) or the port is closed.

By default, the returned port doesn't closes *sink* when itself is closed. If a keyword argument *owner?* is provided and true, however, it closes *sink* when it is closed.

`ces-convert-to` *return-type source from-code :optional to-code* [Function]
`ces-convert` *source from-code :optional to-code* [Function]

{`gauche.charconv`} Convert *source*, which is a string or an u8vector of multibyte encoding in *from-code*, to a string or u8vector encoded in *to-code*. If *to-code* is omitted, the native CES is assumed.

In `ces-convert-to`, you can specify the return type by *return-type* argument; it must be either a class object `<string>` or `<u8vector>`. On the other hand, `ces-convert` always returns a string, regardless of the type of *source*.

If *to-code* is different from the native CES and a string is returned, it can be an incomplete string. It's for the backward compatibility—in general, we recommend to use u8vector to represent multibyte sequence in CES other than the native encoding.

from-code can be a name of character guessing scheme (e.g. "*JP").

`call-with-input-conversion` *iport proc :key encoding conversion-buffer-size* [Function]
`call-with-output-conversion` *oport proc :key encoding conversion-buffer-size* [Function]

{`gauche.charconv`} These procedures can be used to perform character I/O with different encoding temporary from the original port's encoding.

`call-with-input-conversion` takes an input port *iport* which uses the character encoding *encoding*, and calls *proc* with one argument, a conversion input port. From the port, *proc* can read characters in Gauche's internal encoding. Note that once *proc* is called, it has to read all the characters until EOF; see the note below.

`call-with-output-conversion` takes an output port *oport* which expects the character encoding *encoding*, and calls *proc* with one argument, a temporary conversion output port. To the port, *proc* can write characters in Gauche's internal encoding. When *proc* returns, or it exits with an error, the temporary conversion output port is flushed and closed. The caller of `call-with-output-conversion` can continue to use *oport* with original encoding afterwards.

Both procedure returns the value(s) that *proc* returns. The default value of *encoding* is Gauche's internal encoding. Those procedures don't create a conversion port when it is not necessary. If *conversion-buffer-size* is given, it is used as the *buffer-size* argument when the conversion port is open.

You shouldn't use *iport*/*oport* directly while *proc* is active—character encoding is a stateful process, and mixing I/O from/to the conversion port and the underlying port will screw up the state.

Note: for the `call-with-input-conversion`, you can't use *iport* again unless *proc* reads EOF from it. It's because a conversion port needs to buffer the input, and there's no way to undo the buffered input to *iport* when *proc* returns.

`with-input-conversion` *iport thunk :key encoding conversion-buffer-size* [Function]
`with-output-conversion` *oport thunk :key encoding conversion-buffer-size* [Function]

{`gauche.charconv`} Similar to `call-with-*-conversion`, but these procedures call *thunk* without arguments, while the conversion port is set as the current input or output port, respectively. The meaning of keyword arguments are the same as `call-with-*-conversion`.

`wrap-with-input-conversion` *port from-code :key to-code owner? buffer-size* [Function]
`wrap-with-output-conversion` *port to-code :key from-code owner? buffer-size* [Function]

{`gauche.charconv`} Convenient procedures to avoid adding unnecessary conversion port. Each procedure works like `open-input-conversion-port` and `open-output-conversion-port`, respectively, except if system knows no conversion is needed, no conversion port is created and *port* is returned as is.

When a conversion port is created, *port* is always owned by the port. When you want to close the port, always close the port returned by `wrap-with-*-conversion`, instead the original *port*. If you close the original *port* first, the pending conversion won't be flushed. (Some conversion requires trailing sequence that is generated only when the conversion port is closing, so simply calling `flush` isn't enough.)

The *buffer-size* argument is passed to the `open-*-conversion-port`.

9.5 `gauche.collection` - Collection framework

`gauche.collection` [Module]

This module provides a set of generic functions (GFs) that iterate over various collections. The Scheme standard has some iterative primitives such as `map` and `for-each`, and `scheme.list` (see Section 10.3.1 [R7RS lists], page 512, adds a rich set of such functions, but they work only on lists.

Using the method dispatch of the object system, this module efficiently extends those functions for other collection classes such as vectors and hash tables. It also provides a simple way for user-defined class to adapt those operations. So far, the following operations are defined.

Mapping
: `fold`, `fold2`, `fold3`, `map`, `map-to`, `map-accum`, `for-each`

Selection and searching
: `find`, `find-min`, `find-max`, `find-min&max`, `filter`, `filter-to`, `remove`, `remove-to`, `partition`, `partition-to` `group-collection`

Conversion
: `coerce-to`

Miscellaneous
: `size-of`, `lazy-size-of`

Fundamental iterator creator
: `call-with-iterator`, `call-with-builder`, `with-iterator`, `with-builder`, `call-with-iterators`.

Those operations work on *collections* and its subclass, *sequences*. A collection is a certain form of a set of objects that you can traverse all the object in it in a certain way. A sequence is a collection that all its elements are ordered, so that you can retrieve its element by index.

The following Gauche built-in objects are treated as collections and/or sequences.

`<list>`
: A sequence.

`<vector>`
: A sequence.

`<string>`
: A sequence (of characters)

`<hash-table>`
: A collection. Each element is a pair of a key and a value.

`<s8vector>, <u8vector>, ... <f64vector>`
: A sequence (methods defined in `gauche.uvector` module, see Section 9.36 [Uniform vectors], page 476).

See Section 9.29 [Sequence framework], page 441, for it adds more sequence specific methods.

The methods that needs to return a set of objects, i.e. `map`, `filter`, `remove` and `partition`. returns a list (or lists). The corresponding "-to" variant (`map-to`, `filter-to`, `remove-to` and `partition-to`. takes a collection class argument and returns the collection of the class.

9.5.1 Mapping over collection

These generic functions extends the standard mapping procedures. See also Section 9.29.3 [Mapping over sequences], page 442, if you care the index as well as elements.

`fold` *proc knil coll coll2 . . .* [Generic function]

{`gauche.collection`} This is a natural extension of *fold* (see Section 6.6.6 [Other list procedures], page 134).

For each element *Ei* in the collection *coll*, *proc* is called as (*proc Ei Ri-1*), where *Ri-1* is the result of (*i*-1)-th invocation of *proc* for *i* > 0, and *R0* is *knil*. Returns the last invocation of *proc*.

```
(fold + 0 '#(1 2 3 4)) ⇒ 10
(fold cons '() "abc")  ⇒ (#\c #\b #\a)
```

If the *coll* is a sequence, it is guaranteed that the elements are traversed in order. Otherwise, the order of iteration is undefined.

Note: We don't provide `fold-right` on collections, since the order of elements doesn't matter, so only `fold` is sufficient for meaningful traversal. However, sequences do have `fold-right`; see Section 9.29.3 [Mapping over sequences], page 442.

You can fold more than one collection, although it doesn't make much sense unless all of the collections are sequences. Suppose `E(k, i)` for *i*-th element of *k*-th collection. *proc* is called as

```
(proc E(0,i) E(1,i) ... E(K-1,i) Ri-1)
```

Different types of collections can be mixed together.

```
(fold acons '() "abc" '#(1 2 3))
  ⇒ ((#\c 3) (#\b 2) (#\a 1))

;; calculates dot product of two vectors
(fold (lambda (a b r) (+ (* a b) r)) 0
      '#(3 5 7) '#(2 4 6))
  ⇒ 68
```

When more than one collection is given, `fold` terminates as soon as at least one of the collections exhausted.

`fold2` *proc knil1 knil2 coll coll2 . . .* [Generic function]

`fold3` *proc knil1 knil2 knil3 coll coll2 . . .* [Generic function]

{`gauche.collection`} Like `fold`, but they can carry two and three state values instead of one, respectively. The state values are initialized by *knilN*. The procedure *proc* is called with each element of *collN*, and the state values. It must return two (`fold2`) or three (`fold3`) values, which will be used as the state values of next iteration. The values returned in the last iteration will be the return values of `fold2` and `fold3`.

```
(fold2 (lambda (elt a b) (values (min elt a) (max elt b)))
       256 0 '#u8(33 12 142 1 74 98 12 5 99))
 ⇒ 1 and 142  ;; find minimum and maximum values
```

See also `map-accum` below.

`map` *proc coll coll2 . . .* [Generic function]

{`gauche.collection`} This extends the built-in `map` (see Section 6.6.5 [Walking over lists], page 131). Apply *proc* for each element in the collection *coll*, and returns a list of the results.

If the *coll* is a sequence, it is guaranteed that the elements are traversed in order. Otherwise, the order of iteration is undefined.

If more than one collection is passed, *proc* is called with elements for each collection. In such case, `map` terminates as soon as at least one of the collection is exhausted. Note that passing more than one collection doesn't make much sense unless all the collections are sequences.

```
(map (lambda (x) (* x 2)) '#(1 2 3))
  ⇒ #(2 4 6)

(map char-upcase "abc")
  ⇒ (#\A #\B #\C)

(map + '#(1 2 3) '#(4 5 6))
  ⇒ (5 7 9)
```

`map` *always* returns a list. If you want to get the result in a different type of collection, use `map-to` described below. If you wonder why `(map char-upcase "abc")` doesn't return `"ABC"`, read the discussion in the bottom of this subsection.

`map-to` *class proc coll coll2 . . .* [Generic function]

{`gauche.collection`} This works the same as `map`, except the result is returned in a collection of class *class*. `Class` must be a collection class and have a builder interface (see Section 9.5.4 [Fundamental iterator creators], page 349).

```
(map-to <vector> + '#(1 2 3) '#(4 5 6))
  ⇒ #(5 7 9)

(map-to <string> char-upcase "def")
  ⇒ "DEF"

(map-to <vector> char=? "bed" "pet")
  ⇒ #(#f #t #f)
```

`map-accum` *proc seed coll1 coll2 . . .* [Generic function]

{`gauche.collection`} Collects results of *proc* over collections, while passing a state value. *proc* is called like this:

```
(proc elt1 elt2 ... seed)
```

Where *elt1 elt2 . . .* are the elements of *coll1 coll2 . . .* . It must return two values; the first value is collected into a list (like `map`), while the second value is passed as *seed* to the next call of *proc*.

When one of the collections is exhausted, `map-accum` returns two values, the list of the first return values from *proc*, and the second return value of the last call of *proc*.

If the given collections are sequences, it is guaranteed that *proc* is applied in order of the sequence.

This is similar to Haskell's `mapAccumL`, but note that the order of `proc`'s argument and return values are reversed.

`for-each` *proc coll coll2 . . .* [Generic function]

{`gauche.collection`} Extension of built-in `for-each` (see Section 6.6.5 [Walking over lists], page 131). Applies *proc* for each elements in the collection(s). The result of *proc* is discarded. The return value of `for-each` is undefined.

If the *coll* is a sequence, it is guaranteed that the elements are traversed in order. Otherwise, the order of iteration is undefined.

If more than one collection is passed, *proc* is called with elements for each collection. In such case, `for-each` terminates as soon as one of the collection is exhausted. Note that passing more than one collection doesn't make much sense unless all the collections are sequences.

`fold$` *proc* [Generic Function]
`fold$` *proc knil* [Generic Function]
`map$` *proc* [Generic Function]
`for-each$` *proc* [Generic Function]

{`gauche.collection`} Partial-application version of `fold`, `map` and `for-each`.

Discussion: It is debatable what type of collection `map` should return when it operates on the collections other than lists. It may seem more "natural" if `(map * '#(1 2) '#(3 4))` returns a vector, and `(map char-upcase "abc")` returns a string.

Although such interface seems work for simple cases, it'll become problematic for more general cases. What type of collection should be returned if a string and a vector are passed? Furthermore, some collection may only have iterator interface but no builder interface, so that the result can't be coerced to the argument type (suppose you're mapping over database records, for example). And Scheme programmers are used to think `map` returns a list, and the result of `map` are applied to the procedures that takes list everywhere.

So I decided to add another method, `map-to`, to specify the return type explicitly The idea of passing the return type is taken from CommonLisp's `map` function, but taking a class metaobject, `map-to` is much flexible to extend using method dispatch. This protocol ("-to" variant takes a class metaobject for the result collection) is used throughout the collection framework.

9.5.2 Selection and searching in collection

`find` *pred coll* [Generic function]

{`gauche.collection`} Applies *pred* for each element of a collection *coll* until *pred* returns a true value. Returns the element on which *pred* returned a true value, or `#f` if no element satisfies *pred*.

If *coll* is a sequence, it is guaranteed that *pred* is applied in order. Otherwise the order of application is undefined.

```
(find char-upper-case? "abcDe") ⇒ #\D
(find even? '#(1 3 4 6)) ⇒ 4
(find even? '(1 3 5 7))  ⇒ #f
```

`find-min` *coll :key key compare default* [Generic function]
`find-max` *coll :key key compare default* [Generic function]

{`gauche.collection`} Returns a minimum or maximum element in the collection *coll*.

A one-argument procedure *key*, whose default is `identity`, is applied for each element to obtain a comparison value. Then a comparison value is compared by a two-argument procedure *compare*, whose default is `<`. If the collection has zero or one element, the *compare* procedure is never called.

When the collection is empty, a value given to *default* is returned, whose default is `#f`.

```
(find-min '((a . 3) (b . 9) (c . -1) (d . 7)) :key cdr) ⇒ (c . -1)
```

`find-min&max` *coll :key key compare default default-min default-max* [Generic function]

{`gauche.collection`} Does `find-min` and `find-max` simultaneously, and returns two values, the minimum element and the maximum element. The keyword arguments *key*, *compare*, and *default* are the same as `find-min` and `find-max`. Alternatively you can give default values for minimum and maximum separately, by *default-min* and *default-max*.

`filter` *pred coll* [Generic function]

{`gauche.collection`} Returns a list of elements of collection *coll* that satisfies the predicate *pred*. If the collection is a sequence, the order is preserved in the result.

```
(filter char-upper-case? "Hello, World")
```

```
  ⇒ (#\H #\W)
(filter even? '#(1 2 3 4)) ⇒ (2 4)
```

`filter-to` *class pred coll* [Generic function]

{`gauche.collection`} Same as `filter`, but the result is returned as a collection of class *class*.

```
(filter-to <vector> even? '#(1 2 3 4)) ⇒ #(2 4)
(filter-to <string> char-upper-case? "Hello, World")
  ⇒ "HW"
```

`remove` *pred coll* [Generic function]

{`gauche.collection`} Returns a list of elements of collection *coll* that does not satisfy the predicate *pred*. If the collection is a sequence, the order is preserved in the result.

```
(remove char-upper-case? "Hello, World")
  ⇒ (#\e #\l #\l #\o #\, #\space #\o #\r #\l #\d)
(remove even? '#(1 2 3 4)) ⇒ (1 3)
```

`remove-to` *class pred coll* [Generic function]

{`gauche.collection`} Same as `remove`, but the result is returned as a collection of class *class*.

```
(remove-to <vector> even? '#(1 2 3 4)) ⇒ #(1 3)
(remove-to <string> char-upper-case? "Hello, World")
  ⇒ "ello, orld"
```

`partition` *pred coll* [Generic function]

{`gauche.collection`} Does `filter` and `remove` the same time. Returns two lists, the first consists of elements of the collection *coll* that satisfies the predicate *pred*, and the second consists of elements that doesn't.

```
(partition char-upper-case? "PuPu")
  ⇒ (#\P #\P) and (#\u #\u)
(partition even? '#(1 2 3 4))
  ⇒ (2 4) and (1 3)
```

`partition-to` *class pred coll* [Generic function]

{`gauche.collection`} Same as `partition`, except the results are returned in the collections of class *class*.

```
(partition-to <string> char-upper-case? "PuPu")
  ⇒ "PP" and "uu"
(partition-to <vector> even? '#(1 2 3 4))
  ⇒ #(2 4) and #(1 3)
```

`group-collection` *coll :key key test* [Generic function]

{`gauche.collection`} Generalized `partition`. Groups elements in *coll* into those who has the same key value, and returns the groups as of lists. Key values are calculated by applying the procedure *key* to each element of *coll*. The default value of *key* is `identity`. For each element of *coll*, *key* is applied exactly once. The equal-ness of keys are compared by *test* procedure, whose default is `eqv?`.

If *coll* is a sequence, then the order of elements in each group of the result is the same order in *coll*.

```
(group-collection '(1 2 3 2 3 1 2 1 2 3 2 3))
  ⇒ ((1 1 1) (2 2 2 2 2) (3 3 3 3))
```

```
(group-collection '(1 2 3 2 3 1 2 1 2 3 2 3) :key odd?)
  ⇒ ((1 3 3 1 1 3 3) (2 2 2 2 2))

(group-collection '(("a" 2) ("b" 5) ("c" 1) ("b" 3) ("a" 6))
  :key car :test string=?)
  ⇒ ((("a" 2) ("a" 6)) (("b" 5) ("b" 3)) (("c" 1)))
```

See also `group-sequence` in `gauche.sequence` (see Section 9.29.4 [Other operations over sequences], page 443), which only groups adjacent elements.

9.5.3 Miscellaneous operations on collection

`size-of` *coll* [Generic function]

{`gauche.collection`} Returns the number of elements in the collection. Default method iterates over the collection to calculate the size, which is not very efficient and may diverge if the collection is infinite. Some collection classes overload the method for faster calculation.

`lazy-size-of` *coll* [Generic function]

{`gauche.collection`} Returns either the size of the collection, or a promise to calculate it. The intent of this method is to avoid size calculation if it is expensive. In some cases, the caller wants to have size just for optimization, and it is not desirable to spend time to calculate the size. Such caller uses this method and just discards the information if it is a promise.

`coerce-to` *class coll* [Generic function]

{`gauche.collection`} Convert a collection *coll* to another collection which is an instance of *class*. If *coll* is a sequence and *class* is a sequence class, the order is preserved.

```
(coerce-to <vector> '(1 2 3 4))
  ⇒ #(1 2 3 4)

(coerce-to <string> '#(#\a #\b #\c))
  ⇒ "abc"
```

9.5.4 Fundamental iterator creators

These are fundamental methods on which all the rest of iterative method are built. The method interface is not intended to be called from general code, but suitable for building other iterator construct. The reason why I chose this interface as fundamental methods are explained at the bottom of this subsection.

`call-with-iterator` *collection proc :key start* [Generic function]

{`gauche.collection`} A fundamental iterator creator. This creates two procedures from *collection*, both take no argument, and then call *proc* with those two procedures. The first procedure is terminate predicate, which returns `#t` if the iteration is exhausted, or `#f` if there are still elements to be visited. The second procedure is an incrementer, which returns one element from the collection and sets the internal pointer to the next element. The behavior is undefined if you call the incrementer after the terminate predicate returns `#t`.

If the collection is actually a sequence, the incrementer is guaranteed to return elements in order, from 0-th element to the last element. If a keyword argument *start* is given, however, the iteration begins from *start*-th element and ends at the last element. If the collection is not a sequence, the iteration order is arbitrary, and *start* argument has no effect.

An implementation of *call-with-iterator* method may limit the extent of the iterator inside the dynamic scope of the method. For example, it allocates some resource (e.g. connect to a database) before calling *proc*, and deallocates it (e.g. disconnect from a database) after *proc* returns.

This method returns the value(s) *proc* returns.

```
(call-with-iterator '(1 2 3 4 5)
  (lambda (end? next)
    (do ((odd-nums 0))
        ((end?) odd-nums)
      (when (odd? (next)) (inc! odd-nums)))))
 ⇒ 3
```

See also `with-iterator` macro below, for it is easier to use.

`with-iterator` (*collection end? next args* ...) *body* ... [Macro]

{`gauche.collection`} A convenience macro to call `call-with-iterator`.

```
(with-iterator (coll end? next args ...) body ...)
 ≡
(call-with-iterator coll
  (lambda (end? next) body ...)
   args ...)
```

`call-with-iterators` *collections proc* [Function]

{`gauche.collection`} A helper function to write n-ary iterator method. This function applies `call-with-iterator` for each *collections*, and makes two lists, the first consists of terminate predicates and the second of incrementers. Then *proc* is called with those two lists. Returns whatever *proc* returns.

`call-with-builder` *collection-class proc :key size* [Generic function]

{`gauche.collection`} A fundamental builder creator. Builder is a way to construct a collection incrementally. Not all collection classes provide this method.

Collection-class is a class of the collection to be built. This method creates two procedures, adder and getter, then calls *proc* with those procedures. Adder procedure takes one argument and adds it to the collection being built. Getter takes no argument and returns a built collection object. The effect is undefined if adder is called after getter is called.

A keyword argument *size* may be specified if the size of the result collection is known. Certain collections may be built much more efficiently if the size is known; other collections may just ignore it. The behavior is undefined if more than *size* elements are added, or the collection is retrieved before *size* elements are accumulated.

If the collection class is actually a sequence class, adder is guaranteed to add elements in order. Otherwise, the order of elements are insignificant.

Some collection class may take more keyword arguments to initialize the collection.

This method returns the value(s) *proc* returned.

```
(call-with-builder <list>
  (lambda (add! get)
    (add! 'a) (add! 'b) (add! 'c) (get)))
 ⇒ (a b c)

(call-with-builder <vector>
  (lambda (add! get)
    (add! 'a) (add! 'b) (add! 'c) (get)))
 ⇒ #(a b c)
```

See also `with-builder` macro below, for it is much easier to use.

`with-builder` (*collection add! get args* ...) *body* ... [Macro]

{`gauche.collection`} A convenience macro to call `call-with-builder`.

```
(with-builder (coll add! get args ...) body ...)
```

```
  ≡
(call-with-builder coll
  (lambda (add! get) body ...)
  args ...)
```

Discussion: Other iterator methods are built on top of call-with-iterator and call-with-builder. By implementing those methods, you can easily adapt your own collection class to all of those iterative operations. Optionally you can overload some of higher-level methods for efficiency.

It is debatable that which set of operations should be primitives. I chose call-with-iterator style for efficiency of the applications I see most. The following is a discussion of other possible primitive iterators.

`fold`
: It is possible to make `fold` a primitive method, and build other iterator method on top of it. Collection-specific iterating states can be kept in the stack of `fold`, thus it runs efficiently. The method to optimize a procedure that uses `fold` as a basic iterator construct. However, it is rather cumbersome to derive generator-style interface from it. It is also tricky to iterate irregularly over more than one collections.

CPS
: Passes iteratee the continuation procedure that continues the iteration. The iteratee just returns when it want to terminate the iteration. It has resource management problem described in Oleg Kiselyov's article ([OLEG2], page 831).

Iterator object
: Like C++ iterator or Common Lisp generator. Easy to write loop. The problem is that every call of checking termination or getting next element must be dispatched.

Series
: Common Lisp's series can be very efficient if the compiler can statically analyze the usage of series. Unfortunately it is not the case in Gauche. Even if it could, the extension mechanism doesn't blend well with Gauche's object system.

Macros
: Iterator can be implemented as macros, and that will be very efficient; e.g. Scheme48's iterator macro. It uses macros to extend, however, and that doesn't blend well with Gauche's object system.

The current implementation is close to the iterator object approach, but using closures instead of iterator objects so that avoiding dispatching in the inner loop. Also it allows the iterator implementor to take care of the resource problem.

9.5.5 Implementing collections

The minimum requirements of the collection class implementation is as follow:

- The class inherits `<collection>` abstract class.
- A method `call-with-iterator` is implemented.

This makes iterator methods such as `map`, `for-each`, `find` and `filter` to work.

In order to make the constructive methods (e.g. `map-to` to create your collection), you have to implement `call-with-builder` method as well. Note that `call-with-builder` method must work a sort of class method, dispatched by class, rather than normal method dispatched by instance. In Gauche, you can implement it by using a metaclass. Then the minimal code will look like this:

```
(define-class <your-collection-meta> (<class>) ())

(define-class <your-collection> (<collection>)
 (...) ;; slots
```

```
      :metaclass <your-collection-meta>)

    (define-method call-with-iterator
        ((coll <your-collection>) proc . options)
      ...
      )

    (define-method call-with-builder
        ((coll <your-collection-meta>) proc . options)
      ...
      )
```

Optionally, you can overload other generic functions to optimize performance.

9.6 gauche.config - Configuration parameters

`gauche.config` [Module]

This module allows the Scheme program to access the configuration information the same as you can get from the `gauche-config` program.

`gauche-config` *option* [Function]

{`gauche.config`} Returns the configured value of the *option*.

See the manpage of `gauche-config`, or run `gauche-config` without any argument from the shell, to find out the valid options.

```
(gauche-config "--cc")
  ⇒ "gcc"
(gauche-config "-L")
  ⇒ "-L/usr/lib/gauche/0.6.5/i686-pc-linux-gnu"
(gauche-config "-l")
  ⇒ "-ldl -lcrypt -lm -lpthread"
```

9.7 gauche.configure - Generating build files

`gauche.configure` [Module]

This is a utility library to write a `configure` script. It is used to check the system properties and generates build files (usually `Makefile`) from templates.

The primary purpose is to replace autoconf-generated `configure` shell scripts in Gauche extension packages.

The advantage of using autoconf is that it generates a script that runs on most vanilla unix, for it only uses minimal shell features and basic unix commands. However, when you configure Gauche extension, you sure have Gauche already, so you don't need to limit yourself with minimal environment.

Writing a `configure` script directly in Gauche means developers don't need an extra step to generate `configure` before distribution. They can directly check in `configure` in the source repo, and anybody who pulls the source tree can run `configure` at once without having autoconf.

Currently, `gauche.configure` only covers small subset of autoconf, though, so if you need to write complex tests you may have to switch back to autoconf. We'll add tests as needed.

The core feature of `gauche.configure` is the ability to generate files (e.g. `Makefile`) from templates (e.g. `Makefile.in`) with replacing parameters. We follow autoconf convention, so the substitution parameters in a template is written like @VAR@. You should be able to reuse Makefile.in used for autoconf without changing them.

The API corresponds to autoconf's AC_* macros, while we use cf- prefix instead.

9.7.1 Structure of configure script and build files

A configure script tests running system's properties to determine values of substitution parameters, then read one or more template build files, and write out one output build file for each, replacing substitution parameters for the assigned values.

By convention, a template file has a suffix .in, and the corresponding output file is named without the suffix. For example, Makefile.in is a template that generates Makefile.

Templates may contain substitution parameters, noted @PARAMETER_NAME@. This is a fragment of a typical Makefile template:

```
GAUCHE_PACKAGE = "@GAUCHE_PACKAGE@"
SOEXT          = @SOEXT@
LOCAL_PATHS    = "@LOCAL_PATHS@"

foo.$(SOEXT): $(foo_SRCS)
        $(GAUCHE_PACKAGE) compile \
          --local=$(LOCAL_PATHS) --verbose foo $(foo_SRCS)
```

When processed by configure, @GAUCHE_PACKAGE@, @SOEXT@ and @LOCAL_PATHS@ are replaced with appropriate values. If you know autoconf, you are already familiar with this.

The Gauche configure script is structurally similar to autoconf's configure.in, but you can use full power of Scheme. The following is the minimal configure script:

```
#!/usr/bin/env gosh
(use gauche.configure)
(cf-init-gauche-extension)
(cf-output-default)
```

This script does several common tasks. The cf-init-gauche-extension does the following:

- First, it handles command-line arguments given to configure. In the default settings, it recognizes standard configure arguments such as --prefix, and --with-local=PATH:PATH:... which adds PATH/includes and PATH/libs to the header and library search paths. You can handle more arguments by adding cf-arg-with and cf-arg-enable before cf-init-gauche-extension.
- Then it reads package.scm. Package name and version are taken from it. Dependencies are also checked.
- It sets up global environment to run other configure checks.
- It sets up default values for standard substitution parameters such as @prefix@.

And cf-output-default does the following:

- Generate gpd (Gauche package description) file.
- Writes package version to VERSION file.
- Scan Makefile.in's in the source directory and its subdirectories, and process them to generate Makefiles. If config header files (typically config.h) are specified by cf-config-headers, process input files (e.g. config.h.in) to generate the header files.

In general, a configure script consists of the following parts:

1. Extra argument declarations (optional): Declare --with-PACKAGE and/or --enable-FEATURE options you want to handle, by cf-with-arg and cf-enable-arg, respectively.

2. Initialization. Call to `cf-init` or `cf-init-gauche-extension` sets up global context and parses command-line arguments passed to `configure`. It also process package metainformation in `package.scm`, if it exists.
3. Tests and other substitution parameter settings (optional): Check system characteristics and sets up substitution parameters and/or C preprocessor definitions.
4. Output generation. Call `cf-output` or `cf-output-default` to process template files.

Most `cf-*` API corresponds to autoconf's `AC_*` or `AS_*` macros. We need argument declarations before `cf-init` so that it can generate help message including custom arguments in one pass.

9.7.2 Configure API

Initialization

`cf-init-gauche-extension` [Function]

{`gauche.configure`} This is a convenience API that packages several boilerplate `cf-*` function calls in one call. This must be called exactly once in a configure script.

Specifically, it calls `cf-arg-with` to process `--with-local`, then calls `cf-init` with no arguments to initialize, then sets the following substitution parameters:

`GOSH` Path to `gosh`.

`GAUCHE_CONFIG`
Path to `gauche-config`.

`GAUCHE_PACKAGE`
Path to `gauche-package`.

`GAUCHE_INSTALL`
Path to `gauche-install`.

`GAUCHE_CESCONV`
Path to `gauche-cesconv`.

`GAUCHE_PKGINCDIR`
Result of `gauche-config --pkgincdir`

`GAUCHE_PKGLIBDIR`
Result of `gauche-config --pkglibdir`

`GAUCHE_PKGARCHDIR`
Result of `gauche-config --pkgarchdir`

The `--with-local` command-line argument should take a parameter, which is a colon-separated list of paths. It becomes a value of substitution parameter `LOCAL_PATHS`. The default `Makefile.in` template passes its value to `gauche-package` via the `--local` argument. When the C extension is compiled and linked, the `include` and `lib` subdirectories of the given paths are searched for headers and libraries, respectively. For example, if the extension requires a library foo and you install it under `/opt/foo` (that is, headers in `/opt/foo/include` and library objects in `/opt/foo/lib`), then you can pass `--with-local=/opt/foo` to configure the extension.

If you don't like these default behavior, you can call individual `cf-*` functions instead. See `cf-init` below.

`cf-init` *:optional package-name package-version maintainer-email homepage-url* [Function]

{`gauche.configure`} Initialize the configure system. Corresponds to autoconf's `AC_INIT`. This must be called once in the configure script, before any feature-test procedures. (If you call `cf-init-gauche-extension`, `cf-init` is called from it.)

First, it checks if a file named `package.scm` is in the same directory as the configure script, and reads the Gauche package description from it. The package description contains package name, version, dependencies, etc. See Section 9.21 [Package metainformation], page 411, for the details.

It then parses the command-line arguments, sets up the configure environment, and (if `package.scm` defines dependencies) check if the system has required packages.

The optional arguments are only supported for the backward compatibility if you don't have `package.scm`. You need at least to provide *package-name* and *package-version* to tell what package you're configuring. They are used as the value of substitution parameter `PACKAGE_NAME` and `PACKAGE_VERSION`. The other optional arguments, *maintainer-email* and *homepage-url*, are used to initialize `PACKAGE_BUGREPORT` and `PACKAGE_URL`. These arguments are compatible to autoconf's `AC_INIT` macro.

We recommend to always use `package.scm` and omit all the optional arguments, because it allows you to maintain the package metainformation in one place. When `package.scm` is read, `PACKAGE_BUGREPORT` is initialized by the first entry of `maintainers` slot of the package description, and `PACKAGE_URL` is initialized by its `homepage` slot. See Section 9.21 [Package metainformation], page 411, for description of slots of the package description.

Note that if there's `package.scm` *and* you provide the optional arguments, they must match, or `cf-init` raises an error. It is to catch errors during transition in which you forgot to update either one.

This procedure sets up a bunch of standard substitution parameters such as `prefix`, `bindir` or `srcdir`. To see what substitution parameters are set, you can call `cf-show-substs` after `cf-init`.

Command-line arguments

`cf-arg-enable` *feature help-string :optional proc-if-given proc-if-not-given* [Function]

`cf-arg-with` *package help-string :optional proc-if-given proc-if-not-given* [Function]

{`gauche.configure`} Make the configure script accept feature selection argument and package selection argument, respectively. The corresponding autoconf macros are `AC_ARG_ENABLE` and `AC_ARG_WITH`.

Those procedures must be called before calling `cf-init` or `cf-init-gauche-extension`.

The *feature* and *package* arguments must be a symbol.

A feature selection argument is in a form of either `--enable-`*`feature`*`=`*`val`*, `--enable-`*`feature`*, or `--disable-`*`feature`*. The latter two are equivalent to `--enable-`*`feature`*`=yes` and `--enable-`*`feature`*`=no`, respectively. It is to select an optional feature provided with the package itself.

A package selection argument is in a form of either `--with-`*`package`*`=`*`val`*, `--with-`*`package`* and `--without-`*`package`*. The latter two are equivalent to `--with-`*`package`*`=yes` and `--with-`*`package`*`=no`, respectively. It is to select an external software package to be used with this package.

When `cf-init` finds these arguments, it adds entry of *feature* or *package* to the global tables, with the value *val*. Those global tables can be accessed with `cf-feature-ref` and `cf-package-ref` procedures below.

The *help-string* argument must be a string and is used as is to list the help of the option in part of usage message displayed by `configure --help`. You can use `cf-help-string` below to create a help string that fits nicely in the usage message.

If optional *proc-if-given* argument is given, it must be a procedure that accepts one argument, *val*. It is called when `cf-init` finds one of those arguments.

If optional *proc-if-not-given* argument is given, it must be a procedure that accepts no arguments. It is called when `cf-init` doesn't find any of those arguments.

The following is to accept `--with-local=`*PATH*`:`*PATH*`:....` (This `cf-arg-with` call is included in `cf-init-gauche-extension`). Note that the help string (the second argument) is generated by `cf-help-string` call. The command-line parameter followed by `--with-local` is passed as the argument of the procedure in the third argument:

```
(cf-arg-with 'local
             (cf-help-string
              "--with-local=PATH:PATH..."
              "For each PATH, add PATH/include to the include search
  paths and PATH/lib to the library search paths.  Useful if you have some
  libraries installed in non-standard places. ")
             (^[with-local]
               (unless (member with-local '("yes" "no" ""))
                 (cf-subst 'LOCAL_PATHS with-local)))
             (^[] (cf-subst 'LOCAL_PATHS "")))
```

`cf-help-string` *item description* [Function]

{`gauche.configure`} Return a string formatted suitable to show as an option's help message. The result can be passed to *help-string* argument of `cf-arg-enable` and `cf-arg-with`. This corresponds to autoconf's `AS_HELP_STRING`.

Call it as follows, and it'll indent and fill the description nicely.

```
(cf-help-string "--option=ARG" "Give ARG as the value of option")
```

`cf-feature-ref` *name* [Function]
`cf-package-ref` *name* [Function]

{`gauche.configure`} Lookup a symbol *name* from the global feature table and the global package table, respectively. These can be called after `cf-init`.

For example, if you've called `cf-arg-enable` with `foofeature`, and the user has invoked the `configure` script with `--with-foofeature=full`, then `(cf-feature-ref 'foofeature)` returns `"full"`. If the user hasn't given the command-line argument, `#f` is returned.

If you add or change the value of features or packages, you can use generalized `set!` as `(set! (cf-feature-ref 'NAME) VALUE)` etc.

Messages

The `cf-init` procedure opens the default log drain that goes to `config.log`, and you can use `log-format` to write to it (See Section 9.16 [User-level logging], page 393, for the details of logging).

However, to have consistent message format conveniently, the following procedures are provided. They emits the message both to log files and the current output port (in slightly different formats so that the console messages align nicely visually.)

`cf-msg-checking` *fmt arg* . . . [Function]

{`gauche.configure`} Writes out "checking XXX..." message. The *fmt* and *arg* . . . arguments are passed to `format` to produce the "XXX" part (see Section 6.22.8.4 [Formatting output], page 235).

For the current output port, this does not emit the trailing newline, expecting `cf-msg-result` will be called subsequently.

Here's an excerpt of the source that uses `cf-msg-checking` and `cf-msg-result`:

```
(define (compiler-can-produce-executable?)
  (cf-msg-checking "whether the ~a compiler works" (~ (cf-lang)'name))
  (rlet1 result ($ run-compiler-with-content
                   (cf-lang-link-m (cf-lang))
                   (cf-lang-null-program-m (cf-lang)))
    (cf-msg-result (if result "yes" "no"))))
```

This produces a console output like this:

```
checking whether the C compiler works... yes
```

while the log file records more info:

```
checking: whether the C compiler works
... whatever logging message from run-compiler-with-content ...
result: yes
```

This corresponds to autoconf's `AC_MSG_CHECKING`.

`cf-msg-result` *fmt arg ...* [Function]

{`gauche.configure`} The *fmt* and *arg ...* are passed to `format`, and the formatted message and newline is written out (see Section 6.22.8.4 [Formatting output], page 235). For the log file, it records "result: XXX" where XXX is the formatted message. Supposed to be used with `cf-msg-checking`.

This corresponds to autoconf's `AC_MSG_RESULT`.

`cf-msg-notice` *fmt arg ...* [Function]

{`gauche.configure`} Produces formatted message to both console and log. Newline is added. This corresponds to autoconf's `AC_MSG_NOTICE`.

`cf-msg-warn` *fmt arg ...* [Function]
`cf-msg-error` *fmt arg ...* [Function]

{`gauche.configure`} Produces "Warning: XXX" and "Error: XXX" messages, respectively. The *fmt* and *arg ...* are passed to `format` to generate XXX part (see Section 6.22.8.4 [Formatting output], page 235). These corresponds to autoconf's `AC_MSG_WARN` and `AC_MSG_ERROR`.

`cf-echo` *arg ...* [> *file*][>> *file*] [Function]

{`gauche.configure`} Convenience routine to replace shell's `echo` command.

If the argument list ends with `> file` or `>> file`, where *file* is a string file name, then this works just like shell's `echo`; that is, *arg*s except the last two are written to *file*, space separated, newline terminated. Using `>` supersedes *file*, while `>>` appends to it.

If the argument list doesn't end with those redirection message, it writes out the argument to both the current output port and the log file, space separated, newline terminated. For the log file, the message is prefixed with "Message:".

Parameters and definitions

The configure script maintains two global tables, definition tables and parameter tables. Definition tables is used for C preprocessor definitions, and parameter tables are used for `@PARAMETER@` substitutions. (Do not confuse substitution parameters with Scheme's parameter objects (see Section 9.22 [Parameters], page 411)).

`cf-define` *symbol :optional value* [Function]

{`gauche.configure`} Registers C preprocessor definition of *symbol* with *value*. *Value* can be any Scheme objects, but it is emitted to a command line (in `-DSYMBOL=VALUE` form) or in `config.h` (in `#define SYMBOL VALUE` form) using `display`, so you want to avoid including funny characters. If *value* is omitted, `1` is assumed.

NB: To `#define` a string value, e.g. `#define FOO "foo"`, you have to call as `(cf-define 'FOO "\"foo\"")`.

This corresponds to autoconf's `AC_DEFINE`.

`cf-subst` *symbol value* [Function]

{`gauche.configure`} Registers a substitution parameter *symbol* with *value*. *Value* can be any Scheme objects; it's `display` representation is used to substitute *@SYMBOL@* in the template.

This corresponds to autoconf's `AC_SUBST`, but we require the value (while autoconf can refer to the shell variable value as default).

`cf-subst-prepend` *symbol value :optional delim default* [Function]

`cf-subst-append` *symbol value :optional delim default* [Function]

{`gauche.configure`} Prepend or append *value* to the substitution parameter *symbol*, using delimiter *delim*. If the substitution parameter isn't defined, *value* becomes the sole value of the parameter, except when *default* is given and not an empty string. If *delim* is omitted, single whitespace is used.

`with-cf-subst` ((*symbol value*) ...) ... [Special Form]

{`gauche.configure`} Temporarily replace substitution parameters with new values. This could be useful for example to run some compilation tests with different parameters

```
(with-cf-subst ((LIBS "-L<path> -l<lib>"))
  (cf-try-compile-and-link ...))
```

`cf-have-subst?` *symbol* [Function]

{`gauche.configure`} Returns true iff `symbol` is registered as a substitution parameter by `cf-subst`.

`cf-arg-var` *symbol* [Function]

{`gauche.configure`} Lookup the environment variable `symbol` and if it is found, use its value as the substitution value. For example, if you call `(cf-arg-var 'MYCFLAGS)`, then the user can provide the value of `@MYCFLAGS@` as `MYCFLAGS=-g ./configure`.

This corresponds to autoconf's `AC_ARG_VAR`, but we lack the ability of setting the help string. That's because `cf-arg-var` must be run after `cf-init`, but the help message is constructed within `cf-init`.

`cf-ref` *symbol :optional default* [Function]

{`gauche.configure`} This looks up the value of the substitution parameter *symbol*. If there's no such substitution parameter registered, it returns *default* when it's provided, otherwise throws an error.

`cf$` *symbol* [Function]

{`gauche.configure`} Looks up the value of the substitution parameter *cf-ref*, but it returns empty string if it's unregistered. Useful to use within string interpolation, e.g. `#"gosh ~(cf$'GOSHFLAGS)"`.

Predefined tests

`cf-check-prog` *sym prog-or-progs :key value default paths filter* [Function]
`cf-path-prog` *sym prog-or-progs :key value default paths filter* [Function]

{`gauche.configure`} Check if a named executable program exists in search paths, and if it exists, sets the substitution parameter *sym* to the name of the found program. The name to search is specified by *prog-or-progs*, which is either a string or a list of strings.

The difference of `cf-check-prog` and `cf-path-prog` is that `cf-check-prog` uses the basename of the found program, while `cf-path-prog` uses its full path. These corresponds to autoconf's `AC_CHECK_PROG`, `AC_CHECK_PROGS`, `AC_PATH_PROG` and `AC_PATH_PROGS`.

For example, the following feature test searches either one of `cc`, `gcc`, `tcc` or `pcc` in `PATH` and sets the substitution parameter `MY_CC` to the name of the found one.

```
(cf-check-prog 'MY_CC '("cc" "gcc" "tcc" "pcc"))
```

If multiple program names is given, the search is done in the following order: First, we search for the first item (`cc`, in the above example) for each of paths, then the second, etc. For example, if we have `/usr/local/bin:/usr/bin:/bin` in `PATH` and we have `/usr/local/bin/tcc` and `/usr/bin/gcc`, the above feature test sets `MY_CC` to `"gcc"`. If you use `cf-path-prog` instead, `MY_CC` gets `"/usr/bin/gcc"`.

If no program is found, *sym* is set to the keyword argument *default* if it is given, otherwise *sym* is left unset.

If the *value* keyword argument is given, its value is used instead of the found program name to be set to *sym*.

The list of search paths is taken from `PATH` environment variable. You can override the list by the *paths* keyword argument, which must be a list of directory names. It may contain nonexistent directory names, which are silently skipped.

The *filter* keyword argument, if given, must be a predicate that takes full pathname of the executable program. It is called when the procedure finds matching executable; the *filter* procedure may reject it by returning `#f`, in which case the procedure keeps searching.

Note: If the substitution parameter *sym* is already set at the time these procedure is called, these procedures do nothing. Combined with `cf-arg-var`, it allows the configure script caller to override the feature test. For example, suppose you have the following in the `configure` script:

```
(cf-arg-var 'GREP)
(cf-path-prog 'GREP '("egrep" "fgrep" "grep"))
```

A user can override the test by calling `configure` like this:

```
$ ./configure GREP=mygrep
```

`cf-prog-cxx` [Function]

{`gauche.configure`} A convenience feature test to find C++ compiler. This searches popular names of C++ compilers from the search paths, sets the substitution parameter `CXX` to the compiler's name, then tries to compile a small program with it to see it can generate an executable.

This corresponds to autoconf's `AC_PROG_CXX`.

`CXX` is `cf-arg-var`'ed in this procedure. If a user provide the value when he calls `configure`, the searching is skipped, but the check of generating an executable is still performed.

If the substitution parameter `CXXFLAGS` is set, its value is used to check if the compiler can generate an executable. `CXXFLAGS` is `cf-arg-var`'ed in this procedure.

This procedure also emulates autoconf's `AC_PROG_CXX` behavior— if `CXX` is not set, but `CCC` is set, then we set `CXX` by the value of `CCC` and skip searching.

`cf-check-header` *header :key includes* [Function]

{`gauche.configure`} Check if a header file *header* exists and usable, by compiling a source program of the current language that includes the named header file. This is intended to be used as a predicate—returns `#t` if the header is usable, `#f` if not. This corresponds to autoconf's `AC_CHECK_HEADER`.

If *header* requires other headers being included or preprocessor symbols defined before it, you can pass a list of strings to be emitted before the check in the *includes* keyword arguments. The given strings are just concatenated and used as a C program fragment. The default value is provided by `cf-includes-default`.

The following example sets C preprocessor symbol `HAVE_CRYPT_H` to 1 if `crypt.h` is available. (Note: For this kind of common task, you can use `cf-check-headers` below. The advantage of using `cf-check-header` is that you can write other actions in Scheme depending on the result.)

```
(when (cf-check-header "crypt.h")
  (cf-define "HAVE_CRYPT_H" 1))
```

`cf-check-headers` *headers :key includes if-found if-not-found* [Function]

{`gauche.configure`} Codify a common pattern of checking the availability of headers and sets C preprocessor definitions. This corresponds to autoconf's `AC_CHECK_HEADERS`.

See this example:

```
(cf-check-headers '("unistd.h" "stdint.h" "inttypes.h" "rpc/types.h"))
```

This checks availability of each of listed headers, and sets C preprocessor definition `HAVE_UNISTD_H`, `HAVE_STDINT_H`, `HAVE_INTTYPES_H` and `HAVE_RPC_TYPES_H` to 1 if the corresponding header file is available.

A list of strings given to *includes* are emitted to the C source file before the inclusion of the testing header. You can give necessary headers and/or C preprocessor definitions there; if omitted, `cf-includes-default` provides the default list of such headers.

The keyword argument *if-found* and *if-not-found* are procedures to be called when a header is found to be available or to be unavailable, respectively. The procedure receives the name of the header.

The name of the C preprocessor definition is derived from the header name by upcasing it and replacing non-alphanumeric characters for `_`. Note that this substitution is not injective: Both `gdbm/ndbm.h` and `gdbm-ndbm.h` yield `GDBM_NDBM_H`. If you need to distinguish such files you have to use `cf-check-header`.

`cf-includes-default` [Function]

{`gauche.configure`} Returns a list of strings that are included in the check program by default. It is actually a combination of C preprocessor `#ifdef`s and `#include`s, and would probably be better to be called `cf-prologue-default` or something, but the corresponding autoconf macro is `AC_INCLUDES_DEFAULT` so we stick to this name.

Usually you don't need to call this explicitly. Not giving the `includes` argument to `cf-check-header` and `cf-check-headers` will make `cf-includes-default` called implicitly.

`cf-check-type` *type :key includes* [Function]

{`gauche.configure`} Tests if *type* is defined as a type name. This can be used as predicate—it returns `#t` if *type* is defined, `#f` otherwise. This corresponds to `AC_CHECK_TYPE`.

A list of strings given to *includes* are emitted to the C source file before the inclusion of the testing header. You can give necessary headers and/or C preprocessor definitions there; if omitted, `cf-includes-default` provides the default list of such headers.

`cf-check-types` *types :key includes if-found if-not-found* [Function]

{`gauche.configure`} For each type in the list *types*, call `cf-check-type` to see it is defined as a type. If it is, defines `HAVE_type`, and calls *if-found* with the type as an argument if provide. If the type is not defined and *if-not-found* is provided, calls it with the type as an argument. This corresponds to autoconf's `AC_CHECK_TYPES`.

The argument *includes* is passed to `cf-check-type`.

```
;; May define HAVE_PTRDIFF_T and/or HAVE_UNSIGNED_LONG_LONG_INT
;; depending on its availability:
(cf-check-types '("ptrdiff_t" "unsigned long long int"))

;; Example of using includes to add an extra header.
(cf-check-types '("float_t")
                :includes `(,@(cf-includes-default)
                            "#include <math.h>\n"))
```

`cf-check-decl` *symbol :key includes* [Function]

{`gauche.configure`} Tests if *symbol* is declared as a cpp macro, a variable, a constant, or a function. This can be used as predicate—it returns `#t` if *type* is defined, `#f` otherwise. This corresponds to autoconf's `AC_CHECK_DECL`.

A list of strings given to *includes* are emitted to the C source file before the inclusion of the testing header. You can give necessary headers and/or C preprocessor definitions there; if omitted, `cf-includes-default` provides the default list of such headers.

`cf-check-decls` *symbols :key includes if-found if-not-found* [Function]

{`gauche.configure`} For each symbol in *symbols*, call `cf-check-decl` to see if it is declared. If it is, define `HAVE_DECL_symbol` to 1, and calls *if-found* with the symbol if provided. If it is not declared, define `HAVE_DECL_symbol` to 0, and calls *if-not-found* with the symbol if provided. This corresponds to autoconf's `AC_CHECK_DECLS`.

The argument *includes* is passed to `cf-check-decl`.

Note that, unlike other `cf-check-*` routines which leave `HAVE_*` macro undefined when the item isn't found, this one always defines the macro and differentiate the result with its value. This behavior is the same as `AC_CHECK_DECLS`.

`cf-check-member` *aggregate.member :key includes* [Function]

{`gauche.configure`} The *aggregate.member* argument is a string of aggregate type name and its member concatenated by a dot, e.g. `"struct password.pw_gecos"`. It can also be a submember, e.g. `"struct foo.bar.baz"`. The *aggregate* part can be any type name (typedef-ed name is ok).

This corresponds to autoconf's `AC_CHECK_MEMBER`.

This test checks if *member* is a member of *aggregate*, and returns `#t` if so, or returns `#f` if not. This procedure is intended to be used as a predicate; it doesn't have side effects.

A list of strings given to *includes* are emitted to the C source file before the inclusion of the testing header. You can give necessary headers and/or C preprocessor definitions there; if omitted, `cf-includes-default` provides the default list of such headers.

`cf-check-members` *members :key includes if-found if-not-found* [Function]

{`gauche.configure`} For each *aggregate.member* in *members*, call `cf-check-member`. If the test passes, defines `HAVE_aggregate_member`, and calls *if-found* with *aggregate.member* if provided. If the test fails, calls *if-not-found* with *aggregate.member* if provided.

This corresponds to autoconf's `AC_CHECK_MEMBERS`.

The *include* argument is passed to `cf-check-member`.

```
;; Defines HAVE_STRUCT_ST_RDEV and/or HAVE_STRUCT_ST_BLKSIZE
;; depending on their availability:
(cf-check-members '("struct stat.st_rdev"
                    "struct stat.st_blksize"))
```

`cf-check-func` *func* [Function]

{`gauche.configure`} See if a function *func* is available. This emits C code to call *func* (with dummy declaration) and tries to compile and link, using current value of substitution parameter `LIBS`.

This procedure is meant to be used as predicate; it returns `#t` if *func* is available, `#f` otherwise.

This corresponds to autoconf's `AC_CHECK_FUNC`.

`cf-check-funcs` *funcs :key if-found if-not-found* [Function]

{`gauche.configure`} For each function name *func* in *funcs*, call `cf-check-func` to determine availability. If it is available, define `HAVE_func`, and calls `if-found` with *func* if provided. If it is not available, calls `if-not-found` with *func* if provided. This corresponds to autoconf's `AC_CHECK_FUNCS`.

`cf-check-lib` *lib fn :key other-libs if-found if-not-found* [Function]

{`gauche.configure`} See if a library *lib* can be linked and a function *fn* in it is callable. This corresponds to autoconf's `AC_CHECK_LIB`. Give the name you pass after `-l` option to *lib*; for example, if you want to check availability of `libm`, you can say as follows:

```
(cf-check-lib "m" "sin")
```

This generates a C source that calls *fn* and try to compile and link it to generate executable. If linking *lib* requires additional libraries, it should be listed in *other-libs*:

```
(cf-check-lib "Xt" "XtDisplay" :other-libs '("-lX11" "-lSM" "-lICE"))
```

If compilation and linking succeeds, *if-found* is called at the tail position with the library name (`"m"` and `"Xt"` in the above examples, respectively) as the argument. The default behavior is to add `-llib` in the left of substitution parameter `LIBS`, and set `HAVE_LIBlib` definition, then returns `#t`.

If compilation or linking fails, *if-not-found* is called at the tail position with the library name. The default behavior is to return `#f`.

The default behavior of *if-found* and *if-not-found* allows `cf-check-lib` to be used as predicate as well. For example, you can write like this:

```
(unless (cf-check-lib "foo" "foo_fn)
  ... do something if libfoo isn't available ...)
```

`cf-search-libs` *fn libs :key other-libs if-found if-not-found* [Function]

{`gauche.configure`} Like `cf-check-lib`, but can be used if you're not sure which library contains desired function. This corresponds to autoconf's `AC_SEARCH_LIBS`. Note that this takes function name first, while `cf-check-lib` takes function name second—blame autoconf for this inconsistency.

First it tests if *fn* is available without any library in *libs* (that is, with the ones already in `LIBS` and specified in *other-libs*). If not, it tests each library in *libs* in turn.

If *fn* is found, *if-found* is called at the tail position, with the name of the library as an argument (if *fn* is available without any library, the argument is `#f`). If omitted, and a library is required, then the library is added to the substitution parameter `LIBS`, and `HAVE_LIBlib` is defined. The default procedure returns `#t`.

If *fn* isn't found in any of the libraries, *if-not-found* is called at the tail position with `#f` as the argument. The default procedure does nothing and just returns `#f`.

The default behavior of *if-found* and *if-not-found* allows `cf-search-libs` to be used as predicate as well.

Running compiler

The `gauche.configure` module provides a generic mechanism to construct a small test program, compile it, and run it. Currently we only support `C` and `C++`; we'll add support for other languages as needed.

`cf-lang` [Parameter]
{`gauche.configure`}

`cf-lang-program` *prologue body* [Function]
{`gauche.configure`} Returns a string tree that consists a stand-alone program for the current language. *Prologue* and *body* must be a string tree. *Prologue* comes at the beginning of the source, and *body* is included in the part of the program that's executed. If the current language is C, the code fragment:

```
(use text.tree)
(write-tree (cf-lang-program "#include <stdio.h>\n" "printf(\"()\");\n"))
```

would produce something like this:

```
#include <stdio.h>

int main(){
printf("()");

; return 0;
}
```

`cf-lang-io-program` [Function]
{`gauche.configure`} This is a convenience routine. It returns a string tree of a program in the current language, that creates a file named `conftest.out`, then exits with zero status on success, or nonzero status on failure.

`cf-lang-call` *prologue func-name* [Function]
{`gauche.configure`}

`cf-try-compile` *prologue body* [Function]
{`gauche.configure`}

`cf-try-compile-and-link` *prologue body* [Function]
{`gauche.configure`}

Output

`cf-output-default` *file* ... [Function]
{`gauche.configure`} A convenience routine to produce typical output. It does the following:

- Generate gpd (Gauche package description) file using `cf-make-gpd`.
- Generate `VERSION` file that contains the value of `PACKAGE_VERSION` substitution parameter.
- Search `Makefile.in`'s under the source directory (the value of substitution parameter `srcdir`), and process them to produce `Makefile`'s. If *file* ... are given, ***file*.in** are also processed as well to produce *file*.

  See `cf-output` below for the details.
- If config header is registered by `cf-config-headers`, process them as well.

`cf-output` *file ...* [Function]

{`gauche.configure`} Generates *file*'s from the input templates. This corresponds to autoconf's `AC_OUTPUT`.

For each *file*, a file named ***`file`***`.in` is read as a template. Within the file, `@PARAMETER@` is substituted with the value of `(cf$ 'PARAMETER)`. If the named parameter isn't registered, a warning is issued and the parameter is left unsubstituted.

If config headers are not registered via `cf-config-headers`, a substitution parameter `DEFS` is replaced with all the definitions in the form of `-D...`. For example, if you have checked header files `foo/bar.h` and `foo/baz.h`, `DEFS` gets the value `-DHAVE_FOO_BAR_H -DHAVE_FOO_BAZ_H`.

If config header is registered by `cf-config-headers`, they are processed as well. In such case, the substitution parameter `DEFS` gets the value `-DHAVE_CONFIG_H`.

`cf-config-headers` *header-or-headers* [Function]

{`gauche.configure`} Sets up config header files to be processed. Usually a config header file is named `config.h`, and contains definitions determined by feature tests.

The *header-or-headers* argument may be a string header-spec or a list of string header-specs, where each header spec is a header file name (e.g. `"config.h"`) or a header name and a input file name concatenated with a colon (e.g. `"config.h:config.h.templ"`). If it's just a header name, input file name is assumed to be the header file name with `".in"` appended.

The input template of config header file contains a bunch of `#undef` directives, such as the following:

```
/* Gauche ABI version string */
#undef GAUCHE_ABI_VERSION

/* Define if Gauche handles multi-byte character as EUC-JP */
#undef GAUCHE_CHAR_ENCODING_EUC_JP

/* Define if Gauche handles multi-byte character as Shift JIS */
#undef GAUCHE_CHAR_ENCODING_SJIS

/* Define if Gauche handles multi-byte character as UTF-8 */
#undef GAUCHE_CHAR_ENCODING_UTF_8
```

Once processed, the generated header file has either `#undef` line is replaced with `#define`, or commented out, depending on the definitions determined by feature tests.

```
/* Gauche ABI version string */
#define GAUCHE_ABI_VERSION "0.97"

/* Define if Gauche handles multi-byte character as EUC-JP */
/* #undef GAUCHE_CHAR_ENCODING_EUC_JP */

/* Define if Gauche handles multi-byte character as Shift JIS */
/* #undef GAUCHE_CHAR_ENCODING_SJIS */

/* Define if Gauche handles multi-byte character as UTF-8 */
#define GAUCHE_CHAR_ENCODING_UTF_8 /**/
```

Note that the lines other than `#undef` are copied as they are.

The substitution parameter `DEFS` behaves differently whether config header is specified or not. If no config header is registered, The value of `DEFS` is a C command-line arguments for definitions, e.g. `-DGAUCHE_ABI_VERSION=0.97 -DGAUCHE_CHAR_ENCODING_UTF8`. If config header files are registered, the value of `DEFS` becomes simply `-DHAVE_CONFIG_H`.

`cf-show-substs` *:key formatter* [Function]

{`gauche.configure`} Print all substitution parameters; this is for debugging.

For each substitution parameter name and value, *formatter* is called with them; the default is `(^[k v] (format #t "~16s ~s" k v))`.

`cf-make-gpd` [Function]

{`gauche.configure`} Generate gpd (Gauche package description) file, `PACKAGE_NAME.gpd`, where `PACKAGE_NAME` is the package's name either taken form `package.scm` or the argument to `cf-init`. See Section 9.21 [Package metainformation], page 411, for the package description file format.

9.8 gauche.connection - Connection framework

`gauche.connection` [Module]

A connection is an abstract class to handle full-duplex communication channel. The actual data I/O is done through Scheme ports; a channel provides an interface to retrieve those ports. It also has interface to know endpoint names, and the way to shutdown the communication.

The `<socket>` class, for example, implements the connection framework (see Section 9.20 [Networking], page 398). The `<tls>` class, which wraps a socket to provide TLS communication, also implements it. That means you can write a client code that works both plain socket connection and secure connection. You can also abstract communication to the external process as a connection (see Section 9.25.5 [Process connection], page 433).

Each of the connection endpoints (*self* for our side, and *peer* fo the other side) has *addresses*, some object that identifies the endpoint. The framework doesn't specify the actual type of addresses; it only requires that addresses can be passed to `connection-address-name` method to get its string representation, so that can be used for logging and monitoring. The concrete class can choose suitable address representation. For example, `<socket>` uses `<sockaddr>` for the addresses.

A concrete class must implement the following methods.

```
connection-self-address (c <connection>)
connection-peer-address (c <connection>)
connection-input-port (c <connection>)
connection-output-port (c <connection>)
connection-shutdown (c <connection>) how
connection-close (c <connection>)
connection-address-name obj                ; optional
```

At this moment this framework doesn't provide a generic way to *create* a connection, since the way to do it may greatly vary depending on the concrete implementation. Each concrete implementation should provide its own procedure to create and return a new connection.

`connection-self-address` *connection* [Method]
`connection-peer-address` *connection* [Method]

Returns the address of this *connection*'s endpoint and its peer's. For sockets and `<tls>`, it is an instance of `<sockaddr>`. For processes, it is a string describing the process.

If *connection* is not connected, these methods can return `#f`.

The returned value (other than `#f`) must be able to be passed to `connection-address-name` method.

Currently addresses are only used for logging purpose within the connection framework; however, we may enhance the framework to add "connect" operation, for example, so the concrete class is encouraged to use objects that can be used to create connections.

`connection-address-name` *address* [Method]

This method returns a string representation of *address*, which is a returned value from `connection-self-address` and `connection-peer-address`. The default method just uses *address*'s `display`-representation. If a concrete class chooses aggregate objects to represent addresses, it should provide this method as well.

`connection-input-port` *connection* [Method]
`connection-output-port` *connection* [Method]

Returns an input port and an output port to read from and write to the connection, respectively.

If the connection is not connected, or already shutdown or closed, the return value of these methods are unspecified.

`connection-shutdown` *connection :optional how* [Method]

Shutdown the connection. You can either shutdown the connection entirely at once, or shutdown only read or write channel of it.

Shutdown is about telling the peer to terminate the communication. Usually the peer will detect the termination by reading EOF from their input channel. The port corresponding to the shutdown channel is closed so no further communication is possible.

Note that merely closing the connection doesn't shutdown the connection—the process may fork after creating the connection, and in that case, one process may close the connection without shutting down.

The *how* argument must be one of those symbols:

`read` Shutdown read channel of the connection.

`write` Shutdown write channel of the connection.

`both` Shutdown both channels of the connection.

If omitted, *both* is assumed.

Shutting down already shut down channel has no effect.

Note: Some concrete implementation of connection may not allow to shutdown each channels independently. In such case, the connection is shut down entirely regardless of *how* argument.

The one-argument case is handled by the default method (it calls two-argument method with `both` as *how*). So the concrete class only need to define two argument method.

`connection-close` *connection* [Method]

Close the connection. This destroys the connection and frees local resources. Note that this does not shutdown the connection itself. If this connection is the only endpoint of this side, the peer will get an error when it tries to communicate.

See `connection-shutdown` for the details of shutting down a connection.

9.9 `gauche.dictionary` - Dictionary framework

`gauche.dictionary` [Module]

A dictionary is an abstract class for objects that can map a key to a value. This module provides some useful generic functions for dictionaries, plus generic dictionary classes built on top of other dictionary classes.

9.9.1 Generic functions for dictionaries

These generic functions are useful to implement algorithms common to any dictionary-like objects, a data structure that maps discrete, finite set of keys to values. (Theoretically we can think of continuous and/or infinite set of keys, but implementation-wise it is cleaner to limit the dictionary

Among built-in classes, `<hash-table>` and `<tree-map>` implement the dictionary interface. All the `<dbm>` classes provided by `dbm` module also implement it.

To make your own class implement the dictionary interface, you have to provide at least `dict-get`, `dict-put!`, `dict-delete!`, `dict-fold` and `dict-comparator`. (You can omit `dict-delete!` if the datatype doesn't allow deleting entries.) Other generic functions have default behavior built on top of these. You can implement other methods as well, potentially to gain better performance.

(Note: Dictionaries are also collections, so you can use collection methods as well; for example, to get the number of entries, just use `size-of`).

`dict-get` (*dict* `<dictionary>`) *key :optional default* [Generic function]
{`gauche.dictionary`} Returns the value corresponding to the *key*. If the dictionary doesn't have an entry with *key*, returns *default* when it is provided, or raises an error if not.

`dict-put!` (*dict* `<dictionary>`) *key value* [Generic function]
{`gauche.dictionary`} Puts the mapping from *key* to *value* into the dictionary.

`(setter dict-get)` (*dict* `<dictionary>`) *key value* [Generic function]
{`gauche.dictionary`} This works the same as `dict-put!`.

`dict-exists?` (*dict* `<dictionary>`) *key* [Generic function]
{`gauche.dictionary`} Returns `#t` if the dictionary has an entry with *key*, `#f` if not.

`dict-delete!` (*dict* `<dictionary>`) *key* [Generic function]
{`gauche.dictionary`} Removes an entry with *key* form the dictionary. If the dictionary doesn't have such an entry, this function is noop.

`dict-clear!` (*dict* `<dictionary>`) [Generic function]
{`gauche.dictionary`} Empties the dictionary. Usually this is much faster than looping over keys to delete them one by one.

`dict-comparator` (*dict* `<dictionary>`) [Generic function]
{`gauche.dictionary`} Should return a comparator used to compare keys.

`dict-fold` (*dict* `<dictionary>`) *proc seed* [Generic function]
{`gauche.dictionary`} Calls a procedure *proc* over each entry in a dictionary *dict*, passing a seed value. Three arguments are given to *proc*; an entry's key, an entry's value, and a seed value. Initial seed value is *seed*. The value returned from *proc* is used for the seed value of the next call of *proc*. The result of the last call of *proc* is returned from *dict-fold*.

If *dict* is `<ordered-dictionary>`, *proc* is called in the way to keep the following associative order, where the key is ordered from *K0* (minimum) to *Kn* (maximum), and the corresponding values is from *V0* to *Vn*:

```
(proc Kn Vn (proc Kn-1 Vn-1 ... (proc K0 V0 seed)))
```

`dict-fold-right` (*dict* `<ordered-dictionary>`) *proc seed* [Generic function]
{`gauche.dictionary`} Like `dict-fold`, but the associative order of applying *proc* is reversed as follows:

```
(proc K0 V0 (proc K1 V1 ... (proc Kn Vn seed)))
```

This generic function is only defined on `<ordered-dictionary>`.

dict-for-each (*dict* <dictionary>) *proc* [Generic function]

{gauche.dictionary} Calls *proc* with a key and a value of every entry in the dictionary *dict*. For ordered dictionaries, *proc* is guaranteed to be called in the increasing order of keys.

dict-map (*dict* <dictionary>) *proc* [Generic function]

{gauche.dictionary} Calls *proc* with a key and a value of every entry in the dictionary *dict*, and gathers the result into a list and returns it. For ordered dictionaries, the result is in the increasing order of keys (it doesn't necessarily mean *proc* is called in that order).

dict-keys (*dict* <dictionary>) [Generic function]
dict-values (*dict* <dictionary>) [Generic function]

{gauche.dictionary} Returns a list of all keys or values of a dictionary *dict*, respectively. For ordered dictionaries, the returned list is in the increasing order of keys.

dict->alist (*dict* <dictionary>) [Generic function]

{gauche.dictionary} Returns a list of pairs of key and value in the dictionary. The order of pairs is undefined.

dict-push! (*dict* <dictionary>) *key value* [Generic function]

{gauche.dictionary} A shorthand way to say (dict-put! dict key (cons value (dict-get dict key '()))). A concrete implementation may be more efficient (e.g. it may not search *key* twice.)

dict-pop! (*dict* <dictionary>) *key :optional fallback* [Generic function]

{gauche.dictionary} If (dict-get dict key) is a pair *p*, the entry value is replaced with (cdr p) and the procedure returns (car p). If no entry for *key* is in the table, or the entry isn't a a pair, the table isn't modified, and *fallback* is returned if given, or an error is raised.

dict-update! (*dict* <dictionary>) *key proc :optional fallback* [Generic function]

{gauche.dictionary} Works like the following code, except that the concrete implementation may be more efficient by looking up *key* only once.

```
(rlet1 x (proc (dict-get dict key fallback))
  (dict-put! dict key x))
```

define-dict-interface *dict-class method proc method2 proc2 . . .* [Macro]

{gauche.dictionary} Many dictionary-like datatypes already has their own procedures that directly corresponds to the generic dictionary API, and adding dictionary interface tends to become a simple repetition of define-methods, like this:

```
(define-method dict-put! ((dict <my-dict>) key value)
  (my-dict-put! key value))
```

The define-dict-interface macro is a convenient way to define those methods in a batch. Each *method* argument is a keyword that corresponds to dict-*method*, and *proc* is the name of the datatype-specific procedure. Here's the definition of dict interface for <tree-map> and you'll get the idea. You don't need to provide every dictionary interface.

```
(define-dict-interface <tree-map>
  :get        tree-map-get
  :put!       tree-map-put!
  :delete!    tree-map-delete!
  :clear!     tree-map-clear!
  :comparator tree-map-comparator
  :exists?    tree-map-exists?
  :fold       tree-map-fold
  :fold-right tree-map-fold-right
```

```
:for-each   tree-map-for-each
:map        tree-map-map
:keys       tree-map-keys
:values     tree-map-values
:pop!       tree-map-pop!
:push!      tree-map-push!
:update!    tree-map-update!
:->alist    tree-map->alist)
```

9.9.2 Generic dictionaries

`<bimap>` [Class]

{`gauche.dictionary`} Provides a bidirectional map (*bimap*), a relation between two set of values, of which you can lookup both ways.

Internally, a bimap consists of two dictionaries, *left* map and *right* map. Think a bimap as a relation between *x*s and *y*s. The left map takes an *x* as a key and returns corresponding *y* as its value. The right map takes an *y* as a key and returns corresponding *x* as its value.

Currently, `<bimap>` only supports strict one-to-one mapping. Mutating interface (`bimap-*-put!`, `bimap-*-delete!` etc) modifies both left and right maps to maintain this one-to-one mapping. (In future, we may provide an option to make many-to-one and many-to-many mappings).

A bimap can be used as a dictionary, with the generic dictionary functions such as `dict-get`. In such cases, the left map takes precedence; that is, the key given to `dict-get` etc. is regarded as the key to the left map.

`make-bimap` *left-map right-map :key on-conflict* [Function]

{`gauche.dictionary`} Creates a new bimap consists of two dictionaries, *left-map* and *right-map*. It is the caller's responsibility to choose appropriate type of dictionaries; for example, if you want to create a relation between a string and a number, you man want to create it like this:

```
(make-bimap (make-hash-table 'string=?)  ; string -> number
            (make-hash-table 'eqv?))     ; number -> string
```

The keyword argument *on-conflict* specifies what will happen when the added entry would conflict the existing entries. The following values are allowed:

`:supersede`
: This is the default behavior. Duplicate relations are silently removed in order to maintain one-to-one mapping. For example, suppose a bimap between strings and numbers has had `("foo", 1)` and `("bar", 2)`. When you try to put `("bar", 2)` with this option, the first two entries are removed. Returns `#t`.

`:error`
: Raises an error when duplicate relations are found.

`#f`
: When duplicate relations are found, does nothing and returns `#f`.

Note: At this moment, an attempt to add a relation exactly same as the existing one is regarded as a conflict. This limitation may be lifted in future.

`bimap-left` *bimap* [Function]
`bimap-right` *bimap* [Function]

{`gauche.dictionary`} Returns the left or right map of *bimap*, respectively. Do not mutate the returned map, or you'll break the consistency of the bimap.

`bimap-left-get` *bimap key :optional default* [Function]
`bimap-right-get` *bimap key :optional default* [Function]

{`gauche.dictionary`} Lookup the value corresponding to the *key* in the left or right map of *bimap*. If no entry is found for *key*, *default* is returned if provided, otherwise an error is raised.

`bimap-left-exists?` *bimap key* [Function]
`bimap-right-exists?` *bimap key* [Function]

{`gauche.dictionary`} Returns `#f` if the left or right map of *bimap* has an entry of the key, `#t` otherwise.

`bimap-put!` *bimap x y :key on-conflict* [Function]

{`gauche.dictionary`} Put a relation (*x*, *y*) into the bimap. After this, `(bimap-left-get` *x*`)` will return *y*, and `(bimap-left-get` *y*`)` will return *x*.

If the bimap already have relations with *x* and/or *y*, the conflict is handled according to the value of *on-conflict*; see `make-bimap` for the possible values and their meanings. The *on-conflict* keyword argument can override the bimap's default setting specified at its creation time.

`bimap-left-delete!` *bimap key* [Function]
`bimap-right-delete!` *bimap key* [Function]

{`gauche.dictionary`} Deletes an relation with the given left key or right key from *bimap*. Both left and right maps are modified so that the consistency is maintained. If there's no relations with given key, these are noop.

9.10 `gauche.fcntl` - Low-level file operations

`gauche.fcntl` [Module]

Provides an interface to `fcntl(2)`, including advisory file locking.

`sys-fcntl` *port-or-fd operation :optional arg* [Function]

{`gauche.fcntl`} Performs certain operation on the file specified by *port-or-fd*, which should be a port object or an integer that specifies a system file descriptor. If it is a port, it must be associated to the opened file (i.e. `port-type` returns `file`, see Section 6.22.3 [Common port operations], page 218).

The operation is specified by an integer *operation*. Several variables are defined for valid *operation*.

`F_GETFD` Returns flags associated to the file descriptor of *port-or-fd*. The optional argument *arg* is not used. The return value is an integer whose definition is system specific, except one flag, `FD_CLOEXEC`, which indicates the file descriptor should be closed on `exec`. See the manual entry of `fcntl(2)` of your system for the details.

`F_SETFD` Sets the file descriptor flags given as *arg* to *port-or-fd*. For example, the portable way of setting `FL_CLOEXEC` flag is as follows:

```
(sys-fcntl port F_SETFD
           (logior FD_CLOEXEC
                   (sys-fcntl port F_GETFD)))
```

`F_GETFL` Returns flags associated to the open files specified by *port-or-fd*. The flags includes the following information:

- File access mode. When masked by `O_ACCMODE`, it's either one of `O_RDONLY`, `O_WRONLY` or `O_RDWR`.

- File creation options. `O_CREAT`, `O_EXCL` and/or `O_TRUNC`.
- Whether appending is allowed or not, by `O_APPEND`
- Whether I/O is blocking or non-blocking, by `O_NONBLOCK`.
- Whether it grabs terminal control, by `O_NOCTTY`.

The system may define system-specific flags.

`F_SETFL` Sets flags to the open files specified by *port-or-fd*. Among the flags listed above, only `O_NONBLOCK` and `O_APPEND` can be changed.

Note that `F_GETFD`/`F_SETFD` concern flags associated to the file descriptor itself, while `F_GETFL`/`F_SETFL` concern flags associated to the opened file itself. This makes difference when more than one file descriptor points to the same opened file.

`F_DUPFD` Creates new file descriptor that points to the same file referred by *port-or-fd*. An integer must be provided as *arg*, and that specifies the minimum value of file descriptor to be assigned.

`F_GETLK` The third argument must be provided and be an instance of `<sys-flock>` object described below. It searches the lock information specified by *arg*, and modifies *arg* accordingly.

`F_SETLK`
`F_SETLKW` The third argument must be provided and be an instance of `<sys-flock>` object described below. Sets the advisory file lock according to *arg*. If the lock is successfully obtained, `#t` is returned. If the other process has the lock conflicting the request, `F_SETLK` returns `#f`, while `F_SETLKW` waits until the lock is available.

`F_GETOWN` Returns the process id or process group that will receive SIGIO and SIGURG signals for events on the file descriptor. Process group is indicated by a negative value. This flag is only available on the systems that has this feature (BSD and Linux have this).

`F_SETOWN` Sets the process id or process group that will receive SIGIO and SIGURG signals for events on the file descriptor. Process group is indicated by a negative value. This flag is only available on the systems that has this feature (BSD and Linux have this). Check out fcntl(2) manpage of your system for the details.

Other value for *operation* causes an error.

`<sys-flock>` [Builtin Class]

{`gauche.fcntl`} A structure represents POSIX advisory record locking. Advisory record locking means the system may not prevents the process from operating on files that it doesn't have an appropriate lock. All the processes are expected to use `fcntl` to check locks before it operates on the files that may be shared.

The following slots are defined.

Note that `fcntl` lock is per-process, per-file. If you try to lock the same file more than once within the same process, it always succeeds. But it's not a recursive lock, so the process loses any locks to the file as soon as any of such lock is released, or any of such file is closed. It makes `fcntl` lock difficult to use in libraries. See `with-lock-file` (see Section 12.24.6 [Lock files], page 702) for an alternative way to realize inter-process locks.

`type` [Instance Variable of `<sys-flock>`]

An integer represents lock type. Following variables are predefined for the valid values:

`F_RDLCK` Read locking

F_WRLCK   Write locking

F_UNLCK

To remove a lock by F_SETLK, or to indicate the record is not locked by F_GETLK.

whence [Instance Variable of <sys-flock>]

Indicates from where start is measured.

start [Instance Variable of <sys-flock>]

The offset of beginning of the locked region.

len [Instance Variable of <sys-flock>]

The number of bytes to lock. Zero means "until EOF".

pid [Instance Variable of <sys-flock>]

An integer process id that holding the lock; used only by F_GETLK.

9.11 gauche.generator - Generators

gauche.generator [Module]

A generator is merely a procedure with no arguments and works as a source of a series of values. Every time it is called, it yields a value. The EOF value indicates the generator is exhausted. For example, read-char can be seen as a generator that generates characters from the current input port.

It is common practice to abstract the source of values in such a way, so it is useful to define utility procedures that work on the generators. This module provides them.

Srfi-121 (Generators) is a subset of this module. Since gauche.generator predates srfi-121, we have different names for some procedures; for the compatibility, we provide both names. Srfi-151 (Generators and accumulators) adds some more generator procedures, which is also included (but accumulator procedures are left to srfi-158. See Section 11.37 [Generators and accumulators], page 636.)

A generator is very lightweight, and handy to implement simple on-demand calculations. However, keep in mind that it is side-effecting construct; you can't safely backtrack, for example. For more functional on-demand calculation, you can use lazy sequences (see Section 6.19.2 [Lazy sequences], page 200), which is actually built on top of generators.

The typical pattern of using generator is as follows: First you create a source or sources of the values, using one of generator constructors (see Section 9.11.1 [Generator constructors], page 372) or rolling your own one. You may connect generator operators that modifies the stream of generated items as you wish (see Section 9.11.2 [Generator operations], page 376). Eventually you need to extract actual values from the generator to consume; there are utility procedures provided (see Section 9.11.3 [Generator consumers], page 381). Overall, you create a pipeline (or DAG) of generators that works as lazy value-propagation network.

9.11.1 Generator constructors

A generator isn't a special datatype but just an ordinary procedure, so you can make a generator with lambdas. This module provides some common generator constructors for the convenience.

If you want to use your procedure as a generator, note that a generator can be invoked many times even after it returns EOF once. You have to code it so that once it returns EOF, it keeps returning EOF for the subsequent calls.

The result of generator constructors is merely a procedure, and printing it doesn't show much. In the examples in this section we use generator->list to convert the generator to the list. See Section 9.11.3 [Generator consumers], page 381, for the description of generator->list.

`null-generator` [Function]

{`gauche.generator`} An empty generator. Returns just an EOF object when called.

`circular-generator` *arg . . .* [Function]

[SRFI-158] {`gauche.generator`} Returns an infinite generator that repeats the given arguments.

```
(generator->list (circular-generator 1 2 3) 10)
  ⇒ (1 2 3 1 2 3 1 2 3 1)
```

Note that the above example limits the length of the converted list by 10; otherwise `generator->list` won't return.

`giota` *:optional* (*count +inf.0*) (*start 0*) (*step 1*) [Function]

{`gauche.generator`} Like `iota` (see Section 6.6.3 [List constructors], page 126), creates a generator of a series of *count* numbers, starting from *start* and increased by *step*.

```
(generator->list (giota 10 3 2))
  ⇒ (3 5 7 9 11 13 15 17 19 21)
```

If both *start* and *step* are exact, the generator yields exact numbers; otherwise it yields inexact numbers.

```
(generator->list (giota +inf.0 1/2 1/3) 6)
  ⇒ (1/2 5/6 7/6 3/2 11/6 13/6)
(generator->list (giota +inf.0 1.0 2.0) 5)
  ⇒ (1.0 3.0 5.0 7.0 9.0)
```

`grange` *start :optional* (*end +inf.0*) (*step 1*) [Function]

{`gauche.generator`} Similar to `giota`, creates a generator of a series of numbers. The series begins with *start*, increased by *step*, and continues while the number is below *end*.

```
(generator->list (grange 3 8))
  ⇒ (3 4 5 6 7)
```

`generate` *proc* [Function]

{`gauche.generator`} Creates a generator from coroutine.

The *proc* argument is a procedure that takes one argument, *yield*. When called, `generate` immediately returns a generator *G*. When *G* is called, the *proc* runs until it calls *yield*. Calling *yield* causes to suspend the execution of *proc* and *G* returns the value passed to *yield*.

Once *proc* returns, it is the end of the series—*G* returns eof object from then on. The return value of *proc* is ignored.

The following code creates a generator that produces a series 0, 1, and 2 (effectively the same as `(giota 3)` and binds it to `g`.

```
(define g
  (generate
   (^[yield] (let loop ([i 0])
               (when (< i 3) (yield i) (loop (+ i 1)))))))

(generator->list g) ⇒ (0 1 2)
```

`list->generator` *lis :optional start end* [Function]
`vector->generator` *vec :optional start end* [Function]
`reverse-vector->generator` *vec :optional start end* [Function]
`string->generator` *str :optional start end* [Function]
`uvector->generator` *uvec :optional start end* [Function]

bytevector->generator *u8vector :optional start end* [Function]

[SRFI-158+] {gauche.generator} Returns a generator that yields each item in the given argument. A generator returned from `reverse-*` procedures runs in reverse order. Srfi-121 defines these except `uvector->generator`, which can take any type of uniform vectors. The srfi-121 version, `bytevector->generator`, limits the argument to `u8vector`.

```
(generator->list (list->generator '(1 2 3 4 5)))
  ⇒ (1 2 3 4 5)
(generator->list (vector->generator '#(1 2 3 4 5)))
  ⇒ (1 2 3 4 5)
(generator->list (reverse-vector->generator '#(1 2 3 4 5)))
  ⇒ (5 4 3 2 1)
(generator->list (string->generator "abcde"))
  ⇒ (#\a #\b #\c #\d #\e)
(generator->list (uvector->generator '#u8(1 2 3 4 5)))
  ⇒ (1 2 3 4 5)
```

The generator is exhausted once all items are retrieved; the optional *start* and *end* arguments can limit the range the generator walks across; *start* specifies the left bound and *end* specifies the right bound.

For forward generators, the first value the generator yields is *start*-th element, and it ends right before *end*-th element. For reverse generators, the first value is the item right next to the *end*-th element, and the last value is the *start*-th element. at the last element, and reverse generators ends at the first element.

```
(generator->list (vector->generator '#(a b c d e) 2))
  ⇒ (c d e)
(generator->list (vector->generator '#(a b c d e) 2 4))
  ⇒ (c d)
(generator->list (reverse-vector->generator '#(a b c d e) 2))
  ⇒ (e d c b)
(generator->list (reverse-vector->generator '#(a b c d e) 2 4))
  ⇒ (d c)
(generator->list (reverse-vector->generator '#(a b c d e) #f 2))
  ⇒ (b a)
```

bits->generator *n :optional start end* [Function]
reverse-bits->generator *n :optional start end* [Function]

{gauche.generator} These procedures take an exact integer and treat it as a sequence of boolean values (0 for false and 1 for true), as `bits->list` does (see Section 11.34 [Bitwise operations], page 633). `Bits->generator` takes bits from LSB, while `reverse-bits->generator` takes them from MSB.

```
(generator->list (bits->generator #b10110))
 ⇒ (#f #t #t #f #t)
(generator->list (reverse-bits->generator #b10110))
 ⇒ (#t #f #t #t #f)
```

The optional *start* and/or *end* arguments are used to specify the range of bitfield, LSB being 0. Unlike `list->generator` etc, *start* specifies the rightmost position (inclusive) and *end* specifies the leftmost position (exclusive). It is consistent with other procedures that accesses bit fields in integers (see Section 11.34 [Bitwise operations], page 633).

```
(generator->list (bits->generator #x56 0 4)
  ⇒ (#f #t #t #f)  ; takes bit 0, 1, 2 and 3
(generator->list (bits->generator #x56 4 8)
  ⇒ (#t #f #t #f)  ; takes bit 4, 5, 6 and 7
```

```
(generator->list (reverse-bits->generator #x56 4 8)
  ⇒ (#f #t #f #t)  ; takes bit 7, 6, 5 and 4
```

Note: SRFI-151's `make-bitwise-generator` is similar to `bits->generator`, except that it produces an infinite generator. See Section 11.34 [Bitwise operations], page 633.

`port->sexp-generator` *input-port* [Function]
`port->line-generator` *input-port* [Function]
`port->char-generator` *input-port* [Function]
`port->byte-generator` *input-port* [Function]

{`gauche.generator`} Returns a generator that reads characters or bytes from the given port, respectively. They're just `(cut read input-port)`, `(cut read-line input-port)`, `(cut read-char input-port)` and `(cut read-byte input-port)`, respectively, but we provide them for completeness.

`x->generator` *obj* [Generic function]

{`gauche.generator`} A generic version to convert any collection *obj* to a generator that walks across the *obj*. Besides, if *obj* is an input port, `port->char-generator` is called.

`file->generator` *filename reader . open-args* [Function]

{`gauche.generator`} Opens a file *filename*, and returns a generator that reads items from the file by a procedure *reader*, which takes one argument, an input port. The arguments *open-args* are passed to `open-input-file`

The file is closed when the generator is exhausted. If a generator is abandoned before being read to the end, then the file is kept open until the generator is garbage-collected. If you want to make sure the file is closed by a certain point of time, you might want to use a reader procedure as a generator within the dynamic extent of `with-input-from-file` etc.

`file->sexp-generator` *filename . open-args* [Function]
`file->char-generator` *filename . open-args* [Function]
`file->line-generator` *filename . open-args* [Function]
`file->byte-generator` *filename . open-args* [Function]

{`gauche.generator`} Returns a generator that reads a series of sexps, characters, lines and bytes from a file *filename*, respectively. These are versions of `file->generator` specialized by `read`, `read-char`, `read-line` and `read-byte` as the *reader* argument.

Like `file->generator`, *open-args* are passed to `open-input-file` (see Section 6.22.4 [File ports], page 221). The file is closed when the generator is exhausted.

`gunfold` *p f g seed :optional tail-gen* [Function]

{`gauche.generator`} A generator constructor similar to *unfold* (see Section 10.3.1 [R7RS lists], page 512).

P is a predicate that takes a seed value and determines where to stop. *F* is a procedure that calculates a value from a seed value. *G* is a procedure that calculates the next seed value from the current seed value. *Tail-gen* is a procedure that takes the last seed value and returns a generator that generates the tail.

For each call of the resulting generator, *p* is called with the current seed value. If it returns a true, then we see we've done, and `tail-gen` is called (if it is given) to get a generator for the tail. Otherwise, we apply *f* on the current seed value to get the value to generate, and use *g* to update the seed value.

```
(generator->list (gunfold (^s (> s 5)) (^s (* s 2)) (^s (+ s 1)) 0))
  ⇒ '(0 2 4 6 8 10)
```

SRFI-158 compatible procedures

`generator` *item . . .* [Function]

[SRFI-158] {`gauche.generator`} Returns a generator that generates *item* . . . .

`make-iota-generator` *count :optional start step* [Function]

[SRFI-158] {`gauche.generator`} Same as `giota`, except that the *count* argument is required.

`make-range-generator` *start :optional end stop* [Function]

[SRFI-158] {`gauche.generator`} Same as `grange`.

`make-coroutine-generator` *proc* [Function]

[SRFI-158] {`gauche.generator`} Same as `generate`.

`make-for-each-generator` *for-each obj* [Function]

[SRFI-158] {`gauche.generator`} Given collection *obj* and walker *for-each*, creates a generator that retrieves one item at a time from the collection. Trivially defined as follows:

```
(define (make-for-each-generator for-each coll)
  (generate (^[yield] (for-each yield coll))))
```

If *obj* is mutated before the returned generator walks all the values, the behavior depends on how the *for-each* procedure handles the situation; it may or may not be safe. In general it's better to avoid mutation until the generator returns EOF. Once the generator is exhausted, though, it is safe to mutate `obj`.

`make-unfold-generator` *stop? mapper successor seed* [Function]

[SRFI-158] {`gauche.generator`} This is the same as `gunfold`, except it doesn't take optional *tail-gen* argument.

9.11.2 Generator operations

The following procedures take generators (noted as *gen* and *gen2*) and return a generator. For the convenience, they also accept any collection to *gen* and *gen2* parameters; if a collection is passed where a generator is expected, it is implicitly coerced into a generator.

(NB: This is Gauche's extension. For portable srfi-121/srfi-158 programs, you shouldn't rely on this behavior; instead, explicitly convert collections to generators.)

`gcons*` *item . . . gen* [Function]

[SRFI-158] {`gauche.generator`} Returns a generator that adds *items* in front of *gen*.

```
(generator->list (gcons* 'a 'b (giota 2)))
 ⇒ (a b 0 1)
```

`gappend` *gen . . .* [Function]

[SRFI-158] {`gauche.generator`} Returns a generator that yields the items from the first given generator, and once it is exhausted, use the second generator, and so on.

```
(generator->list (gappend (giota 3) (giota 2)))
 ⇒ (0 1 2 0 1)

(generator->list (gappend))
 ⇒ ()
```

`gconcatenate` *gen* [Function]

{`gauche.generator`} The *gen* argument should generate generators and/or sequences. Returns a generator that yields elements from the first generator/sequence, then the second one, then the third, etc.

It is similar to `(apply gappend (generator->list gen))`, except that `gconcatenate` can work even *gen* generates infinite number of generators.

```
($ generator->list $ gconcatenate
   $ list->generator `(,(giota 3) ,(giota 2)))
 ⇒ (0 1 2 0 1)
```

`gflatten` *gen* [Function]

[SRFI-158] {`gauche.generator`} The argument *gen* is a generator that yields lists. This procedure returns a generator that's yield each element of the lists one at a time.

Example: The game Tetris determines the next dropping piece (tetrimino) by the following algorithm: Take a bag of tetriminos with one for each kind (O, I, T, S, Z, L, J), permute it, and draw one by one; once the bag is empty, take another bag and repeat. The algorithm can be implemented by a pipeline of generates as follows. (Tetris is a registered trademark of The Tetris Company).

```
(use gauche.generator)
(use data.random) ; for permutations-of

(define g
  ($ gflatten $ permutations-of
     $ (circular-generator '(O I T S Z L J))))

(generator->list g 21)
  ⇒
  (L O Z T J S I J L Z T I O S T L Z S I J O)
```

Note the subtle difference of this example and the example in `gconcatenate` above—`gconcatenate` takes a generator of generators, while `gflatten` takes a generator of lists.

If we use Haskell-ish type notation, you can see the subtle differences of those similar procedures:

```
gappend              :: (Generator a, Generator a, ...) -> Generator a
(pa$ apply gappend) :: [(Generator a)] -> Generator a
gconcatenate         :: Generator Generator a -> Generator a
gflatten             :: Generator [a] -> Generator a
```

`gmerge` *less-than gen gen2 . . .* [Function]

[SRFI-158] {`gauche.generator`} Creates a generator that yields elements out of input generators, with the order determined by a procedure `less-than`. The procedure is called as `(less-than a b)` and must return `#t` iff the element `a` must precede the element `b`.

Each input generator must yield an ordered elements by itself; otherwise the result won't be ordered.

If only one generator is given, it is just returned (after coercing the input to a generator). In that case, `less-than` won't be called at all.

```
(generator->list (gmerge < '(1 3 8) '(5) '(2 4)))
  ⇒ '(1 2 3 4 5 8)
```

`gmap` *proc gen gen2 . . .* [Function]

[SRFI-158] {`gauche.generator`} Returns a generator that yields a value returned by *proc* applied on the values from given generators. The returned generator terminates when any of the given generator is exhausted.

NB: This differs from `generator-map` (see Section 6.18.9 [Folding generated values], page 197) which consumes all values at once and returns the results as a list, while `gmap` returns a generator immediately without consuming input.

`gmap-accum` *proc seed gen gen2 . . .* [Function]

{`gauche.generator`} A generator version of `map-accum` (see Section 9.5.1 [Mapping over collection], page 345), mapping with states.

The *proc* argument is a procedure that takes as many arguments as the input generators plus one. It is called as `(proc v v2 ... seed)` where `v`, `v2`, `...` are the values yielded from the input generators, and *seed* is the current seed value. It must return two values, the yielding value and the next seed.

NB: This is called `gcombine` in srfi-121.

`gcombine` *proc seed gen gen2 . . .* [Function]

[SRFI-158] {`gauche.generator`} An alias of `gmap-accum`, provided for the compatibility of srfi-121.

`gfilter` *pred gen* [Function]
`gremove` *pred gen* [Function]

[SRFI-158] {`gauche.generator`} Returns a generator that yields the items from the source generator *gen*, except those who makes *pred* answers false (`gfilter`) or those who makes *pred* answers a true value (`gremove`)

```
(generator->list (gfilter odd? (grange 0)) 6)
 ⇒ (1 3 5 7 9 11)
(generator->list (gremove odd? (grange 0)) 6)
 ⇒ (0 2 4 6 8 10)
```

`gdelete` *item gen :optional =* [Function]

[SRFI-158] {`gauche.generator`} Return a generator that yields the items from the source generator *gen*, except those are the same as *item*. The comparison is done by the procedure passed to =, which defaults to `equal?`.

```
;; Note: This example relies on auto-coercing list to generator.
;; SRFI-121 requires list->generator for the second argument.
(generator->list (gdelete 3 '(1 2 3 4 3 2 1)))
  ⇒  (1 2 4 2 1)
```

`gdelete-neighbor-dups` *gen :optional =* [Function]

[SRFI-158] {`gauche.generator`} Returns a generator that yields the items from the source generator *gen*, but the consecutive items of the same value is omitted. The comparison is done by the procedure passed to =, which defaults to `equal?`.

```
;; Note: This example relies on auto-coercing list to generator.
;; SRFI-121 requires string->generator for the second argument.
(generator->list (gdelete-neighbor-dups "mississippi"))
  ⇒ (#\m #\i #\s #\i #\s #\i #\p #\i)
```

`gfilter-map` *proc gen gen2 . . .* [Function]

[SRFI-158] {`gauche.generator`} Works the same as `(gfilter values (gmap proc gen gen2 ...))`, only slightly efficiently.

`gstate-filter` *proc seed gen* [Function]

[SRFI-158] {`gauche.generator`} This allows stateful filtering of a series. The *proc* argument must be a procedure that takes a value *V* from the source generator and a seed value. It should return two values, a boolean flag and the next seed value. If it returns true for the boolean flag, the generator yields *V*. Otherwise, the generator keeps calling *proc*, with updating the seed value, until it sees the true flag value or the source generator is exhausted.

The following example takes a generator of oscillating values and yields only values that are greater than their previous value.

```
(generator->list
 (gstate-filter (^[v s] (values (< s v) v)) 0
                (list->generator '(1 2 3 2 1 0 1 2 3 2 1 0 1 2 3))))
 ⇒ (1 2 3 1 2 3 1 2 3)
```

`gbuffer-filter` *proc seed gen :optional tail-gen* [Function]

{`gauche.generator`} This procedure allows n-to-m mapping between elements in input and output— that is, you can take a look at several input elements to generate one or more output elements.

The procedure *proc* receives the next input element and accumulated seed value. It returns two values: A list of output values, and the next seed value. If you need to look at more input to generate output, you can return an empty list as the first value.

If the input reaches the end, *tail-gen* is called with the current seed value; it should return a list of last output values. If omitted, the output ends when the output of the last call to *proc* is exhausted (the last seed value is discarded).

Suppose you have a text file. Each line contains a command, but if the line ends with backslash, next line is treated as a continuation of the current line. The following code creates a generator that returns *logical* lines, that is, the lines after such line continuations are taken care of.

```
(gbuffer-filter (^[v s]
                  (if-let1 m (#/\\$/ v)
                    (values '() (cons (m 'before) s))
                    (values `(,(string-concatenate-reverse (cons v s))) '())))
                '()
                (file->line-generator "input-file.txt")
                (^[s] `(,(string-concatenate-reverse s))))
```

`gtake` *gen k :optional padding* [Function]

`gdrop` *gen k* [Function]

[SRFI-158] {`gauche.generator`} Returns a generator that takes or drops initial *k* elements from the source generator *gen*.

Those won't complain if the source generator is exhausted before generating *k* items. By default, the generator returned by `gtake` terminates as the source ends, but if you give the optional *padding* argument, then the returned generator does yield *k* items, using the value given to *padding* to fill the rest.

Note: If you pass *padding*, `gtake` always returns a generator that generates exactly *k* elements even the input generator is already exhausted—there's no general way to know whether you've reached the end of the input. If you need to take *k* items repeatedly from the input generator, you may want to use `gslices` below.

Note for the compatibility: Until 0.9.4, `gtake` takes two optional arguments, *fill?* and *padding*. That is consistent with Gauche's builtin `take*`, but incompatible to srfi-121's `gtake`. We think srfi-121's interface is more compact and intuitive, so we renamed the original one to `gtake*` (emphasizing the similarity to `take*`), and made `gtake` compatible to srfi-121. To ease transition, the current `gtake` allows two optional arguments (four in total), in which case we assume the caller wants to call `gtake*`; so the code that gives two optional arguments to `gtake` would work in both pre-0.9.4 and 0.9.5.

`gtake*` *gen k :optional fill? padding* [Function]

{`gauche.generator`} A variation of `gtake`; instead of single optional *padding* argument, this takes two optional arguments just like `take*` (See Section 6.6.4 [List accessors and modifiers], page 127.) Up to 0.9.4 this version is called `gtake`. This is provided for the backward compatibility.

`gtake-while` *pred gen* [Function]
`gdrop-while` *pred gen* [Function]

[SRFI-158] {`gauche.generator`} The generator version of `take-while` and `drop-while` (see Section 6.6.4 [List accessors and modifiers], page 127). The generator returned from `gtake-while` yields items from the source generator as far as *pred* returns true for each. The generator returned from `gdrop-while` first reads items from the source generator while *pred* returns true for them, then start yielding items.

`gslices` *gen k :optional* (*fill? #f*) (*padding #f*) [Function]

{`gauche.generator`} The generator version of `slices` (see Section 6.6.4 [List accessors and modifiers], page 127). This returns a generator, that yields a list of *k* items from the input generator *gen* at a time.

```
(generator->list (gslices (giota 7) 3))
  ⇒ ((0 1 2) (3 4 5) (6))
```

The *fill?* and *padding* arguments controls how the end of input is handled, just like `gtake`. When *fill?* is `#f` (default), the last item from output generator may not have *k* items if the input is short to fill them, as shown in the above example. If *fill?* is true and the input is short to complete *k* items, *padding* argument is used to fill the rest.

```
(generator->list (gslices (giota 6) 3 #t 'x))
  ⇒ ((0 1 2) (3 4 5))
(generator->list (gslices (giota 7) 3 #t 'x))
  ⇒ ((0 1 2) (3 4 5) (6 x x))
```

`ggroup` *gen k :optional padding* [Function]

[SRFI-158] {`gauche.generator`} Returns a generator lists of *k* elements taken from *gen*. If *padding* is omitted, it works just as `(gslices gen k)`. If *padding* is given, it works just as `(gslices gen k #t padding)`.

This is defined in srfi-158, and more portable than `gslices`.

`grxmatch` *regexp gen* [Function]

{`gauche.generator`} The *gen* argument must be, after coerced, a generator that yields characters.

A generator returned from this procedure tries to match *regexp* from the character sequence generated by *gen*, and once it matches, remember the position after the match and returns `#<rxmatch>` object. If no more match is found, the generator is exhausted.

```
($ generator->list
   $ gmap rxmatch-substring
   $ grxmatch #/\w+/ "The quick brown fox jumps over the lazy dog.")
 ⇒ ("The" "quick" "brown" "fox" "jumps" "over" "the" "lazy" "dog")
```

Note: This procedure is efficient if *gen* is a string, in which case we actually bypass coercing it to a generator. If *gen* is other than a string, the current implementation may need to apply *regexp* as many times as O(n^2) where n is the entire length of the character sequence generated by *gen*, although the coefficient is small. This may be improved in future, but be careful using this function on very large input.

Note also that, when *gen* is not a string, *rxmatch* is applied on some buffered partial input. So `rxmatch-after` of the returned match does not represents the whole "rest of input" after the match, but merely the rest of strings within the buffer.

`gindex` *vgen igen* [Function]

[SRFI-158] {`gauche.generator`} Both arguments are generators. The *igen* generator must yield monotonically increasing series of exact nonnegative integers.

Returns a generator that generates items from *vgen* indexed by the numbers from *igen*, exhausted when either source generator is exhausted.

An error is thrown when *igen* yields a value that doesn't conform the condition.

```
;; This example takes advantage of Gauche's auto-coercing
;; list to generator.  For portable srfi-121 programs,
;; you need list->generator for each argument:
(generator->list (gindex '(a b c d e) '(0 2 3)))
  ⇒ (a c d)
```

`gselect` *vgen bgen* [Function]

[SRFI-158] {`gauche.generator`} Both arguments are generators. Creates and returns a generator that yields a value from *vgen* but only the corresponding value from *bgen* is true.

The returned generator is exhausted when one of the source generators is exhausted.

```
;; This example takes advantage of Gauche's auto-coercing
;; list to generator.  For portable srfi-121 programs,
;; you need list->generator for each argument:
(generator->list (gselect '(a b c d e) '(#t #t #f #t #f)))
  ⇒ (a b d)
```

Combined with a bitgenerator, you can use `gselect` to extract items using bitmask:

```
(generator->list (gselect '(a b c d e)
                          (reverse-bits->generator #x1a)))
  ⇒ (a b d)
```

9.11.3 Generator consumers

Some generator consumers are built-in. See Section 6.18.9 [Folding generated values], page 197, for `generator-fold`, `generator-fold-right`, `generator-for-each`, `generator-map`, and `generator-find`.

`generator->list` *generator :optional k* [Function]
`generator->reverse-list` *generator :optional k* [Function]

[SRFI-158] {`gauche.generator`} Reads items from *generator* and returns a list of them (or a reverse list, in case of `generator->reverse-list`). By default, this reads until the generator is exhausted. If an optional argument *k* is given, it must be a nonnegative integer, and the list ends either *k* items are read, or the generator is exhausted.

Be careful not to pass an infinite generator to this without specifying *k*—this procedure won't return but hogs all the memory before crash.

`generator-map->list` *proc gen gen2 . . .* [Function]

[SRFI-158] {`gauche.generator`} The *proc* argument must be a procedure that takes as many arguments as the number of given generators.

Returns a list, each of whose element is created by applying *proc* on each element from given generators *gen gen2 . . .* . The list ends when any of the generator is exhausted.

Note that the list is created eagerly—if all of the generators are infinite, this procedure never returns.

`generator->vector` *gen :optional k* [Function]
`generator->string` *gen :optional k* [Function]

[SRFI-158] {`gauche.generator`} Extracts items from the generator *gen* up to *k* items or until it exhausts, and create a fresh vector or string from the extracted items.

When *k* is omitted, *gen* is called until it exhausts; note that if *gen* is infinite generator this procedure won't return.

For `generator->string`, *gen* must yield a character, or an error is reported.

`generator->uvector` *gen :optional k class* [Function]
`generator->bytevector` *gen :optional k* [Function]

[SRFI-158] {`gauche.generator`} Extracts items from the generator *gen* up to *k* items or until it exhausts, and create a fresh uniform vector of class *class* filled with those items. If *k* is omitted, *gen* is read until it exhausts.

If *class* is specified, it must be one of the uniform vector classes (see Section 9.36 [Uniform vectors], page 476). When omitted, `<u8vector>` is assumed.

`Generator->bytevector` works like `generator->uvector` except that the class is fixed to `<u8vector>`.

The generator must always produce numeric values acceptable to be an element of the specified uvector; otherwise an error is signalled.

`generator->vector!` *vector at gen* [Function]

[SRFI-158] {`gauche.generator`} Fill *vector* from index *at* with the value yielded from *gen*. It terminates when *gen* is exhausted or the index reaches at the end of the vector. Returns the number of items generated.

```
(define v (vector 'a 'b 'c 'd 'e))

(generator->vector! v 2 (giota))
  ⇒ 3

v ⇒ #(a b 0 1 2)
```

`generator->uvector!` *uvector at gen* [Function]
`generator->bytevector!` *u8vector at gen* [Function]

{`gauche.generator`} Like *generator->vector!*, fill a uniform vector *uvector* starting from index *at* with elements read from a generator *gen*. It terminates when *gen* is exhausted or the index reaches at the end of the vector. Returns the number of items generated.

Any type of uvector can be passed to `generator->uvector!`, while `generator->bytevector!` can only accept `u8vector`.

The generator must always produce numeric values acceptable to be an element of the specified uvector; otherwise an error is signalled.

`glet*` (*binding* . . .) *body body2* . . . [Macro]

{`gauche.generator`} This captures a monadic pattern frequently appears in the generator code. It is in a similar spirit of `and-let*`, but returns as soon as the evaluating expression returns EOF, instead of `#f` as `and-let*` does.

The *binding* part can be either `(var expr)` or `( expr )`. The actual definition will explain this syntax clearly.

```
(define-syntax glet*
  (syntax-rules ()
    [(_ () body body2 ...) (begin body body2 ...)]
    [(_ ([var gen-expr] more-bindings ...) . body)
     (let1 var gen-expr
       (if (eof-object? var)
         var
         (glet* (more-bindings ...) . body)))]
```

```
[(_ ([ gen-expr ] more-bindings ...) . body)
 (let1 var gen-expr
   (if (eof-object? var)
     var
     (glet* (more-bindings ...) . body)))]))
```

`glet1` *var expr body body2 . . .* [Macro]

{`gauche.generator`} This is to `glet*` as `let1` is to `let*`. In other words, it is `(glet* ([var expr]) body body2 ...)`.

`do-generator` (*var gexpr*) *body . . .* [Macro]

{`gauche.generator`} This is a generator version of `dolist` and `dotimes` (see Section 4.6 [Binding constructs], page 52).

Gexpr is an expression that yields a generator. It is evaluated once. The resulting generator is called repeatedly until it returns EOF. Every time the generator is called, *body* . . . are evaluated in the scope where *var* is bound to the value yielded from the generator.

Like `dolist` and `dotimes`, this macro exists for side-effects. You can write the same thing with `for-each` families, but sometimes this macro makes the imperative code more readable:

```
(do-generator [line (file->line-generator "filename")]
  ;; do some side-effecting stuff with line
  )
```

`generator-any` *pred gen* [Function]
`generator-every` *pred gen* [Function]

[SRFI-158] {`gauche.generator`} Like `any` and `every` (see Section 6.6.5 [Walking over lists], page 131), but works on a generator.

`generator-count` *pred gen* [Function]

[SRFI-158] {`gauche.generator`} Returns the number of items in a generator *gen* that satisfies *pred*. As a side effect, *gen* is exhausted.

`generator-unfold` *gen unfold arg . . .* [Function]

[SRFI-158] {`gauche.generator`}

9.12 `gauche.hook` - Hooks

`gauche.hook` [Module]

Provides a hook object, which manages a list of closures to be called at certain time.

This API of hooks are upper-compatible of Guile's, with the following extensions.

- Based on Gauche's object system. Most APIs are methods so you can extend the hook features.
- Hook object itself is applicable. You don't need to use `run-hook`.
- The method to remove a procedure from a hook is called `delete-hook!`, for consistency with SRFI-1 and others. `remove-hook!` is defined as an alias of `delete-hook!` for compatibility with Guile.

If you're writing portable code, srfi-173 provides the basic hook functionality (see Section 11.40 [Hooks (srfi)], page 637).

`<hook>` [Class]

{`gauche.hook`} A hook class, which keeps a list of procedures to be called at once.

The `object-apply` method is defined on `<hook>` class, so you can "apply" a hook object as if it were a procedure—which causes all the registered procedure to be invoked.

`make-hook` *:optional* (*arity 0*) [Function]
{`gauche.hook`} Creates a new hook object with given arity, which should be a non-negative integer.

`hook?` *obj* [Function]
{`gauche.hook`} Returns true if *obj* is a hook object.

`hook-empty?` *hook* [Function]
{`gauche.hook`} Returns true if *hook*'s procedure list is empty.

`add-hook!` (*hook* <*hook*>) *proc :optional* (*append? #f*) [Method]
{`gauche.hook`} Adds a procedure *proc* to *hook*. If *append?* is given and true, *proc* is added at the end of the list. Otherwise, *proc* is added at the front of the list. The *proc* has to be called with the arity given at the `make-hook`.

`delete-hook!` (*hook* <*hook*>) *proc* [Method]
`remove-hook!` (*hook* <*hook*>) *proc* [Method]
{`gauche.hook`} Removes *proc* from the procedure list of *hook*. `Remove-hook!` is an alias of `delete-hook!` just for compatibility with Guile.

`reset-hook!` (*hook* <*hook*>) [Method]
{`gauche.hook`} Empties *hook*'s procedure list.

`hook->list` (*hook* <*hook*>) [Method]
{`gauche.hook`} Returns a copy of *hook*'s procedure list.

`run-hook` (*hook* <*hook*>) *arg* ... [Method]
{`gauche.hook`} Calls *hook*'s procedures in order, with arguments *arg* .... The number of arguments must match the arity given at `make-hook`.

9.13 `gauche.interactive` - Utilities for interactive session

`gauche.interactive` [Module]
Provides useful utilities for the interactive session.

This module is automatically loaded when you run `gosh` interactively.

This module also sets autoloads for functions defined in `gauche.reload` module (see Section 9.27 [Reloading modules], page 439), so that those functions can be used by default in interactive development.

`apropos` *pattern :optional module* [Macro]
{`gauche.interactive`} Show a list of defined variables whose name matches *pattern*. If you give a module or a module name *module*, only the variables defined in that module are listed. Without *module*, the variables "visible" from the current module are listed.

pattern may be a symbol or a regexp object. If it is a symbol, the variables whose name contains the substring that matches the symbol's name are listed. If it is a regexp object, the variables whose name matches the regexp are listed.

Some examples:

```
;; List variables that contains "string" in their name
(apropos 'string)

;; Search in srfi-14 module
(apropos 'char 'srfi-14)
```

`describe` *:optional obj* [Generic Function]
`d` *:optional obj* [Generic Function]

{`gauche.interactive`} Prints the detail information about a Scheme object *obj*. The default method shows *obj*'s class, and if it has any slots, the list of slot names and their values. You can specialize this method for customized display. Some built-in types has specialized methods (see how an integer is described in the example below).

If *obj* is omitted, the last evaluation result bound to `*1` in REPL is used. (see Section 3.2.1 [Working in REPL], page 21)

```
gosh> (sys-stat "Makefile")
#<<sys-stat> 0x1e7de60>
gosh> (d)
#<<sys-stat> 0x1e7de60> is an instance of class <sys-stat>
slots:
  type      : regular
  perm      : 436
  mode      : 33204
  ino       : 3242280
  dev       : 2097
  rdev      : 0
  nlink     : 1
  uid       : 500
  gid       : 500
  size      : 19894
  atime     : 1435379061
  mtime     : 1432954340
  ctime     : 1432954340
gosh> (d 1432954340)
1432954340 is an instance of class <integer>
  (#x556925e4, ~ 1.4Gi, 2015-05-30T02:52:20Z as unix-time)
```

`info` *symbol* [Function]

{`gauche.interactive`} Displays an entry of the named function, syntax, module or class from Gauche's info document. If an environment variable `INFOPATH` is defined, this function searches for the info file from the directories in it. Otherwise, this function guesses info file location from the `gosh`'s library directory. If the info file can't be found, an error is signaled. So this function doesn't work if you haven't installed info file.

If no entry exactly matching with *symbol* is found, the procedure tries to look for similar named entries:

```
gosh> (info 'stirng)
There is no entry for stirng.
Did you mean:
  string>
  string?
  string=
  string<
  string
  :string
```

(If you want to search entries using pattern, see `info-serach` below.)

If the current output port is a tty, the info page is displayed by a paging software. If an environment variable `PAGER` is defined, it is used as a paging software. Otherwise, this function looks for `less` and `more` in this order from the directories in `PATH`. If none of them is found, or the output port is not a tty, this function just displays the page.

The first invocation of this function in a session takes some time to parse the info file.

NB: When you use `less` as a pager, make sure you set it to handle utf-8 characters (e.g. setting `LESSCHARSET` environment variable to `UTF-8`), or you'll see some escaped sequences on the screen.

NB: If you invoke `gosh` within the build tree, using `-ftest` option, `info` reads the info files in the build tree if they exist.

`info-search` *regexp* [Function]

{`gauche.interactive`} Lists info entries matching *regexp*. See `info` above about where the info files are searched.

`ed` *filename-or-procedure :key editor load-after* [Function]

{`gauche.interactive`} Invoke an external editor to open the named file, or the file containing the definition of the given procedure (if it can be known). For the latter, it uses `source-location` procedure to find out the source code location (see Section 6.26.1 [Debugging aid], page 277).

The name of the editor to invoke is determined as follows:

1. The *editor* keyword argument.
2. The value of the variable `*editor*` in the `user` module, if defined. This is handy that you can set this in your `.gaucherc`.
3. The value of the environment variable `GAUCHE_EDITOR`.
4. The value of the environment variable `EDITOR`.

If none of the above is defined or `#f`, the procedure prompts the user to type in the name of the editor.

Once the editor name is obtained, it is invoked as a subprocess, with the following format:

```
EDITOR +lineno filename
```

The *lineno* is an integer line number, 1 being the first line. The editor is expected to locate the cursor on the specified line.

Once the editor process exits, the procedure checks if the named file is updated. If so, it may load the file, according to the value of the *load-after* keyword argument. It may take one of the following values:

`#t` Load the file automatically if it's updated.

`#f` Do not load the file.

`ask` The symbol `ask` cause the procedure to prompt the user whether it should load the file. This is the default.

9.14 `gauche.lazy` - Lazy sequence utilities

`gauche.lazy` [Module]

This module provides utility procedures that yields lazy sequences. For the details of lazy sequences, see Section 6.19.2 [Lazy sequences], page 200.

Since lazy sequences are forced implicitly and indistinguishable from ordinary lists, we don't need a separate set of procedures for *taking* lists and lazy sequences; we can use `find` to search in both ordinary lists and lazy sequences.

However, we do need a separate set of procedures for *returning* either lists or lazy sequences. For example, *lmap* can take any kind of sequences, and returns lazy sequence (and calls the procedure on demand).

This distinction is subtle, so I reiterate it. You can use both `map` and `lmap` on lazy sequences. If you want the result list at once, use `map`; it doesn't have overhead of delayed calculation. If you don't know you'll use the entire result, or you know the result will get very large list and don't want to waste space for an intermediate list, you want to use `lmap`.

`x->lseq` *obj* [Function]

{`gauche.lazy`} A convenience function to coerce *obj* to (possibly lazy) list. If *obj* is a list, it is returned as it is. If *obj* is other type of collection, the return value is a lazy sequence that iterates over the collection. If *obj* is other object, it is returned as it is (you can think of it as a special case of dotted list).

If you try `x->lseq` in REPL, it looks as if it just converts the input collection to a list.

```
(x->lseq '#(a b c)) ⇒ (a b c)
```

But that's because the lazy sequence is forced by the output routine of the REPL.

`lunfold` *p f g seed :optional tail-gen* [Function]

{`gauche.lazy`} A lazy version of `unfold` (see Section 10.3.1 [R7RS lists], page 512). The arguments *p*, *f* and *g* are procedures, each of which take one argument, the current seed value. The predicate *p* determines when to stop, *f* creates each element, and *g* generates the next seed value. The *seed* argument gives the initial seed value. If *tail-gen* is given, it should also be a procedure that takes one argument, the last seed value (that is, the seed value `(p seed)` returned `#f`). It must return a (possibly lazy) list, that forms the tail of the resulting sequence.

```
(lunfold ($ = 10 $) ($ * 2 $) ($ + 1 $) 0 (^_ '(end)))
  ⇒ (0 2 4 6 8 10 12 14 16 18 end)
```

`lmap` *proc seq seq2 . . .* [Function]

{`gauche.lazy`} Returns a lazy sequence consists of values calculated by applying *proc* to every first element of *seq seq2 . . .*, every second element of them, etc., until any of the input is exhausted. Application of *proc* will be delayed as needed.

```
;; If you use map instead of lmap, this won't return
(use math.prime)
(take (lmap (pa$ * 2) *primes*) 10)
  ⇒ (4 6 10 14 22 26 34 38 46 58)
```

`lmap-accum` *proc seed seq seq2 . . .* [Function]

{`gauche.lazy`} The procedure *proc* takes one element each from *seq seq2 . . .*, plus the current seed value. It must return two values, a result value and the next seed value. The result of `lmap-accum` is a lazy sequence consists of the first values returned by each invocation of *proc*.

```
(use math.prime)
(take (lmap-accum (^[p sum] (values sum (+ p sum))) 0 *primes*) 10)
  ⇒ (0 2 5 10 17 28 41 58 77 100)
```

This is a lazy version of `map-accum` (see Section 9.5.1 [Mapping over collection], page 345), but `lmap-accum` does not return the final seed value. We only know the final seed value when we have the result sequence to the end, so it can't be calculated lazily.

`lappend` *seq . . .* [Function]

{`gauche.lazy`} Returns a lazy sequence which is concatenation of *seq . . .*. Unlike `append`, this procedure returns immediately, taking O(1) time. It is useful when you want to append large sequences but may use only a part of the result.

lconcatenate *seqs* [Function]

{gauche.lazy} The *seqs* argument is a sequence of sequences. Returns a lazy sequence that is a concatenation of all the sequences in *seqs*.

This differs from `(apply lappend seqs)`, for `lconcatenate` can handle infinite number of lazy *seqs*.

lappend-map *proc seq1 seq ...* [Function]

{gauche.lazy} Lazy version of `append-map`. This differs from a simple composition of *lappend* and *lmap*, since `(apply lappend (lmap proc seq1 seq ...))` would evaluate the result of `lmap` to the end before passing it to `lappend` (it's because `apply` need to determine the list of arguments before calling `lappend`).

It also differs from `(lconcatenate (lmap proc seq1 seq ...))` in the subtle way.

Remember that Gauche's lazy sequence evaluates one element ahead? `lconcatenate` does that to the result of `lmap`. To see the effect, let's define a procedure with a debug print:

```
(define (p x) #?=(list x x))
```

You can see in the following example that `(apply lappend (lmap ...))` wouldn't delay any of application of `p`:

```
gosh> (car (apply lappend (lmap p '(1 2 3))))
(car (apply lappend (lmap p '(1 2 3))))
#?="(standard input)":4:(list x x)
#?-    (1 1)
#?="(standard input)":4:(list x x)
#?-    (2 2)
#?="(standard input)":4:(list x x)
#?-    (3 3)
1
```

How about `lconcatenate`?

```
gosh> (car (lconcatenate (lmap p '(1 2 3))))
(car (lconcatenate (lmap p '(1 2 3))))
#?="(standard input)":4:(list x x)
#?-    (1 1)
#?="(standard input)":4:(list x x)
#?-    (2 2)
1
```

Oops, even though we need only the first element, and the first result of `lmap`, `(1 1)`, provides the second element, too, `p` is already applied to the second input.

This is because the intermediate lazy list of the result of `lmap` is evaluated "one element ahead". On the other hand, `lappend-map` doesn't have this problem.

```
gosh> (car (lappend-map p '(1 2 3)))
(car (lappend-map p '(1 2 3)))
#?="(standard input)":4:(list x x)
#?-    (1 1)
1
```

linterweave *seq ...* [Function]

{gauche.lazy} Returns a lazy seq of the first items from *seq ...*, then their second items, and so on. If the length of shortest sequence of *seqs* is N, the length of the resulting sequence is `(* N number-of-sequences)`. If all of *seqs* are infinite, the resulting sequence is also infinite.

```
(linterweave (lrange 0) '(a b c d e) (circular-list '*))
 ⇒ (0 a * 1 b * 2 c * 3 d * 4 e *)
```

`lfilter` *proc seq* [Function]

{`gauche.lazy`} Returns a lazy sequence that consists of non-false values calculated by applying *proc* on every elements in *seq*.

`lfilter-map` *proc seq seq2 . . .* [Function]

{`gauche.lazy`} Lazy version of `filter-map`.

`lstate-filter` *proc seed seq* [Function]

{`gauche.lazy`} Lazy sequence version of `gstate-filter` (see Section 9.11.2 [Generator operations], page 376).

`ltake` *seq n :optional fill? padding* [Function]
`ltake-while` *pred seq* [Function]

{`gauche.lazy`} Lazy versions of `take*` and `take-while` (see Section 6.6.4 [List accessors and modifiers], page 127). Note that `ltake` works rather like `take*` than `take`, that is, it won't complain if the input sequence has less than *n* elements. Because of the lazy nature of `ltake`, it can't know whether input is too short or not before returning the sequence.

There are no `ldrop` and `ldrop-while`; you don't need them. if you apply `drop` and `drop-while` on lazy sequence, they return lazy sequence.

`lrxmatch` *rx seq* [Function]

{`gauche.lazy`} This is a lazy sequence version of `grxmatch` (see Section 9.11.2 [Generator operations], page 376).

The *seq* argument must be a sequence of characters (including ordinary strings). The return value is a lazy sequence of `<`*rxmatch*`>` objects, each representing strings matching to the regular expression *rx*.

This procedure is convenient to scan character sequences from lazy character sequences, but it may be slow if you're looking for rarely matching string from very large non-string input. Unless *seq* is a string, `lrxmatch` buffers certain length of input, and if matching phrase isn't found, it extend the buffer and scan again from the beginning, since the match may span from the end of previous chunk to the newly added portion.

`lslices` *seq k :optional fill? padding* [Function]

{`gauche.lazy`} Lazy version of `slices` (see Section 6.6.4 [List accessors and modifiers], page 127).

```
(lslices '(a b c d e f) 2)
  ⇒ ((a b) (c d) (e f))
```

9.15 `gauche.listener` - Listener

`gauche.listener` [Module]

This module provides a convenient way to enable multiple read-eval-print loop (repl) concurrently.

An obvious way to run multiple repls is to use threads; creating as many threads as sessions and calling `read-eval-print-loop` (see Section 6.21 [Eval and repl], page 216) from each thread. Nevertheless, sometimes single threaded implementation is preferred. For instance, you're using a library which is not MT-safe, or your application already uses select/poll-based dispatching mechanism.

To implement repl in the single-threaded selection-base application, usually you register a handler that is called when data is available in the listening port. The handler reads the data and add them into a buffer. Then it examines if the data in the buffer consists a complete expression, and if so, it reads the expression from the buffer, evaluates it, then prints the result to the reporting port. The `<listener>` class in this module provides this handler mechanism, so all you need to do is to register the handler to your dispatching mechanism.

Note: it may also be desirable to buffer the output sometimes, but the current version doesn't implement it.

Listener API

`<listener>` [Class]

{`gauche.listener`} An object that maintains the state of a repl session. It has many external slots to customize its behavior. Those slot values can be set at construction time by using the keyword of the same name as the slot, or can be set by `slot-set!` afterwards. However, most of them should be set before calling `listener-read-handler`.

`input-port` [Instance Variable of `<listener>`]

Specifies the input port from which the listener get the input. The default value is the current input port when the object is constructed.

`output-port` [Instance Variable of `<listener>`]

Specifies the output port to which the listener output will go. The default value is the current output port when the object is constructed.

`error-port` [Instance Variable of `<listener>`]

Specifies the output port to which the listener's error messages will go. The default value is the current error port when the object is constructed.

`reader` [Instance Variable of `<listener>`]

A procedure with no arguments. It should read a Scheme expression from the current input port when called. The default value is system's `read` procedure.

`evaluator` [Instance Variable of `<listener>`]

A procedure that takes two arguments, a Scheme expression and an environment specifier. It should evaluate the expression in the given environment and returns zero or more value(s). The default value is system's `eval` procedure.

`printer` [Instance Variable of `<listener>`]

A procedure that takes zero or more argument(s) and prints them out to the current output port. The default value is a procedure that prints each value by `write`, followed by a newline.

`prompter` [Instance Variable of `<listener>`]

A procedure with no arguments. It should prints a prompt to the current output port. The output is flushed by the listener object so this procedure doesn't need to care about it. The default procedure prints `"listener> "`.

`environment` [Instance Variable of `<listener>`]

An environment specifier where the expressions will be evaluated. The default value is the value returned by `(interaction-environment)`.

`finalizer` [Instance Variable of `<listener>`]

A thunk that will be called when EOF is read from `input-port`. During the execution of *finalizer*, the current input, output and error ports are restored to the ones when `listener-read-handler` is called.

It can be `#f` if no such procedure is needed. The default value is `#f`.

`error-handler` [Instance Variable of `<listener>`]

A procedure that takes one argument, an error exception. It is called when an error occurs during read-eval-print stage, with the same dynamic environment as the error is signaled. The default value is a procedure that simply prints the error exception by `report-error`.

`fatal-handler` [Instance Variable of `<listener>`]

A procedure that takes one argument, an error exception. It is called when a *fatal* error occurred (see below for the precise definition). If this handler is called, you should assume you can no longer continue the listener session safely, even write messages to the client. This handler is to log such condition or to clean up the listener. During the execution of *fatal-handler*, the current input, output and error ports are restored to the ones when `listener-read-handler` is called.

If *fatal-handler* returns `#f`, *finalizer* is called afterwards. With this, you can implement a common cleanup work in *finalizer*. If *fatal-handler* returns a true value, *finalizer* will not be called.

`listener-read-handler` (*listener* <*listener*>) [Method]

{`gauche.listener`} Returns a thunk that is to be called when a data is available from `input-port` of the listener.

The returned thunk (read handler) does the following steps. Note that the first prompt is *not* printed by this procedure. See `listener-show-prompt` below.

1. Reads available data from `input-port` and appends it to the listener's internal buffer.
2. Scans the buffer to see if it has a complete S-expression. If not, returns.
3. Reads the S-expression from the buffer. The read data is removed from the buffer.
4. Evaluates the S-expression, then prints the result to `output-port`.
5. Prints the prompt by prompter procedure to `output-port`, then flush `output-port`.
6. Repeats from 2.

`listener-show-prompt` (*listener* <*listener*>) [Method]

{`gauche.listener`} Shows a prompt to the listener's output port, by using listener's prompter procedure. Usually you want to use this procedure to print the first prompt, for instance, when the client is connected to the listener socket.

`complete-sexp?` *str* [Function]

{`gauche.listener`} Returns `#t` if *str* contains a complete S-expression. This utility procedure is exported as well, since it might be useful for other purposes.

Note that this procedure only checks syntax of the expressions, and doesn't rule out erroneous expressions (such as containing invalid character name, unregistered SRFI-10 tag, etc.). This procedure may raise an error if the input contains '`#<`' character sequence.

Error handling

There are a few error situations the listener handles differently.

- *Fatal error* - An error situation that the listener session can no longer go on safely. You cannot even tell so to the listener client, since the connection to the client may be broken. All you can do is to clean up the listener session (e.g. removes the handler). This case happens in (1) a low-level system error occurs during reading from *input-port*. (A syntax error of the input isn't count as fatal, and handled as REPL error described below.), (2) a `SIGPIPE` signal is raised during writing to *output-port*, or (3) an unhandled error occurred during executing *error-handler*.

  When this situation happens, the *fatal-handler* is called if it is given. If *fatal-handler* returns `#f`, or *fatal-handler* isn't given, *finalizer* is also called.

- *Leaked error* - If an error occurs during executing *fatal-handler* or *finalizer*, we don't have no more safety net. The error is 'leaked' outside the listener handler, and should be handled by the user of `gauche.listener`.

  Generally this situation should be considered as a bug of the program; you should make sure to catch foreseeable errors within *fatal-handler* and *finalizer*.

- *REPL error* - Other errors are handled by *error-handler*.

Listener example

The following code snippet opens a server socket, and opens a Scheme interactive session when a client is connected. (Note: this code is just for demonstration. Do not run this program on the machine accessible from outside network!)

```
(use gauche.net)
(use gauche.selector)
(use gauche.listener)

(define (scheme-server port)
  (let ((selector (make <selector>))
        (server   (make-server-socket 'inet port :reuse-addr? #t))
        (cid      0))

    (define (accept-handler sock flag)
      (let* ((client (socket-accept server))
             (id     cid)
             (input  (socket-input-port client :buffering :none))
             (output (socket-output-port client))
             (finalize (lambda ()
                         (selector-delete! selector input #f #f)
                         (socket-close client)
                         (format #t "client #~a disconnected\n" id)))
             (listener (make <listener>
                         :input-port input
                         :output-port output
                         :error-port output
                         :prompter (lambda () (format #t "client[~a]> " id))
                         :finalizer finalize))
             (handler (listener-read-handler listener))
             )
        (format #t "client #~a from ~a\n" cid (socket-address client))
        (inc! cid)
        (listener-show-prompt listener)
        (selector-add! selector input (lambda _ (handler)) '(r))))

    (selector-add! selector
                   (socket-fd server)
                   accept-handler
                   '(r))
    (format #t "scheme server started on port ~s\n" port)
    (do () (#f) (selector-select selector))))
```

9.16 gauche.logger - User-level logging

gauche.logger [Module]

Provides a simple interface to log the program's activity. The information can be written to the specified file, or to the system logger using `syslog(3)`. When a file is used, syslog-like prefix string is added to each message, which is configurable. It can also takes care of locking of the file (see the description of `lock-policy` below).

<log-drain> [Class]

{`gauche.logger`} Represents the destination of log messages. There's one implicit global `<log-drain>` instance, which is used by default. However, you can create as many instances by `make` method as you want, in case if you want to log to more than one destination.

path [Instance Variable of `<log-drain>`]

Designates destination of log output. It can be one of the following values.

string — Pathname of the log file. The output is written to it.

`current-error`
`#t` — The output goes to the current error port.

`current-output` — The output goes to the current output port.

`syslog` — The output is sent to the system logger.

`ignore` — Make `log-format` does nothing.

`#f` — The output is turned to a string and returned from `log-format`.

By default, this slot is `#f`.

prefix [Instance Variable of `<log-drain>`]

Specifies the prefix string that is attached to the beginning of every message. If the message spans to several lines, the prefix is attached to each line. The value of this slot can also be a procedure that takes `<log-drain>` object and returns a string to be used as the prefix. The procedure is called every time prefix is needed.

When the `path` slot is a symbol `syslog`, the value of this slot is ignored. System logger will attach an appropriate prefix.

When the value of the prefix slot is a string, the following character sequences have special meanings and replaced by `log-format` for appropriate information when written out.

`~T` — Current time, in the format of "Mmm DD hh:mm:ss" where "Mmm" is an abbreviated month, "DD" is the day of month, "hh", "mm" and "ss" are hours (in 24 hour basis), minutes and seconds, respectively. This format is compatible with system logs.

`~Y` — Current 4-digit year.

`~P` — The program name. The default value is the basename of `(car (command-line))` (see Section 6.25.2 [Command-line arguments], page 248), but you can change it by the `program-name` slot described below.

`~$` — The process id of this program.

`~U` — The name of the effective user of the process.

`~H` — The hostname the process is running.

The default value of this slot is `"~T ~P[~$]: "`. For example, if a string "this is a log message.\nline 2\nline 3" is given as the message, it produces something like the following log entry.

```
Sep  1 17:30:23 myprogram[441]: this is a log message
Sep  1 17:30:23 myprogram[441]: line 2
Sep  1 17:30:23 myprogram[441]: line 3
```

`program-name` [Instance Variable of `<log-drain>`]

Specifies the program name written by `~P` directive of the prefix slot.

`lock-policy` [Instance Variable of `<log-drain>`]

Specifies the way the log file should be locked. If the value of this slot is a symbol `fcntl`, the log file is locked using fcntl() (see Section 9.10 [Low-level file operations], page 370). If the value is a symbol `file`, the log file is locked by creating auxiliary lock file, whose name is generated by appending ".lock" after the log file path. The logging process needs a write permission to the log file directory. Note that if the process is killed forcibly during writing the log file, a stale lock file may remain. `Log-format` silently removes the lock file if it is unusually old (currently 10 minutes). If the value is `#f`, no locking is performed.

The default value is `fcntl`, except MacOSX which doesn't support fcntl()-style locking and thus `file` is default.

The locking isn't performed if the destination is not a file.

`syslog-option` [Instance Variable of `<log-drain>`]
`syslog-facility` [Instance Variable of `<log-drain>`]
`syslog-priority` [Instance Variable of `<log-drain>`]

The value of these slots are used when the destination of the drain is the system logger. See Section 9.30 [Syslog], page 448, for the detailed information about these values. The default values of these slots are `LOG_PID`, `LOG_USER` and `LOG_INFO`, respectively.

`log-open` *path :key prefix program-name* [Function]

{`gauche.logger`} Sets the destination of the default log message to the path *path*. It can be a string or a boolean, as described above. You can also set prefix and program name by corresponding keyword arguments. See the `<log-drain>` above for those parameters.

Despite its name, this function doesn't open the specified file immediately. The file is opened and closed every time `log-format` is called.

`log-default-drain` [Parameter]

{`gauche.logger`} When called with no argument, returns the current default log-drain `log-format` uses when the explicit drain is omitted. It may return `#f` if the default log drain hasn't been opened by `log-open`.

Calling with new `<log-drain>` object or `#f` alters the default log-drain. You can also use `parameterize` (Section 9.22 [Parameters], page 411) to change the log drain temporary.

`log-format` (*format* <*string*>) *arg* ... [Method]
`log-format` (*drain* <*log-drain*>) (*format* <*string*>) *arg* ... [Method]

{`gauche.logger`} Formats a log message by *format* and *arg* ..., by using `format` (see Section 6.22.8 [Output], page 231). In the first form, the output goes to the default destination. In the second form, the output goes to the specified drain.

The file is opened and closed every time. You can safely move the log file while your program that touches the log file is running. Also `log-format` acquires a write lock of the log file by `sys-fcntl` (see Section 9.10 [Low-level file operations], page 370).

If the first form of `log-format` is called before `log-open` is called, `log-format` does nothing. It is useful to embed debug stubs in your code; once your code is past the debugging stage, you just comment out `log-open` and the code runs without logging.

9.17 gauche.mop.propagate - Propagating slot access

`gauche.mop.propagate` [Module]

Provides a metaclass to add `:propagated` slot allocation option.

When a slot allocation has `:propagated`, access to the slot is redirected to other object's slot. It is handy for composite objects to keep external interface simple, for access to the slot of inner objects can be disguised as if it is a slot of the parent object.

An example would work better than explanation. Suppose you have a `<rect>` class to represent generic rectangular area, and you want to use it when you create a `<viewport>` class by composition, instead of inheritance. A simple way would be as follows:

```
(define-class <rect> ()
  ((width  :init-keyword :width)
   (height :init-keyword :height)))

(define-class <viewport> ()
  ((dimension :init-form (make <rect>))
   ;;   ... other slots ...
   ))
```

With this definition, whenever you want to access the viewport's width or height, you have to go through `<rect>` object, e.g. `(~ viewport'dimension'width)`. This is not only cumbersome, but the users of viewport class have to know that how the viewport is composed (it's not necessarily a bad thing, but sometimes you may want to hide it).

Using `gauche.mop.propagate`, you can define slots `width` and `height` in `<viewport>` class that are proxies of `<rect>`'s slots.

```
(use gauche.mop.propagate)

(define-class <rect> ()
  ((width  :init-keyword :width)
   (height :init-keyword :height)))

(define-class <viewport> (<propagate-mixin>)
  ((dimension :init-form (make <rect>))
   (width     :allocation :propagated :propagate 'dimension
              :init-keyword :width)
   (height    :allocation :propagated :propagate 'dimension
              :init-keyword :height)))
```

With `:propagated` allocation, the slots are not actually allocated in `<viewport>` instance, and accesses to the slots are redirected to the object in the slot specified by `:propagate` slot option—in this case, the `dimension` slot. It is somewhat similar to the virtual slots, but it's more convenient for you don't explicitly write procedures to redirect the access.

Now you can treat `width` and `height` as if they are slots of `<viewport>`. You can even make them initialize via `init-keyword` (but you can't use `:init-form` or `:init-value`; if you want to specify default values, give the default values to the actual object).

```
gosh> (define vp (make <viewport> :width 640 :height 480))
vp
gosh> (d vp)
#<<viewport> 0xc5a1e0> is an instance of class <viewport>
slots:
  dimension : #<<rect> 0xc5a130>
```

```
  width     : 640
  height    : 480
gosh> (set! (~ vp'width) 800)
#<undef>
gosh> (~ vp'width)
800
```

Here's two classes that enables this feature. Usually all you have to do is to inherit `<propagate-mixin>` class.

`<propagate-meta>` [Class]

{`gauche.mop.propagate`} Adds `:propagated` slot allocation. The propagated slot has to have `:propagate` slot option which specifies the name of the slot that points to an object that actually holds the value of the slot. If a slot has `:propagated` slot allocation but does not have `:propagate` slot option, an error is signaled.

The `:propagate` slot option should have a value of either a symbol, or a list of two symbols.

If it is a symbol, it names the slot that contains an object, whose slot with the same name of the propagate slot holds the value.

If it is a list of two symbols as `(X Y)`, then the access to this propagated slot actually works as `(slot-ref (slot-ref obj X) Y)`.

If you want to make a propagated slot initializable by init-keywords, make sure the slot holding the actual object comes before the propagated slots. Slot initialization proceeds in the order of appearance by default, and you want the actual object is created before setting values.

`<propagate-mixin>` [Class]

{`gauche.mop.propagate`} This is a convenience mixin class. Instead of giving `:metaclass <propagate-meta>`, you can just inherit this calss to make propagated slots available.

9.18 gauche.mop.singleton - Singleton

`gauche.mop.singleton` [Module]

Provides a metaclass to define a singleton class.

`<singleton-meta>` [Class]

{`gauche.mop.singleton`} Creates a singleton class. A singleton class is a class that is guaranteed to create only one instance. The first invocation of `make` creates the single instance, and further attempt of creation returns the same instance.

```
(define-class single () () :metaclass <singleton-meta>)

(define a (make single))
(define b (make single))

(eq? a b) ⇒ #t
```

The slots of the instance are initialized at the first invocation of `make`. Initargs of `make` are effective only in the fist invocation, and ignored in the subsequent invocation.

The call of initialization in `make` is thread-safe.

`instance-of` (*class* <*singleton-meta*>) *:rest initargs* [Method]

{`gauche.mop.singleton`} This method just calls `make` with the passed arguments. It is more obvious in the program that you're dealing with singleton.

`<singleton-mixin>` [Class]

{`gauche.mop.singleton`} An instance of `<singleton-meta>`. Instead of specifying `<singleton-meta>` as the `:metaclass` argument of `define-class`, you can inherit this class to give your class the property of singleton.

9.19 gauche.mop.validator - Slot with validator

`gauche.mop.validator` [Module]

Provides a metaclass that adds `:validator` and `:observer` slot options.

`<validator-meta>` [Class]

{`gauche.mop.validator`} This metaclass adds a feature that you can specify callbacks that are called before and after the slot value is set. For example, if you want to guarantee that a certain slot always holds a string value, you can make a procedure be called before the slot is modified, either by `slot-ref` or by a setter method. In the procedure you can either rejects a value except string, or coerce the value to a string.

A *validator* procedure is a callback procedure that is called before the slot value is set. It can be specified by `:validator` slot option. The procedure takes two values, the instance and the value to be set. Whatever the procedure returns is set to the actual slot value.

A *observer* procedure is a callback procedure that is called after the slot value is set. It can be specified by `:observer` slot option. The procedure also takes two values, the instance and the new value. Result of the observer procedure is discarded.

See the following example:

```
(define-class <v> ()
  ((a :accessor a-of
      :validator (lambda (obj value) (x->string value)))
   (b :accessor b-of
      :validator (lambda (obj value)
                   (if (integer? value)
                       value
                       (error "integer required for slot b")))))
  :metaclass <validator-meta>)

(define v (make <v>))
(slot-set! v 'a 'foo)
(slot-ref v 'a) ⇒ "foo"

(set! (a-of v) 1234)
(a-of v) ⇒ "1234"

(slot-set! v 'b 55)
(slot-ref v 'b) ⇒ 55

(slot-set! v 'b 3.4) ⇒ error
(set! (b-of v) 3.4)  ⇒ error
```

You can specify default slot value (`:init-value` etc.) with `:validator`. In that case, the initialization method of the instance calls the validator with the specified default value, if :init-keyword is not given.

```
(define-class <v> ()
  ((a :initform 'foo :init-keyword :a
      :validator (lambda (obj value) (x->string value)))))
```

```
(slot-ref (make <v>) 'a)         ⇒ "foo"
(slot-ref (make <v> :a 555) 'a) ⇒ "555"
```

It looks similar to the virtual slot, but note that a slot with validator has an actual storage in the instance, while a virtual slot doesn't.

It is also a good example of customizing how the slots are accessed using the metaobject protocol. This feature is implemented by only a couple of dozen lines of code.

9.20 gauche.net - Networking

`gauche.net` [Module]

Provides a set of functions necessary for network communications based on BSD socket interface.

The API is provided in two different levels. Lower level routines reflect traditional BSD socket interface, such as `bind(2)`. Higher level routines provides more convenient way to create typical connection-oriented server/client sockets.

This module also provides APIs to obtain various information about hostnames, service ports, and protocols.

Gauche can handle IPv6 if it is compiled with the `--enable-ipv6` configuration option. To check whether IPv6 is enabled or not, you can use `cond-expand` with `gauche.net.ipv6` feature identifier after loading `gauche.net`, as shown below.

```
(use gauche.net)
(cond-expand
  (gauche.net.ipv6
    ... code to use ipv6 ...)
  (else
    ... ipv4 only code ...))
```

See Section 4.12 [Feature conditional], page 68, for the details of `cond-expand`.

Note: If you want to write a portable program using network, take a look at `srfi-106` (see Section 11.20 [Basic socket interface], page 623).

9.20.1 Socket address

Socket address objects

`<sockaddr>` [Builtin Class]

{`gauche.net`} An abstract base class of socket addresses. Each socket address family is implemented as a subclass of this class.

Although socket addresses are built-in classes, you can use `make` method to create an instance of a specific socket address family.

`sockaddr-family` *addr* [Generic Function]

{`gauche.net`} Returns a symbol that indicates the family of the socket address *addr*.

`sockaddr-name` *addr* [Generic Function]

{`gauche.net`} Returns a string which represents the content of the socket address *addr*.

`<sockaddr-in>` [Builtin Class]

{`gauche.net`} AF_INET family socket address. To create an instance of this class, use `make` method as follows:

```
(make <sockaddr-in> :host host :port port)
```

host can be a string, an integer IP address, a `u8vector` IP address, or one of the keywords `:any`, `:broadcast`, `:none` or `:loopback`. If it is a string, it is either a host name or a dotted IP notation. Gauche uses `gethostbyname(3)` to obtain the actual IP address from *host* parameter. If it is a keyword `:any`, or `:broadcast`, the address uses `INADDR_ANY`, or `INADDR_BROADCAST` respectively. The keyword `:loopback` is a synonym to the IPv4 loopback address "127.0.0.1".

port must be a positive integer indicating the port number. See also `make-sockaddrs` below, to create multiple socket addresses on the machine which may have more than one protocol stack.

`sockaddr-family` (*addr* <*sockaddr-in*>) [Method]
{`gauche.net`} Returns a symbol `inet`.

`sockaddr-name` (*addr* <*sockaddr-in*>) [Method]
{`gauche.net`} Returns a string in the form "*a.b.c.d*:*port*", where "*a.b.c.d*" is dotted decimal notion of the IP address and *port* is the port number.

`sockaddr-addr` (*addr* <*sockaddr-in*>) [Method]
`sockaddr-port` (*addr* <*sockaddr-in*>) [Method]
{`gauche.net`} Returns the IP address and the port number as an integer, respectively.

`<sockaddr-un>` [Builtin Class]
{`gauche.net`} AF_UNIX family socket address. To create an instance of this class, use `make` method as follows:

```
(make <sockaddr-un> :path path)
```

path must be a string specifying pathname of the socket.

`sockaddr-family` (*addr* <*sockaddr-un*>) [Method]
{`gauche.net`} Returns a symbol `unix`.

`sockaddr-name` (*addr* <*sockaddr-un*>) [Method]
{`gauche.net`} Returns a pathname of the socket address.

`<sockaddr-in6>` [Builtin Class]
{`gauche.net`} AF_INET6 family socket address. This is only available if gauche is configured with –enable-ipv6 configure option. The constructor and the slots are the same as `<sockaddr-in>`. See also `make-sockaddrs` below, to create multiple socket addresses on the machine which may have more than one protocol stack.

`make-sockaddrs` *host port :optional proto* [Function]
{`gauche.net`} This is a higher-level utility procedure to create all possible inet domain socket addresses that point to ***host*:*port*** of protocol *proto*. Particularly, if the specified host has both IPv4 and IPv6 addresses, and the running system supports both, then both IPv4 and IPv6 socket addresses are returned. If *host* has multiple IP addresses, socket addresses are created for each of these IP address. You can make your network application much more portable among different network stack configurations.

Passing `#f` to *host* creates the local (server) address. You can also pass a service name (e.g. `"http"`) instead of an integer, to the *port* argument. The value of *proto* can be either a symbol `tcp` or `udp`, and the default is `tcp`.

It always returns a list of socket addresses. If the lookup of *host* is failed, null list is returned.

Address and string conversion

`inet-string->address` *address* [Function]

{`gauche.net`} Converts string representing of the internet address *address* to an integer address. If *address* is parsed successfully, returns two values: the integer address value and the recognized protocol (the constant value `2` (= `AF_INET`) for IPv4 addresses, and `10` (= `AF_INET6`) for IPv6 addresses). If *address* can't be parsed, `#f` and `#f` are returned.

```
(inet-string->address "192.168.1.1")
 ⇒ 3232235777 and 2
(inet-string->address "::1")
 ⇒ 1 and 10
(inet-string->address "::192.168.1.1")
 ⇒ 3232235777 and 10
(inet-string->address "ffe0::1")
 ⇒ 340116213421465348979261631549233168385 and 10
(inet-string->address "::192.168.1.1")
 ⇒ 3232235777 and 10
```

`inet-string->address!` *address buf* [Function]

{`gauche.net`} Like `inet-string->address`, but fills the given u8vector *buf* by the parsed address instead of returning it as an integer value. The integer representation of inet addresses is likely to be a bignum, and you can avoid creating bignums with this function. The given u8vector *buf* must be mutable. Returns the protocol on success, or `#f` on failure.

The caller must provide big enough buffer. If *buf* is larger than required, the result is filled from the top of the u8vector and the rest of the vector remains intact.

```
(let* ((buf (make-u8vector 16 0))
       (proto (inet-string->address! "192.168.1.1" buf)))
  (list proto buf))
 ⇒ (2 #u8(192 168 1 1 0 0 0 0 0 0 0 0 0 0 0 0))
```

`inet-address->string` *address protocol* [Function]

{`gauche.net`} Converts the given *address* to its string representation of the protocol *protocol*, which can be either `2` (the constant `AF_INET`) or `10` (the constant `AF_INET6`). An integer or a u8vector can be used as *address*. If it is a u8vector, only the necessary portion of the vector is read; i.e. the vector can be longer than the required length.

```
(inet-address->string 3232235777 AF_INET)
  ⇒ "192.168.1.1"

(inet-address->string '#u8(192 168 1 1) AF_INET)
  ⇒ "192.168.1.1"

(inet-address->string 3232235777 AF_INET6)
  ⇒ "::c0a8:101"
```

9.20.2 High-level network functions

`<socket>` [Builtin Class]

{`gauche.net`} Abstracts a socket, a communication endpoint.

For a connection-oriented socket, you can access the communication channel by two ports associated to the socket, one for input and another for output. `socket-input-port` and `socket-output-port` returns those ports, respectively.

The `<socket>` class implements `<connection>` interface. See Section 9.8 [Connection framework], page 365, for the details. The `connection-self-address` and `connection-peer-address` methods return a socket address object.

The following three functions are convenient ways to create a connection-oriented socket. Those functions are to provide an easy methods for typical cases, but have less control. If you need more than these functions provide, use low-level interface.

`make-client-socket` *:optional address-spec . . .* [Function]

{`gauche.net`} Creates and returns a client socket, connected to the address specified by *address-spec . . .*.

`(make-client-socket 'unix` *path*`)`
: The client socket is connected to the unix domain server socket of address *path*.

`(make-client-socket 'inet` *host port*`)`
: The client socket is connected to the inet domain server socket with hostname *host* and port *port*. TCP protocol is assumed. *host* can be either a hostname, or a dotted decimal notation of IPv4 address. If gauche is compiled with –enable-ipv6, IPv6 address notation can also be used. *Port* must be an exact integer specifying a port number, or a string service name (e.g. `"http"`).

  If gauche is compiled with –enable-ipv6, and the hostname is given, and the hostname has both IPv6 and IPv4 addresses, then IPv6 connection is tried first, and IPv4 is used when IPv6 fails.

`(make-client-socket` *host port*`)`
: This works the same as above. This form is for compatibility with STk.

`(make-client-socket` *sockaddr*`)`
: If an instance of `<sockaddr>` is passed, a socket suitable for *sockaddr* is opened and then connected to the given address.

This function raises an error if it cannot create a socket, or cannot connect to the specified address.

```
(make-client-socket 'inet "www.w3.com" 80)
  ⇒ ;a socket connected to www.w3.com, port 80
(make-client-socket "127.0.0.1" 23)
  ⇒ ;a socket connected to localhost, port 23
(make-client-socket 'unix "/tmp/.sock"
  ⇒ ;a socket connected to a unix domain socket "/tmp/.sock"
```

`make-server-socket` *:optional address-spec . . .* [Function]

{`gauche.net`} Creates and returns a server socket, listening the address specified by *address-spec*.

`(make-server-socket 'unix` *path* `[:backlog` *num*`])`
: The socket is bound to a unix domain socket with a name *path*. The keyword argument *backlog* is passed to `socket-listen` to specify the maximum number of connection request the server can keep before accepting them. The default is 5. If your server is very busy and you see "connection refused" often, you might want to increase it.

`(make-server-socket 'inet` *port* `[:reuse-addr?` *flag*`] [:sock-init` *proc*`] [:backlog` *num*`])`
: The socket is bound to an inet domain TCP socket, listening port *port*, which must be a non-negative exact integer or a string service name (e.g. `"http"`).

If *port* is zero, the system assigns one of available port numbers. If a keyword argument *reuse-addr?* is given and true, `SO_REUSEADDR` option is set to the socket before bound to the port. This allows the process to bind the server socket immediately after other process releases the port.

Alternatively, you can pass a list of positive exact integers to *port*. In that case, Gauche tries to bind each port in the list until it succeeds.

If keyword argument `sock-init` is given, it should be a procedure that takes two arguments, a created socket and the socket address. The procedure is called just after the socket is created. It is useful to set some special socket options. The keyword argument *backlog* is the same as in unix sockets; see the description above.

`(make-server-socket` *`port`* `[:reuse-addr?` *`flag`*`] [:sock-init` *`proc`*`] [:backlog` *`num`*`])`

This is a synonym to the above form (except *port* must be an integer). This form is backward-compatible with STk's `make-server-socket`.

`(make-server-socket` *`sockaddr`* `[:reuse-addr?` *`flag`*`] [:sock-init` *`proc`*`] [:backlog` *`num`*`])`

This form explicitly specifies the socket address to listen by an instance of `<sockaddr>`.

```
(make-server-socket 'inet 8080)
  ⇒ #<socket (listen "0.0.0.0:8080")>
(make-server-socket 8080)
  ⇒ #<socket (listen "0.0.0.0:8080")>
(make-server-socket 'inet 0)
  ⇒ #<socket (listen "0.0.0.0:35628")>
(make-server-socket 'unix "/tmp/.sock")
  ⇒ #<socket (listen "/tmp/.sock")>
```

`make-server-sockets` *host port :key reuse-addr? sock-init* [Function]

{`gauche.net`} Creates one or more sockets that listen at *port* on all available network interfaces of *host*. You can specify a service name (such as `"http"`) to *port*, as well as an integer port number. Returns a list of opened, bound and listened sockets.

This procedure is particularly useful when the host has multiple protocol stacks, such as IPv4 and IPv6. In that case, this procedure may return a list of IPv4 socket(s) and IPv6 socket(s). (On some OSes, single socket can listen both IPv4 and IPv6. On such platform, a list of single socket will be returned.)

The meaning of keyword arguments are the same as of `make-server-socket`.

You can pass 0 to *port*, just like `make-server-socket`, to let the system choose an available port number. If pass 0 as port and this procedure returns multiple sockets, it is guaranteed that all the sockets share the same port number.

Several accessors are available on the returned socket object.

`socket-address` *socket* [Function]

{`gauche.net`} Returns a socket address associated with *socket*. If no address has been associated to the socket, `#f` is returned.

`socket-input-port` *socket :key* (*buffering :modest*) [Function]
`socket-output-port` *socket :key* (*buffering :line*) [Function]

{`gauche.net`} Returns an input and output port associated with *socket*, respectively.

The keyword argument *buffering* specifies the buffering mode of the port. See Section 6.22.4 [File ports], page 221, for explanation of the buffering mode.

`socket-close` *socket* [Function]

{`gauche.net`} Closes *socket*. All the ports associated to *socket* are closed as well. Note: as of release 0.7.2, this procedure does not shutdown the connection. It is because *socket* may be referenced by forked process(es) and you might want to close it without interfering the existing connection. You can call `socket-shutdown` to shutdown the connection explicitly.

`call-with-client-socket` *socket proc :key input-buffering output-buffering* [Function]

{`gauche.net`} *socket* must be a connected client socket. *proc* is called with two arguments, an input port that reads from the socket and an output port that writes to the socket. The socket is closed after *proc* returns or *proc* raises an error.

The keyword arguments *input-buffering* and *output-buffering* are, if given, passed as the *buffering* keyword arguments of `socket-input-port` and `socket-output-port`, respectively.

This is an example of usage of high-level socket functions, a very simple http client.

```
#!/usr/bin/env gosh
(use gauche.net)

(define (usage)
  (display "Usage: swget url\n" (current-error-port))
  (exit 1))

;; Returns three values: host, port, and path.
(define (parse-url url)
  (rxmatch-let (rxmatch #/^http:\/\/([-A-Za-z\d.]+)(:(\d+))?(\/.*)?/ url)
      (#f host #f port path)
    (values host port path)))

(define (get url)
  (receive (host port path) (parse-url url)
    (call-with-client-socket
        (make-client-socket 'inet host (string->number (or port "80")))
      (lambda (in out)
        (format out "GET ~a HTTP/1.0\r\n" path)
        (format out "host: ~a\r\n\r\n" host)
        (flush out)
        (copy-port in (current-output-port))))))

(define (main args)
  (if (= (length args) 2)
      (get (cadr args))
      (usage))
  0)
```

9.20.3 Low-level socket interface

These functions provide APIs similar to the system calls. Those who are familiar to programming with socket APIs will find these functions useful since you can have more detailed control over the sockets.

`make-socket` *domain type :optional protocol* [Function]

{`gauche.net`} Returns a socket with specified parameters.

`PF_UNIX` [Constant]
`PF_INET` [Constant]
`PF_INET6` [Constant]

{`gauche.net`} These constants are bound to the system's constants `PF_UNIX`, `PF_INET` and `PF_INET6`. You can use those values for *domain* argument of `make-socket`.

(`PF_INET6` is defined only if the underlying operating system supports IPv6.)

`AF_UNIX` [Constant]
`AF_INET` [Constant]
`AF_INET6` [Constant]

{`gauche.net`} These constants are bound to `AF_UNIX`, `AF_INET` and `AF_INET6`.

(`AF_INET6` is defined only if the underlying operating system supports IPv6.)

`SOCK_STREAM` [Constant]
`SOCK_DGRAM` [Constant]
`SOCK_RAW` [Constant]

{`gauche.net`} These constants are bound to `SOCK_STREAM`, `SOCK_DGRAM` and `SOCK_RAW`, and suitable to pass to the *type* argument of `make-socket`.

`socket-fd` *socket* [Function]

{`gauche.net`} Returns an integer system file descriptor of the underlying socket.

`socket-status` *socket* [Function]

{`gauche.net`} Returns a internal status of *socket*, by one of the following symbols.

none	The socket is just created.
bound	The socket is bound to an address by `socket-bind`
listening	The socket is listening a connection by `socket-listen`
connected	The socket is connected by `socket-connect` or `socket-accept`.
shutdown	The socket is shutdown by `socket-shutdown`
closed	The socket is closed by `socket-close`.

`socket-bind` *socket address* [Function]

{`gauche.net`} Binds *socket* to the local network address *address*. It is usually used to associate specific address to the server port. If binding failed, an error is signaled (most likely the address is already in use).

For the inet domain address, you can pass *address* with port=0; the system assigns the port number and sets the actual address to the `address` slot of *socket*.

`socket-listen` *socket backlog* [Function]

{`gauche.net`} Listens *socket*. The socket must be already bound to some address. *backlog* specifies maximum number of connection requests to be queued.

`socket-accept` *socket* [Function]

{`gauche.net`} Accepts a connection request coming to *socket*. Returns a new socket that is connected to the remote entity. The original *socket* keeps waiting for further connections. If there's no connection requests, this call waits for one to come.

You can use `sys-select` to check if there's a pending connection request.

`socket-connect` *socket address* [Function]

{`gauche.net`} Connects *socket* to the remote address *address*. This is the way for a client socket to connect to the remote entity.

`socket-shutdown` *socket how* [Function]

{`gauche.net`} Shuts down connection of *socket*. If *how* is `SHUT_RD` (or 0), the receive channel of *socket* is disallowed. If *how* is `SHUT_WR` (or 1), the send channel of *socket* is disallowed. If *how* is `SHUT_RDWR` (or 2), both receive and send channels are disallowed. It is an error to call this function on a non-connected socket.

If you shut down the send channel of the socket, the remote peer sees EOF from its receive channel. This is useful if the remote peer expects EOF before sending something back to you.

`socket-getsockname` *socket* [Function]

{`gauche.net`} Returns a `<sockaddr>` instance that is the local address of *socket*.

`socket-getpeername` *socket* [Function]

{`gauche.net`} Returns a `<sockaddr>` instance that is the peer address of *socket*.

`socket-send` *socket msg :optional flags* [Function]
`socket-sendto` *socket msg to-address :optional flags.* [Function]

{`gauche.net`} Interfaces to `send(2)` and `sendto(2)`, respectively. Transmits the content of *msg* through *socket*. *msg* can be either a string or a uniform vector; if you send binary packets, uniform vectors are recommended.

Returns the number of octets that are actually sent.

When `socket-send` is used, *socket* must already be connected. On the other hand, `socket-sendto` can be used for non-connected socket, and the destination address is specified by a `<sockaddr>` instance *to-address*.

The optional *flags* can be a bitwise OR of the integer constants `MSG_*`. See the system's manpage of `send(2)` and `sendto(2)` for the details.

`socket-sendmsg` *socket msghdr :optional flags* [Function]

{`gauche.net`} Sends a packet described by *msghdr* through *socket* using `sendmsg(3)`. The *msghdr* argument must be a string or u8vector, and it must be prepared as a binary representation of `struct msghdr`. A reliable way to build a *msghdr* is to use `socket-buildmsg` described below.

The *flags* argument is the same as `socket-send` and `socket-sendto`.

Returns number of octets sent.

This procedure is not yet supported under the Windows native platform. You can use the feature identifier `gauche.os.windows` to check availability of this procedure (see Section 3.5 [Platform-dependent features], page 29).

`socket-buildmsg` *addr iov control flags :optional buf* [Function]

{`gauche.net`} Builds a binary representation of `struct msghdr` which is suitable to be given to `socket-sendmsg`. You have to be familiar with `sendmsg(3)` system call to understand this procedure.

The *addr* argument must be an instance of `<sockaddr>` or `#f`. If it is a sockaddr, the `msg_name` field of the msghdr is filled with the address.

The *iov* argument must be a vector or `#f`. If it is a vector, each element must be either a string or a u8vector. They are used to fill `msg_iov` field of the msghdr. Their contents will be concatenated in the kernel to make a payload.

The *control* argument represents ancillary data, a.k.a. cmsg. It can be `#f` if you don't need ancillary data. Otherwise, it must be a list in the following form:

```
((level type data) ...)
```

Where *level* and *type* are exact integers, and *data* is either a string or a u8vector. The former two are used to fill cmsg's `cmsg_level` and `cmsg_type` fields. The *data* is for cmsg's data (`cmsg_len` is calculated from *data*).

The *flags* argument is used to fill `msg_flags`.

If the *buf* argument is `#f` or omitted, new memories are allocated to construct the msghdr. If a mutable u8vector is given to *buf*, `socket-buildmsg` tries to use it to construct the msghdr as much as possible; it allocates memory only if *buf* is used up.

Returns the constructed msghdr as a u8vector.

This procedure is not yet supported under the Windows native platform. You can use the feature identifier `gauche.os.windows` to check availability of this procedure (see Section 3.5 [Platform-dependent features], page 29).

`socket-recv!` *socket buf :optional flags* [Function]

{`gauche.net`} Interface to `recv(2)`. Receives a message from *socket*, and stores it into *buf*, which must be a mutable uniform vector. Returns the number of bytes actually written. *socket* must be already connected. If the size of *buf* isn't enough to store the entire message, the rest may be discarded depending on the type of *socket*.

The optional *flags* can be a bitwise OR of the integer constants `MSG_*`. See the system's manpage of `recv(2)` for the details.

`socket-recvfrom!` *socket buf addrs :optional flags* [Function]

{`gauche.net`} Interface to `recvfrom(2)`. Receives a message from *socket*, which may be unconnected, and stores it to a mutable uniform vector *buf*. Like *socket-recv*, if the size of *buf* isn't enough to store the entire message, the rest may be discarded depending on the type of *socket*.

Returns two values; the number of bytes actually written into *buf*, and an instance of a subclass of `<sys-sockaddr>` which shows the sender's address.

The *addrs* argument must be a list of instances of socket addresses, optionally its last cdr being `#t` (as a special case, if there's zero addresses to pass, just `#t` may be given). The content of each address doesn't matter; if the protocol family of one of them matches the sender's address, the sender's address is written into the passed sockaddr object. By listing sockaddrs of possible families, you can count on `socket-recvfrom!` to allocate no memory on successful operation. It is useful if you call `socket-recvfrom!` in a speed-sensitive inner loop.

If the sender's address family doesn't match any of the addresses given to *addrs*, the behavior depends on whether the list is terminated by `()` or `#t`. If it is terminated by `()`, (i.e. *addrs* is a proper list), the sender's address is simply discarded and `socket-recvfrom!` returns `#f` as the second value. If the list is terminated by `#t`, `socket-recvfrom!` allocates a fresh sockaddr object and returns it as the second value.

Two simple cases: If you pass `()` to *addrs*, the sender's address is always discarded, which is useful if *socket* is connected (that is, you already know your sender's address). If you pass `#t` to *addrs*, a new socket address object is always allocated for the sender's address, which is convenient if you don't mind memory allocation.

The optional *flags* can be a bitwise OR of the integer constants `MSG_*`. See the system's manpage of `recvfrom(2)` for the details.

`socket-recv` *socket bytes :optional flags* [Function]
`socket-recvfrom` *socket bytes :optional flags* [Function]

{`gauche.net`} Like `socket-recv!` and `socket-recvfrom!`, but these returns the received message as a (possibly incomplete) string, up to *bytes* size. Additionally, `socket-recvfrom` always allocates a socket address object for the sender's address.

The use of these procedures are discouraged, since they often returns incomplete strings for binary messages. Using strings for binary data creates many pitfalls. Uniform vectors (especially `u8vectors`) are for binary data. (The reason these procedures return strings is merely historical.)

`MSG_CTRUNC` [Variable]
`MSG_DONTROUTE` [Variable]
`MSG_EOR` [Variable]
`MSG_OOB` [Variable]
`MSG_PEEK` [Variable]
`MSG_TRUNC` [Variable]
`MSG_WAITALL` [Variable]

{`gauche.net`} Pre-defined integer constants to be used as *flags* values for `socket-send`, `socket-sendto`, `socket-recv` and `socket-recvfrom`. Some of these constants may not be defined if the underlying operating system doesn't provide them.

Further control over sockets and protocol layers is possible by getsockopt/setsockopt interface, as described below.

`socket-setsockopt` *socket level option value* [Function]
`socket-getsockopt` *socket level option rsize* [Function]

{`gauche.net`} These are the interface to setsockopt() and getsockopt() calls. The interface is a bit clumsy, in order to allow full access to those low-level calls.

socket must be a non-closed socket object. *level* and *option* is an exact integer to specify the level of protocol stack and the option you want to deal with. There are several variables pre-bound to system constants listed below.

To set the socket option, you can pass either an exact integer or a string to *value*. If it is an integer, the value is passed to setsockopt(2) as C `int` value. If it is a string, the byte sequence is passed as is. The required type of value depends on the option, and Gauche can't know if the value you passed is expected by setsockopt(2); it is your responsibility to pass the correct values.

To get the socket option, you need to tell the maximum length of expected result by *rsize* parameter, for Gauche doesn't know the amount of data each option returns. `socket-getsockopt` returns the option value as a byte string. If you know the option value is an integer, you can pass 0 to *rsize*; in that case `socket-getsockopt` returns the value as an exact integer.

Note about the name: I tempted to name these function socket-{set|get}opt or socket-{set|get}-option, but I rather took the naming consistency. Hence duplicated "sock"s.

The following predefined variables are provided. Note that some of them are not available on all platforms. See manpages `socket(7)`, `tcp(7)` or `ip(7)` of your system to find out exact specification of those values.

For "level" argument:

`SOL_SOCKET` [Variable]
`SOL_TCP` [Variable]
`SOL_IP` [Variable]

{`gauche.net`} These variables are bound to `SOL_SOCKET`, `SOL_TCP` and `SOL_IP`, respectively.

For "option" argument:

`SO_KEEPALIVE` [Variable]

{`gauche.net`} Expects integer value. If it is not zero, enables sending of keep-alive messages on connection-oriented sockets.

SO_OOBINLINE [Variable]

{gauche.net} Expects integer value. If it is not zero, out-of-band data is directly placed into the receive data stream. Otherwise out-of-band data is only passed when the MSG_OOB flag is set during receiving.

SO_REUSEADDR [Variable]

{gauche.net} Expects integer value. If it is not zero, socket-bind allows to reuse local addresses, unless an active listening socket bound to the address.

SO_TYPE [Variable]

{gauche.net} Gets the socket type as an integer (like sock_stream). Can be only used with socket-getsockopt.

SO_BROADCAST [Variable]

{gauche.net} Expects integer value. If it is not zero, datagram sockets are allowed to send/receive broadcast packets.

SO_PRIORITY [Variable]

{gauche.net} Expects integer value, specifying the protocol-defined priority for all packets to be sent on this socket.

SO_ERROR [Variable]

{gauche.net} Gets and clears the pending socket error as an integer. Can be only used with socket-getsockopt.

inet-checksum *packet size* [Function]

{gauche.net} Calculates one's complement of Internet Checksum (RFC1071) of the *packet*, which must be given as a uniform vector. First *size* bytes of *packet* are used for calculation. Returned value is in network byte order (big-endian). It is an error if *size* is greater than the size of *packet*.

Note: The used algorithm assumes *packet* is not too big (< 64K).

9.20.4 Netdb interface

<sys-hostent> [Builtin Class]

{gauche.net} A class of objects for network hosts. Corresponding to struct hostent in C. The following slots are available read-only.

name [Instance Variable of <sys-hostent>]

The formal name of the host (string).

aliases [Instance Variable of <sys-hostent>]

A list of alias names of the host (list of strings).

addresses [Instance Variable of <sys-hostent>]

A list of addresses (list of strings). Only ipv4 address is supported currently. Each address is represented by dotted decimal notation.

sys-gethostbyname *name* [Function]

Looks up a host named *name*. If found, returns a <sys-hostent> object. Otherwise, returns #f.

```
(let ((host (sys-gethostbyname "www.w3c.org")))
  (list (slot-ref host 'name)
        (slot-ref host 'aliases)
        (slot-ref host 'addresses)))
  ⇒ ("www.w3.org" ("www.w3c.org") ("18.29.1.34" "18.29.1.35"))
```

sys-gethostbyaddr *addr proto* [Function]

{gauche.net} Looks up a host that has an address *addr* of protocol *proto*. *addr* is a natural string representation of the address; for ipv4, it is a dotted decimal notation. *proto* is a protocol number; only `AF_INET` is supported currently. If the host is found, returns a `<sys-hostent>` object. Otherwise, returns `#f`.

```
(let ((host (sys-gethostbyaddr "127.0.0.1" AF_INET))
  (list (slot-ref host 'name)
        (slot-ref host 'aliases)
        (slot-ref host 'addresses))
  ⇒ ("localhost" ("localhost.localdomain") ("127.0.0.1"))
```

<sys-servent> [Builtin Class]

{gauche.net} An entry of the network service database. Corresponding to `struct servent` in C. The following slots are available read-only.

name [Instance Variable of `<sys-servent>`]

The formal name of the service (string).

aliases [Instance Variable of `<sys-servent>`]

A list of alias names of the service (list of strings).

port [Instance Variable of `<sys-servent>`]

A port number registered for this service (exact integer).

proto [Instance Variable of `<sys-servent>`]

A protocol name for this service (string).

sys-getservbyname *name proto* [Function]

{gauche.net} Looks up the network service database with a service name *name* and a protocol *proto*. Both *name* and *proto* must be a string. If a service is found, an instance of `<sys-servent>` is returned. Otherwise, `#f` is returned.

```
(let ((serv (sys-getservbyname "http" "tcp")))
  (list (slot-ref serv 'name)
        (slot-ref serv 'aliases)
        (slot-ref serv 'port)
        (slot-ref serv 'proto)))
  ⇒ ("http" () 80 "tcp")
```

sys-getservbyport *port proto* [Function]

{gauche.net} Looks up the network service database with a service port *port* and a protocol *proto*. *port* must be an exact integer, and *proto* must be a string. If a service is found, an instance of `<sys-servent>` is returned. Otherwise, `#f` is returned.

```
(let ((serv (sys-getservbyport 6000 "tcp")))
  (list (slot-ref serv 'name)
        (slot-ref serv 'aliases)
        (slot-ref serv 'port)
        (slot-ref serv 'proto)))
  ⇒ ("x-server" () 6000 "tcp")
```

<sys-protoent> [Builtin Class]

{gauche.net} An entry of the protocol database. Corresponds to `struct protoent` in C. The following slots are available read-only.

name [Instance Variable of `<sys-servent>`]

The formal name of the protocol (string).

`aliases` [Instance Variable of `<sys-servent>`]
A list of alias names of the protocol (list of strings).

`proto` [Instance Variable of `<sys-servent>`]
A protocol number (exact integer).

`sys-getprotobyname` *name* [Function]
{`gauche.net`} Looks up the network protocol database with a name *name*, which must be a string. If a protocol is found, an instance of `<sys-protoent>` is returned. Otherwise, `#f` is returned.

```
(let ((proto (sys-getprotobyname "icmp")))
  (list (slot-ref proto 'name)
        (slot-ref proto 'aliases)
        (slot-ref proto 'proto)))
  ⇒ ("icmp" ("ICMP") 1)
```

`sys-getprotobynumber` *number* [Function]
{`gauche.net`} Looks up the network protocol database with a protocol number *number*, which must be an exact integer. If a protocol is found, an instance of `<sys-protoent>` is returned. Otherwise, `#f` is returned.

```
(let ((proto (sys-getprotobynumber 17)))
  (list (slot-ref proto 'name)
        (slot-ref proto 'aliases)
        (slot-ref proto 'proto)))
  ⇒ ("udp" ("UDP") 17)
```

`<sys-addrinfo>` [Builtin Class]
{`gauche.net`} The new interface to keep address information. Corresponds to `struct addrinfo` in C. This is only available if gauche is configured with –enable-ipv6 option. The following slots are provided.

`flags` [Instance Variable of `<sys-addrinfo>`]

`family` [Instance Variable of `<sys-addrinfo>`]

`socktype` [Instance Variable of `<sys-addrinfo>`]

`protocol` [Instance Variable of `<sys-addrinfo>`]

`addrlen` [Instance Variable of `<sys-addrinfo>`]

`addr` [Instance Variable of `<sys-addrinfo>`]

`sys-getaddrinfo` *nodename servname hints* [Function]
{`gauche.net`} Returns a list of `<sys-addrinfo>` instances from the given *nodename*, *servname* and *hints*. This is only available if gauche is compiled with –enable-ipv6 option.

`sys-ntohs` *integer* [Function]
`sys-ntohl` *integer* [Function]
`sys-htons` *integer* [Function]
`sys-htonl` *integer* [Function]
{`gauche.net`} Utility functions to convert 16bit (`s`) or 32bit (`l`) integers between *network* byte order (`n`) and *host* byte order (`h`).

Scheme API to the netdb interface calls those byte order conversion functions internally, so you don't usually need them so much as in C programs. However, it may be useful when you're constructing or analyzing binary packets. See also Section 12.2 [Packing binary data], page 641, to handle binary data.

9.21 gauche.package - Package metainformation

`gauche.package` [Module]

Gauche manages extra libraries and extension modules as *packages*.

Each package source tree has `package.scm` on top directory, which contains `define-gauche-package` form that provides metainformation about the package—the package name, version, author, dependencies, etc.

When the package is installed, the standard installation process copies that information, with additional information such as the version of Gauche used to build the package, into `.packages` subdirectory of the library installation path, with the name `PACKAGENAME.gpd`, where `PACKAGENAME` is the name of the package.

We collectively call `package.scm` and `*.gpd` as *package description file*.

This module provides utility procedures to read and write package description files, and search installed `*.gpd` files.

define-gauche-package form

configure script and *.gpd file generation

Utility procedures

`<gauche-package-description>` [Function]

{gauche.package}

`path->gauche-package-description` *filename* [Function]

{gauche.package}

`write-gauche-package-description` *description :optional oport* [Function]

{gauche.package}

`make-gauche-package-description` [Function]

{gauche.package}

`gauche-package-description-paths` *:key all-versions* [Function]

{gauche.package}

`find-gauche-package-description` *name :key all-versions* [Function]

{gauche.package}

9.22 gauche.parameter - Parameters

`gauche.parameter` [Module]

A *parameter* is something like a stateful procedure that takes zero or one argument. If no argument is given, the parameter returns the current value it is keeping. If single argument is given, it will be the current value of the parameter. A parameter has several advantages over global variables to store states.

Value of parameters are kept for each thread, so you can use a parameter as a thread-local storage. When a new thread is created, it inherits a copy of parameters from its creator thread.

You can give a "filter procedure" that checks the new value before setting it as the parameter value, so that you can guarantee the value of the parameter is always sane.

With the macro `parameterize`, you can change the parameter's value within certain dynamic scope. It can effectively replace dynamically scoped variables.

You can also register callback procedures to be called whenever the value of the parameter is changed. It can effectively replace so-called "traced variables"

The basic parameter object feature is defined in SRFI-39. It also can be found in other Scheme implementations, including ChezScheme, Chicken and MzScheme. Gauche's one is upper compatible to SRFI-39.

`<parameter>` [Class]

{`gauche.parameter`} A parameter class. A `object-apply` method is defined so that a parameter can be used as if it were a procedure.

```
;; p is a parameter with initial value 2
(define p (make-parameter 2))

;; calling p with no arguments returns the current value
(p) ⇒ 2

;; modify p's value to 3
(p 3)

(p) ⇒ 3

;; you can also use generalized set (srfi-17)
(set! (p) 5)

(p) ⇒ 5

;; using parameterize dynamically changes p's value.

(define (get-p) (p))

(parameterize ([p 7])
  (get-p)) ⇒ 7

(get-p)    ⇒ 5
```

`make-parameter` *value :optional filter* [Function]

[SRFI-39] {`gauche.parameter`} Creates a parameter whose initial value is *value*. If an optional argument *filter* is given, it must be a procedure that takes one argument and returns one value; whenever the parameter's value is about to change, the procedure is called with the given value, and the value the procedure returns will be the parameter's value. The filter procedure can raise an error or reject to change the parameter's value.

`parameterize` ((*param value*) . . .) *body* . . . [Macro]

[SRFI-39] {`gauche.parameter`} Evaluates *body* . . ., with change parameter *param*'s value to the given *value* within the dynamic scope of *body* . . . . Returns the value(s) of the result of the last *body*.

Some examples:

```
(define a (make-parameter 1))
(a) ⇒ 1
(a 2) ⇒ 1
(a) ⇒ 2
(parameterize ((a 3))
```

```
    (a)) ⇒ 3
  (a) ⇒ 2
```

parameter-observer-add! (*p* <*parameter*>) *proc :optional when where* [Method]

{gauche.parameter} Adds *proc* to "observer" procedures of a parameter *p*. Observer procedures are called either (1) just before a new value is set to the parameter, or (2) just after the new value is set to the parameter. In case of (1), a filter procedure is already applied before a callback is called. In either case, observer procedures are called with two arguments, the old value and the new value. The return value(s) of observer procedures are discarded.

The optional *when* argument must be either a symbol `before` or `after`, to specify whether *proc* should be called before or after the value is changed. If omitted, `after` is assumed.

The optional *where* argument must be either a symbol `append` or `prepend`, to specify whether *proc* should be prepended or appended to the existing observer procedure list. If omitted, `append` is assumed.

Note: Although the parameter value itself is thread-local, the observer list is shared by all threads.

parameter-observer-delete! (*p* <*parameter*>) *proc :optional when* [Method]

{gauche.parameter} Deletes *proc* from observer procedure list of a parameter *p*. If *proc* is not in the list, nothing happens. You can give either a symbol `before` or `after` to *when* argument to specify from which list *proc* should be deleted. If *when* argument is omitted, *proc* is deleted from both lists.

parameter-pre-observers (*p* <*parameter*>) [Method]
parameter-post-observers (*p* <*parameter*>) [Method]

{gauche.parameter} Returns a hook object (see Section 9.12 [Hooks], page 383) that keeps "before" or "after" observers, respectively.

Note: Although the parameter value itself is thread-local, these hook objects are shared by all threads.

9.23 gauche.parseopt - Parsing command-line options

gauche.parseopt [Module]

This module defines a convenient way to parse command-line options. The interface is hinted by Perl, and conveniently handles long-format options with multiple option arguments.

Actually, you have a few choices to parse command-line options in Gauche. SRFI-37 (see Section 11.10 [A program argument processor], page 607) provides functional interface to parse POSIX/GNU compatible argument syntax. SLIB has `getopt`-compatible utility. Required features may differ from application to application, so choose whichever fits your requirement.

High-level API

let-args *args* (*bind-spec* . . . [. *rest*]) *body* . . . [Macro]

{gauche.parseopt} This macro captures the most common pattern of argument processing. It takes a list of arguments, *args*, and scans it to find Unix-style command-line options and binds their values to local variables according to *bind-spec*, then executes *body* . . . .

Let's look at a simple example first, which gives you a good idea of what this form does. (See the "Examples" section below for more examples).

```
(define (main args)
  (let-args (cdr args)
      ((verbose     "v|verbose")
```

```
         (outfile     "o|outfile=s")
         (debug-level "d|debug-level=i" 0)
         (help        "h|help" => (cut show-help (car args)))
         . restargs
        )
      ....))

(define (show-help progname)
  ...)
```

The local variable *verbose* will be bound to `#t` if a command-line argument `-v` or `--verbose` is given, and to `#f` otherwise. The variable *output* is specified to take one option argument; if the command-line arguments are given like `-o out.txt`, *outfile* receives `"out.txt"`. The *debug-level* one is similar, but the option argument is coerced to an integer, and also it has default value 0 when the option isn't given. The *help* clause invokes an action rather than merely binding the value.

(Note: Currently `let-args` does not distinguish so-called short and long options, e.g. `-v` and `--v` have the same effect, so as `-verbose` and `--verbose`. In future we may add an option to make it compatible with `getopt_long(3)`.)

The final *restargs* variable after the dot receives a list of non-optional command-line arguments.

Let's look at *bind-spec* in detail. It must be one of the following forms.

```
1. (var option-spec)
2. (var option-spec default)
3. (var option-spec => callback)
4. (var option-spec default => callback)

5. (else => handler)
6. (else formals body ...)
```

A list of command-line arguments passed to *args* are parsed according to *option-spec*s. If the corresponding option is given, a variable *var* is bound to a value as follows:

```
(a) If the bind-spec is 1. or 2., then
  (a1) If option-spec doesn't require an argument, then #t:
  (a2) If option-spec requires one argument, then the value of
       the argument:
  (a3) If option-spec requires more than one argument,
       the list of the values of the arguments.
(b) If the bind-spec is 3. or 4., then callback is called with
  the value(s) of arguments, and its return value.
```

We'll explain the details of *option-spec* later.

As a special case, *var* can be `#f`, in which case the value is ignored. It is only useful for side effects in *callback*.

If the corresponding option is not given in *args*, *var* is bound to *default* if it is given, or `#f` otherwise.

The last *bind-spec* may be the form 5 or 6. in which case the clause is selected when no other *option-spec* matches a given command-line option. In the form 5, *handler* will be called with three arguments; the given option, a list of remaining command-line arguments, and a continuation procedure. The *handler* is supposed to handle the given option, and it may call the continuation procedure with the remaining arguments to continue processing, or it may return a list of arguments which will be treated as non-optional command-line arguments. The form 6 is a shorthand notion of `(else => (lambda formals body ...))`.

The *bind-spec* list can be an improper list, whose last cdr is a symbol. In which case, a list of the rest of the command-line arguments is bound to the variable named by the symbol.

Note that the *default*, *callback*, and forms in `else` clause is evaluated outside of the scope of binding of *vars* (as the name `let-args` implies).

Unlike typical `getopt` or `getopt_long` implementation in C, `let-args` does not permute the given command-line arguments. It stops parsing when it encounters a non-option argument (argument without starting with a minus sign).

If the parser encounters an argument with only two minus signs '`--`', it stops argument parsing and returns a list of arguments after '`--`'.

After all the bindings is done, *body* ... are evaluated. *Body* may began with internal define forms.

Option spec

option-spec is a string that specifies the name of the option and how the option takes the arguments. An alphanumeric characters, underscore, plus and minus sign is allowed for option's names, except that minus sign can't be the first character, i.e. the valid option name matches a regexp `#/[\w+][-\w+]*/`.

If the option takes argument(s), it can be specified by attaching equal character and a character (or characters) that represents the type of the argument(s) after the name. The option can take more than one arguments. The following characters are recognized as a type specifier of the option's argument.

`s` String.

`n` Number.

`f` Real number (coerced to flonum).

`i` Exact integer.

`e` S-expression.

`y` Symbol (argument is converted by `string->symbol`).

Let's see some examples of *option-spec*:

`"name"` Specifies option *name*, that doesn't take any argument.

`"name=s"` Option *name* takes one argument, and it is passed as a string.

`"name=i"` Option *name* takes one argument, and it is passed as an exact integer.

`"name=ss"`
Option *name* takes two arguments, both string.

`"name=iii"`
Option *name* takes three integer arguments.

`"name=sf"`
Option *name* takes two arguments, the first is a string and the second is a number.

If the option has alternative names, they can be concatenated by `"|"`. For example, an option spec `"h|help"` will match both `"h"` and `"help"`.

In the command line, the option may appear with preceding single or double minus signs. The option's argument may be combined by the option itself with an equal sign. For example, all the following command line arguments match an option spec `"prefix=s"`.

```
-prefix /home/shiro
-prefix=/home/shiro
--prefix /home/shiro
--prefix=/home/shiro
```

Error handling

`<parseopt-error>` [Condition Type]

{gauche.parseopt} When `let-args` encounters an argument that cannot be processed as specified by option specs, an error of condition type `<parseopt-error>` is raised. The cases include when a mandatory option argument is missing, or when an option argument has a wrong type.

```
(let-args '("-a" "foo") ((a "a=i")) ; option a requires integer
  (list a))
 ⇒ parseopt-error
```

Note that this condition is about parsing the given *args*. If an invalid *option-spec* is given, an ordinary error is thrown.

Examples

This example is taken from `gauche-install` script. The *mode* option takes numbers in octal, so it uses the callback procedure to convert it. See also the `else` clause how to handle unrecognized option.

```
(let-args (cdr args)
    ((#f      "c")        ;; ignore for historical reason
     (mkdir   "d|directory")
     (mode    "m|mode=s" #o755 => (cut string->number <> 8))
     (owner   "o|owner=s")
     (group   "g|group=s")
     (srcdir  "S|srcdir=s")
     (target  "T|target=s")
     (utarget "U|uninstall=s")
     (shebang "shebang=s")
     (verb    "v")
     (dry     "n|dry-run")
     (#f      "h|help" => usage)
     (else (opt . _) (print "Unknown option : " opt) (usage))
     . args)
  ...)
```

The next example is a small test program to show the usage of `else` clause. It gathers all options into the variable *r*, except that when it sees `-c` it stops argument processing and binds the rest of the arguments to *restargs*.

```
(use gauche.parseopt)

(define (main args)
  (let1 r '()
    (let-args (cdr args)
      ((else (opt rest cont)
         (cond [(equal? opt "c") rest]
               [else (push! r opt) (cont rest)]))
       . restargs)
     (print "options: " (reverse r))
     (print "restargs: " restargs)
     0)))
```

Sample session of the above script (suppose it is saved as `example`).

```
$ ./example -a -b -c -d -e foo
```

```
options: (a b)
restargs: (-d -e foo)
$ ./example -a -b -d -e foo
options: (a b d e)
restargs: (foo)
```

Low-level API

The followings are lower-level API used to build `let-args` macro.

`parse-options` *args* (*option-clause* . . .) [Macro]

{`gauche.parseopt`} *args* is an expression that contains a list of command-line arguments. This macro scans the command-line options (an argument that begins with '`-`') and processes it as specified in *option-clause*s, then returns the remaining arguments.

Each *option-clause* is consisted by a pair of *option-spec* and its action.

If a given command-line option matches one of *option-spec*, then the associated action is evaluated. An action can be one of the following forms.

bind-spec body . . .
: *bind-spec* is a proper or dotted list of variables like lambda-list. The option's arguments are bound to *bind-spec*, then then *body* . . . is evaluated.

`=>` *proc*
: If a command-line option matches *option-spec*, calls a procedure *proc* with a list of the option's arguments.

If a symbol `else` is at the position of *option-spec*, the clause is selected when no other option clause matches a given command-line option. Three "arguments" are associated to the clause; the unmatched option, the rest of arguments, and a procedure that represents the option parser.

`make-option-parser` (*option-clause* . . .) [Macro]

{`gauche.parseopt`} This is a lower-level interface. *option-clause*s are the same as `parse-options`. This macro returns a procedure that can be used later to parse the command line options.

The returned procedure takes one required argument and one optional argument. The required argument is a list of strings, for given command-line arguments. The optional argument may be a procedure that takes more than three arguments, and if given, the procedure is used as if it is the body of `else` option clause.

9.24 `gauche.partcont` - Partial continuations

`gauche.partcont` [Module]

Gauche internally supports partial continuations (a.k.a. delimited continuations) natively. This module exposes the feature for general use.

Note: Partial continuations use two operators, `reset` and `shift`. Those names are introduced in the original papers, and stuck in the programming world. Unfortunately those names are too generic as library function names. We thought giving them more descriptive names, but decided to keep them after all; when you talk about partial continuations you can't get away from those names. If these names conflict to other names in your program, you can use `:prefix` import specifier (see Section 4.13.4 [Using modules], page 73), for example as follows:

```
;; Add prefix pc: to the 'reset' and 'shift' operators.
(use gauche.partcont :prefix pc:)

(pc:reset ... (pc:shift k ....) )
```

`reset` *expr . . .* [Macro]

{`gauche.partcont`} Saves the current continuation, and executes *expr . . .* with a *null continuation* or *empty continuation*. The `shift` operator captures the continuation from the `shift` expression to this null continuation.

Note on *implicit delimited continuations*: There's an occasion Gauche effectively calls `reset` internally: When C routine calls back to Scheme in non-CPS manner. (If you know C API, it is `Scm_EvalRec()`, `Scm_ApplyRec*()`, `Scm_Eval()` and `Scm_Apply()` family of functions.) The callers of such routines expect the result is returned at most once, which won't work well with Scheme's continuations that have unlimited extent. Such calls create delimited continuations implicitly.

For example, the `main` routine of `gosh` calls Scheme REPL by `Scm_Eval()`, which means the entire REPL is effectively surrounded by a `reset`. So, if you call `shift` without corresponding `reset`, the continuation of `shift` becomes the continuation of the entire REPL—which is to exit from `gosh`. This may be surprising if you don't know about the implicit delimited continuation.

Other places the implicit delimited continuations are created are the handlers virtual ports (see Section 9.38 [Virtual ports], page 493), `object-apply` methods called from `write` and `display`, and GUI callbacks such as the one registered by `glut-display-func` (See the document of Gauche-gl for the details), to name a few.

In general you don't need to worry about it too much, since most built-in and extension routines written in C calls back Scheme in CPS manner, and works with both full and delimited continuations.

`shift` *var expr . . .* [Macro]

{`gauche.partcont`} Packages the continuation of this expression until the current null continuation marked by the most recent `reset` into a procedure, binds the procedure to *var*, then executes *expr . . .* with the continuation saved by the most recent `reset`.

That is, after executing *expr . . .*, the value is passed to the expression waiting for the value of the most recent `reset`. When a partial continuation bound to *var* is executed, its argument is passed to the continuation waiting for the value of this `shift`. When the execution of the partial continuation reaches its end, it returns from the expression waiting for the value of invocation of *var*.

`call/pc` *proc* [Function]

{`gauche.partcont`} This is a wrapper of `shift`. `(shift k expr ...)` is equivalent to `(call/pc (lambda (k) expr ...))`. Sometimes this similarity of `call/cc` comes handy.

Well, . . . I bet you feel like your brain is twisted hard unless you are one of those rare breed from the land of continuation. Let me break down what's happening here informally and intuitively.

Suppose a procedure A calls an expression B. If A expects a return value from B and continue processing, we split the part after returning from B into a separate chunk A', then we can think of the whole control flow as this straight chain:

```
A -> B -> A'
```

`A -> B` is a procedure call and `B -> A'` is a return, but we all know procedure call and return is intrinsically the same thing, right?

Procedure B may call another procedure C, and so on. So when you look at an execution of particular piece of code, you can think of a chain of control flow like this:

```
... -> A -> B -> C -> .... -> C' -> B' -> A' -> ...
```

The magic procedure `call/cc` picks the head of the chain following its execution (marked as `*` in the figure below), and passes it to the given procedure (denoted `k` in the figure below). So, whenever `k` is invoked, the control goes through the chain from `*`.

```
... -> A -> B -> (call/cc -> (lambda (k) ... ) ) -> B' -> A' -> ...
                                     |             ^
                                     \-----------> *
```

One difficulty with `call/cc` is that the extracted chain is only one-ended—we don't know what is chained to the right. In fact, what will come after that depends on the whole program; it's outside of local control. This global attribute of `call/cc` makes it difficult to deal with.

The `reset` primitive *cuts* this chain of continuation. The original chain of continuation (the `x`-end in the following figure) is saved somewhere, and the continuation of `reset` itself becomes open-ended (the `o`-end in the following figure).

```
... -> A -> B -> (reset ... ) -> o

                                 x -> B ' -> A' -> ...
```

A rule: If control reaches to the `o`-end, we pick the `x`-end *most recently saved*. Because of this, `reset` alone doesn't show any difference in the program behavior.

Now what happens if we insert `shift` inside `reset`? This is a superficial view of inserting `shift` into somewhere down the chain of reset:

```
... -> (reset -> X -> Y -> (shift k ... ) -> Y' -> X' ) -> o
```

What actually happens is as follows.

1. `shift` packages *the rest of the chain of work* until the end of `reset`, and bind it to the variable *k*.
2. The continuation of `shift` becomes a null continuation as well, so after `shift` returns, the control skips the rest of operations until the corresponding `reset`.

```
... -> (reset -> X -> Y -> (shift k ... ) ---------> ) -> o
                                 |
                                 \-------> Y' -> X' ) -> o
```

In other words, when you consider the `reset` form as one chunk of task, then `shift` in it *stashes away* the rest of the task and immediately returns from the task.

Let's see an example. The *walker* argument in the following example is a procedure that takes a procedure and some kind of collection, and applies the procedure to the each element in the collection. We ignore the return value of *walker*.

```
(define (inv walker)
  (lambda (coll)
    (define (continue)
      (reset (walker (lambda (e) (shift k (set! continue k) e)) coll)
             (eof-object)))
    (lambda () (continue))))
```

A typical example of *walker* is `for-each`, which takes a list and applies the procedure to each element of the list. If we pass `for-each` to `inv`, we get a procedure that is inverted *inside-out*. What does that mean? See the following session:

```
gosh> (define inv-for-each (inv for-each))
inv-for-each
gosh> (define iter (inv-for-each '(1 2 3)))
iter
gosh> (iter)
1
```

```
gosh> (iter)
2
gosh> (iter)
3
gosh> (iter)
#<eof>
```

When you pass a list to `inv-for-each`, you get an iterator that returns each element in the list for each call. That's because every time `iter` is called, `shift` in `inv` stashes away the task of *walking the rest of the collection* and set it to `continue`, then returns the current element *e*.

walker doesn't need to work just on list. The following function `for-each-leaf` traverses a tree and apply *f* on each non-pair element.

```
(define (for-each-leaf f tree)
  (match tree
   [(x . y) (for-each-leaf f x) (for-each-leaf f y)]
   [x (f x)]))
```

And you can inverse it just like `for-each`.

```
gosh> (define iter2 ((inv for-each-leaf) '((1 . 2) . (3 . 4))))
iter2
gosh> (iter2)
1
gosh> (iter2)
2
gosh> (iter2)
3
gosh> (iter2)
4
gosh> (iter2)
#<eof>
```

The `util.combinations` module (see Section 12.62 [Combination library], page 794) provides a procedure that *calls* a given procedure with every permutation of the given collection. If you pass it to `inv`, you get a procedure that *returns* every permutation each time.

```
gosh> (define next ((inv permutations-for-each) '(a b c)))
next
gosh> (next)
(a b c)
gosh> (next)
(a c b)
gosh> (next)
(b a c)
gosh> (next)
(b c a)
gosh> (next)
(c a b)
gosh> (next)
(c b a)
gosh> (next)
#<eof>
```

9.25 gauche.process - High-level process interface

`gauche.process` [Module]

This module provides a higher-level API of process control, implemented on top of low-level system calls. This module also provides "process ports", a convenient way to send/receive information to/from subprocesses.

9.25.1 Running subprocess

`do-process` *cmd/args :key redirects input output error fork directory host sigmask on-abnormal-exit* [Function]

`do-process!` *cmd/args :key redirects input output error fork directory host sigmask* [Function]

`run-process` *cmd/args :key redirects input output error fork directory host sigmask wait* [Function]

{`gauche.process`} Runs a command with arguments given to *cmd/args* in a subprocess. The *cmd/args* argument must be a list, whose car specifies the command name and whose cdr is the command-line arguments.

If the command name contains a slash, it is taken as the pathname of the executable. Otherwise the named command is searched from the directories in the `PATH` environment variable.

Each element in *cmd/args* are converted to a string by `x->string`, for the convenience.

`Do-process` always waits the subprocess to terminate, and returns `#t` if it exits successfully (i.e. with zero exit status). If the subprocess terminates abnormally, it returns `#f` by default, or raise an error if `:error` is passed to the keyword argument `on-abnormal-exit`.

`Do-process!` is like `do-process` except that it raises `<process-abnormal-exit>` error when the process exists with non-zero status. It's the same behavior as giving `:error` to the `on-abnormal-exit` keyword argument of `do-process`. It is often more convenient to let the commands fail in shell-script type tasks.

`Run-process` can run the subprocess concurrently by default, that is, it returns immediately. The return value is a `<process>` object, which can be used to track the status of the subprocess (see Section 9.25.3 [Process object], page 427).

For example, the following expression runs `ls -al`.

```
(do-process '(ls -al))
```

You see the output of `ls -al`, then it returns `#t`, unless the execution of `ls` command fails with some reason.

Since `do-process` returns the success or failure of the command by a boolean value, you can use `and` and `or` to combine commands pretty much the same way as shell's `&&` and `||` operators.

```
;; shell: make && make -s check
(and (do-process '(make))
     (do-process '(make -s check)))

;; shell: mv x.tmp x.c || rm -f x.tmp
(or (do-process '(mv x.tmp x.c))
    (do-process '(rm -f x.tmp)))
```

If you use `run-process` instead, you'll get `<process>` object without waiting `ls -al` to finish. If you run the following expression on REPL, you'll likely to see the return value before output of `ls`.

```
(run-process '(ls -al))
```

You can keep the returned <process> object and call process-wait on it to wait for its termination. See Section 9.25.3 [Process object], page 427, for the details of process-wait.

```
(let1 p (run-process '(ls -al))
  ... do some other work ...
  (process-wait p))
```

You can tell run-process to wait for the subprocess to exit; in that case, run-process calls process-wait internally. It is useful if you want to examine the exit status of the subprocess, rather than just caring its success/failure as do-process does.

Note that -i is read as an imaginary number, so be careful to pass -i as a command-line argument; you should use a string, or write |-i| to make it a symbol.

```
(run-process '(ls "-i"))
```

Note: An alternative way to run external process is sys-system, which takes a command line as a single string (see Section 6.25.10 [Process management], page 271). The string is passed to the shell to be interpreted, so you can include redirections, or can pipe several commands. It would be handy for quick throwaway scripts.

On the other hand, with sys-system, if you want to change command parameters at runtime, you need to worry about properly escape them (actually we have one to do the job in gauche.process; see shell-escape-string below); you need to be aware that /bin/sh, used by sys-system via system(3) call, may differ among platforms and be careful not to rely on specific features on certain systems. As a rule of thumb, keep sys-system for really simple tasks with a constant command line, and use run-process and do-process for all other stuff.

Note: Old version of this procedure took arguments differently, like (run-process "ls" "-al" :wait #t), which was compatible to STk. This is still supported but deprecated.

Large number of keyword arguments can be passed to do-process and run-process to control execution of the child process. We describe them by categories.

Synchronization

wait *flag* [Subprocess argument]

This can only be given to run-process. If *flag* is true, run-process waits until the subprocess terminates, by calling process-wait internally. Otherwise the subprocess runs asynchronously and run-process returns immediately, which is the default behavior.

Note that if the subprocess is running asynchronously, it is the caller's responsibility to call process-wait at a certain timing to collect its exit status.

```
;; This returns after wget terminates.
(define p (run-process '(wget http://practical-scheme.net/) :wait #t))

;; Check the exit status
(let1 st (process-exit-status p)
  (cond [(sys-wait-exited? st)
         (print "wget exited with status " (sys-wait-exit-status st))]
        [(sys-wait-signaled? st)
         (print "wget interrupted by signal " (sys-wait-termsig st))]
        [else
         (print "wget terminated with unknown status " st)]))
```

on-abnormal-exit *how* [Subprocess argument]

This can only be given to do-process. If *how* is #f, which is the default, do-process returns #f when the subprocess exits abnormally (i.e. with nonzero exit status). If *how* is :error, it raises an error in such a case.

`fork` *flag* [Subprocess argument]

If *flag* is true, `do-process` and `run-process` forks to run the subprocess, which is the default behavior. If *flag* is false, `do-process` and `run-process` directly calls `sys-exec`, so it never returns.

I/O redirection

`redirects` (*iospec* ...) [Subprocess argument]

Specifies how to redirect child process's I/Os. Each *iospec* can be one of the followings, where *fd*, *fd0*, and *fd1* are nonnegative integers referring to the file descriptor *of the child process.*

(Note: If you just want to run a command and get its output as a string take a look at `process-output->string` (see Section 9.25.4 [Process ports], page 429). If you want to pipe multiple commands together, see Section 9.25.2 [Running process pipeline], page 426.)

`(< ` *fd source*`)`

source can be a string, a symbol, a keyword `:null`, an integer, or an input port.

If it is a string, it names a file opened for read and the child process can reads the content of the file from *fd*. An error is signaled if the file does not exist or cannot open for read.

If it is a symbol, an unidirectional pipe is created, whose reader end is connected to the child's *fd*, and whose writer end is available as an output port returned from `(process-input` *process source*`)`.

If it is `:null`, the child's *fd* is connected to the null device.

If it is an integer, it should specify a parent's file descriptor opened for read. The child sees the duped file descriptor as *fd*.

If it is an input port, the underlying file descriptor is duped into child's *fd*. It is an error to pass an input port without associated file descriptor (See `port-file-number` in Section 6.22.3 [Common port operations], page 218).

`(<< ` *fd value*`)`

`(<<< ` *fd obj*`)`

Feeds *value* or *obj* to the input file descriptor *fd* of the child process.

With `<<`, *value* must be either a string or a uniform vector (see Section 9.36 [Uniform vectors], page 476). It is sent to the child process as is. Using a uniform vector is good to pass binary content.

With `<<<`, *obj* can be any Scheme object, and the result of `(write-to-string obj)` is sent to the child process.

`(<& ` *fd0 fd1*`)`

Makes child process's file descriptor *fd0* refer to the same input as its file descriptor *fd1*. Note the difference from `<`; `(< 3 0)` makes the parent's stdin (file descriptor 0) be read by the child's file descriptor 3, while `(<& 3 0)` makes the child's file descriptor 3 refer to the same input as child's stdin (which may be redirected to a file or something else by another *iospec*).

See the note below on the order of processing `<&`.

`(> ` *fd sink*`)`

`(>> ` *fd sink*`)`

sink must be either a string, a symbol, a keyword `:null`, an integer or a file output port.

If it is a string, it names a file. The output of the child to the file descriptor *fd* is written to the file. If the named file already exists, `>` first truncates its content, while `>>` appends to the existing content.

For other arguments, `>` and `>>` works the same.

If *sink* is a symbol, an unidirectional pipe is created whose writer end is connected to the child's *fd*, and whose reader end is available as an input port returned by `(process-output process sink)`.

If *sink* is `:null`, child's *fd* is connected to the system's null device.

If *sink* is an integer, it must specify a parent's file descriptor opened for output. The child sees the duped file descriptor as *fd*.

If *sink* is an output port, the underlying file descriptor is duped into *fd* in the child process.

`(>& fd0 fd1)`

Makes child process's file descriptor *fd0* refer to the same output as its file descriptor *fd1*. Note the difference from `>`; `(> 2 1)` makes the child's stderr go to parent's stdout, while `(>& 2 1)` makes the child's stderr go to the same output as child's stdout (which may be redirected by another *iospec*).

```
;; Read both child's stdout and stderr
(let1 p (run-process '(command arg)
                     :redirects '((>& 2 1) (> 1 out)))
  (begin0 (port->string (process-output p 'out))
          (process-wait p)))
```

Note: You can't use the same name (symbol) more than once for the pipe of source or sink. For example, the following code signals an error:

```
(run-process '(command) :redirects '((> 1 out) (> 2 out))) ; error!
```

You can use `>&` to "merge" the output to one sink, or `<&` to "split" the input from one source, instead:

```
(run-process '(command) :redirects '((> 1 out) (>& 2 1)))
```

It is allowed to give the same file name more than once, just like the Unix shell. However, note that the file is opened individually for each file descriptor, so simply writing to them may not produce desired result (for regular files, most likely that one output would overwrite another).

Note: I/O redirections are processed *at once*, unlike the way unix shell does. For example, both of the following expression works the same way, that is, they redirect both stdout and stderr to a file `out`.

```
(run-process '(command arg) :redirects '((>& 2 1) (> 1 "out")))
(run-process '(command arg) :redirects '((> 1 "out") (>& 2 1)))
```

Most unix shells process redirections *in order*, so the following two command line works differently: The first one redirects child's stderr to the *current* stdout, which is the same as the parent's stdout, then redirects child's stdout to a file `out`. So the error messages appear in the parent's stdout. The second one first redirects the child's stdout to a file `out`, so at the time of processing `2>&1`, the child's stderr also goes to the file.

```
$ command arg 2>&1 1>out
$ command arg 1>out 2>&1
```

You can say `do-process` and `run-process` always works like the latter, regardless of the order in *redirects* argument.

If you want to redirect child's stderr to parent's stdout, you can use `>` like the following:

```
(run-process '(command arg) :redirects '((> 2 1) (> 1 "out")))
```

`input source` [Subprocess argument]
`output sink` [Subprocess argument]

`error sink` [Subprocess argument]

Redirects child's standard i/o. *source* and *sink* may be either a string, a keyword `:null`, a keyword `:pipe`, an integer file descriptor or a symbol.

These are really shorthand notations of the *redirects* argument:

```
:input x   ≡ :redirects '((< 0 x))
:output x  ≡ :redirects '((> 1 x))
:error x   ≡ :redirects '((> 2 x))
```

The keyword `:pipe` as *source* or *sink* is supported just for the backward compatibility. They work as if a symbol `stdin`, `stdout` or `stderr` is given, respectively:

```
:input :pipe   ≡ :redirects '((< 0 stdin))
:output :pipe  ≡ :redirects '((> 1 stdout))
:error :pipe   ≡ :redirects '((> 2 stderr))
```

That is, a pipe is created and its one end is connected to the child process's stdio, and the other end is available by calling `(process-input` *process*`)`, `(process-output` *process*`)` or `(process-error` *process*`)`. (That is because `process-input` and `process-output` uses `stdin` and `stdout` respectively when *name* argument is omitted, and `(process-error p)` is equivalent to `(process-output p 'stderr)`.)

See the description of *redirects* above for the meanings of the argument values.

Execution environment

`directory directory` [Subprocess argument]

If a string is given to *directory*, the process starts with *directory* as its working directory. If *directory* is `#f`, this argument is ignored. An error is signaled if *directory* is other type of objects, or it is a string but is not a name of a existing directory.

When *host* keyword argument is also given, this argument specifies the working directory of the *remote* process.

Note: `do-process` and `run-process` check the validity of *directory*, but actual `chdir(2)` is done just before `exec(2)`, and it is possible that `chdir` fails in spite of previous checks. At the moment when `chdir` fails, there's no reliable way to raise an exception to the caller, so it writes out an error message to standard error port and exits. A robust program may take this case into account.

`sigmask mask` [Subprocess argument]

Mask must be either an instance of `<sys-sigset>`, a list of integers, or `#f`. If an instance of `<sys-sigset>` is given, the signal mask of executed process is set to it. A list of integers are treated as a list of signals to mask. It is important to set an appropriate mask if you call `run-process` from multithreaded application. See the description of `sys-exec` (Section 6.25.10 [Process management], page 271) for the details.

If the *host* keyword argument is specified, this argument merely sets the signal mask of the local process (`ssh`).

`detached flag` [Subprocess argument]

When a true value is given, the new process is detached from the parent's process group and belongs to its own group. It is useful when you run a daemon process. See `sys-fork-and-exec` (see Section 6.25.10 [Process management], page 271), for the detailed description of *detached* argument.

`host hostspec` [Subprocess argument]

This argument is used to execute *command* on the remote host. The full syntax of *hostspec* is `protocol:user@hostname:port`, where *protocol:*, *user@*, or *:port* part can be omitted.

The *protocol* part specifies the protocol to communicate with the remote host; currently only `ssh` is supported, and it is also the default when *protocol* is omitted. The *user* part specifies the login name of the remote host. The *hostname* specifies the remote host name, and the *port* part specifies the alternative port number which *protocol* connects to.

The command line arguments are interpreted on the remote host. On the other hand, the I/O redirection is done on the local end. For example, the following code reads the file `/foo/bar` on the remote machine and copies its content into the local file `baz` in the current working directory.

```
(do-process '(cat "bar")
            :host "remote-host.example.com"
            :directory "/foo"
            :output "baz")
```

9.25.2 Running process pipeline

`do-pipeline` *commands :key input output error directory sigmask on-abnormal-exit* [Function]

`run-pipeline` *commands :key input output error wait directory sigmask* [Function]

{`gauche.process`} Convenience routines to run pipeline of processes at once. Example:

```
(do-pipeline '((ls "src/")
               (grep "\\.c$")
               (wc -l)))
```

This is equivalent to shell command pipeline `ls src/ | grep '\.c$' | wc -l`, i.e. shows the number of C source files in the `src` subdirectory.

The *commands* argument is a list of lists. Each list must be `cmd/args` argument `do-process/run-process` can accept. At least one command must be specified.

The specified commands will run concurrently, with the stdout of the first command is connected to the stdin of the second, and stdout of the second to the stdin of the third, and so on. The stdin of the first command is fed from the source specified by the *input* keyword argument, and the stdout of the last command is sent to the sink specified by the *output* keyword argument. The default values of these are the calling process's stdin and stdout, respectively. See `do-process/run-process`, for the possible values of these arguments (see Section 9.25.1 [Running subprocess], page 421).

The stderr of all the processes are sent to the sink specified by the *error* keyword argument, which is defaulted by the calling process's stderr.

Like `do-process`, `do-pipeline` waits for completion of all the processes, and returns `#t` if the tail process succeeds (i.e. exits with zero status) or `#f` if the last process fails (i.e. exits with non-zero status). If you give `:error` to `on-abnormal-exit` keyword arguments, however, a failure of the tail process raises an error. Exit statuses of subprocesses other than the tail one are collected by `process-wait`, but won't affect the return value, and won't cause an error even `on-abnormal-exit` is `:error`.

On the other hand, `run-pipeline` returns a `<process>` object of the tail process. You can get other process objects in the pipeline by applying `process-upstreams` to the tail process. By default, `run-pipeline` runs all the subprocesses in background and returns immediately. Calling `process-wait` on the returned process object will waits for all the subprocesses. If you give a true value to *wait* keyword argument, `run-process` waits for all the subprocesses to finish before returning.

The *directory* and *sigmask* keyword arguments are applied to all the processes; see `do-process/run-process` for the description of these arguments (see Section 9.25.1 [Running subprocess], page 421).

Note: In Gauche 0.9.5, we introduced `run-process-pipeline`. It is similar to the current `run-pipeline` but returns a list of subprocess objects instead of a single one. We realized it's not very convenient, so we deprecated `run-process-pipeline` and replaced it with `run-pipeline`. We still support `run-process-pipeline`, but strongly recommend to move to `run-pipeline` as soon as possible.

9.25.3 Process object

`<process>` [Class]

{`gauche.process`} An object to keep the status of a child process. You can create the process object by `run-process` procedure described below. The process ports explained in the next section also use process objects.

The `<process>` class keeps track of the child processes spawned by high-level APIs such as `run-process` or `open-input-process`. The exit status of such children must be collected by `process-wait` or `process-wait-any` calls, which also do some bookkeeping. Using the low-level process calls such as `sys-wait` or `sys-waitpid` directly will cause inconsistent state.

`<process-abnormal-exit>` [Class]

{`gauche.process`} A condition type mainly used by the process port utility procedures. Inherits `<error>`. This type of condition is thrown when the high-level process port utilities detect the child process exited with non-zero status code.

`process` [Instance Variable of `<process-abnormal-exit>`]

A process object.

Note: In Unix terms, exiting a process by calling `exit(2)` or returning from `main()` is a normal exit, regardless of the exit status. Some commands do use non-zero exit status to tell one of the normal results of execution (such as `grep(1)`). However, large number of commands uses non-zero exit status to indicate that they couldn't carry out the required operation, so we treat them as exceptional situations.

`process?` *obj* [Function]

{`gauche.process`} ≡ `(is-a?` *obj* `<process>)`

`process-pid` (*process* <*process*>) [Method]

{`gauche.process`} Returns the process ID of the subprocess *process*.

`process-command` (*process* <*process*>) [Method]

{`gauche.process`} Returns the command invoked in the subprocess *process*.

`process-input` (*process* <*process*>) *:optional name* [Method]
`process-output` (*process* <*process*>) *:optional name* [Method]

{`gauche.process`} Retrieves one end of a pipe, whose another end is connected to the process's input or output, respectively. *name* is a symbol given to the *redirects* argument of `run-process` to distinguish the pipe. See the following example:

```
(let1 p (run-process '(command arg)
                     :redirects '((< 3 aux-in)
                                  (> 4 aux-out)))
  (let ([auxin  (process-input p 'aux-in)]
        [auxout (process-output p 'aux-out)])
    ;; feed something to the child's input
    (display 'something auxin)
    ;; read data from the child's output
    (read-line auxout)
```

```
    ...
    )
  (process-wait p))
```

The symbols `aux-in` and `aux-out` is used to identify the pipes. Note that `process-input` returns *output* port, and `process-output` returns *input* port.

When *name* is omitted, `stdin` is used for `process-input` and `stdout` is used for `process-output`. These are the names used if child's stdin and stdout are redirected by `:input :pipe` and `:output :pipe` arguments, respectively.

If there's no pipe with the given name, `#f` is returned.

```
(let* ((process (run-process '("date") :output :pipe))
       (line (read-line (process-output process))))
  (process-wait process)
  line)
 ⇒ "Fri Jun 22 22:22:22 HST 2001"
```

If *process* is a result of `run-pipeline`, `(process-input process)` and `(process-input process 'stdin)` behave slightly differently—they return the pipe connected to the stdin of the *head* process of the pipeline, not the process represented by *process* (which is the tail of the pipeline). This allows you to treat the whole pipeline as one entity.

```
(let1 p (run-pipeline `((cat)
                        (grep "aba"))
                       :input :pipe :output :pipe)
  (display "banana\nhabana\ntabata\ncabara\n"
           (process-input p))    ; head of the pipeline
  (close-port (process-input p))
  (process-wait p)
  (port->string (process-output p)))
  ⇒ "habana\ntabata\ncabara\n"
```

`process-error` (*process* <*process*>) [Method]

{`gauche.process`} This is equivalent to `(process-output` *process* `'stderr)`.

`process-alive?` *process* [Function]

{`gauche.process`} Returns true if *process* is alive. Note that Gauche can't know the subprocess' status until it is explicitly checked by `process-wait`.

`process-upstreams` *process* [Function]

{`gauche.process`} If *process* is the result of `run-pipeline`, this returns a list of processes that are upstream of *process* in the pipeline. If *process* is not the result of `run-pipeline`, this returns an empty list.

```
(define p (run-pipeline `((cat) (grep "ho") (wc)) :input :pipe))

p ⇒ #<process 20658 "wc" active>

(process-upstreams p)
  ⇒ (#<process 20656 "cat" active> #<process 20657 "grep" active>)
```

`process-list` [Function]

{`gauche.process`} Returns a list of active processes. The process remains active until its exit status is explicitly collected by `process-wait`. Once the process's exit status is collected and its state changed to inactive, it is removed from the list `process-list` returns.

`process-wait` *process :optional nohang error-on-nonzero-status* [Function]

{`gauche.process`} Obtains the exit status of the subprocess *process*, and stores it to *process*'s status slot. The status can be obtained by `process-exit-status`.

This suspends execution until *process* exits by default. However, if a true value is given to the optional argument *nohang*, it returns immediately if *process* hasn't exit.

If a true value is given to the optional argument *error-on-nonzero-status*, and the obtained status code is not zero, this procedure raises `<process-abnormal-exit>` error.

Returns `#t` if this call actually obtains the exit status, or `#f` otherwise.

If the process object is created by `run-pipeline` (see Section 9.25.2 [Running process pipeline], page 426), `process-wait` waits *all* of the subprocesses in the pipeline, not just the last one, unless true value is given to the *nohang* argument. However, *error-on-nonzero-status* only affects to the status of *process*, which represents the last process in the pipeline; if an other subprocess exits with nonzero status, it is stored in its respective process objects, but won't cause a fuss.

If you specify a true value to *nohang* for the pipelined process, `process-wait` still probes other subprocesses in the pipeline and updates exit statuses of terminated ones, but doesn't wait unterminated subprocesses. The unterminated subprocesses should be waited individually, or by `process-wait-any`, to collect their exit statuses.

`process-wait-any` *:optional nohang* [Function]

{`gauche.process`} Obtains the exit status of any of the subprocesses created by `run-process`. Returns a process object whose exit status is collected.

If a true value is given to the optional argument *nohang*, this procedure returns `#f` immediately even if no child process has exit. If *nohang* is omitted or `#f`, this procedure waits for any of children exits.

If there's no child processes, this procedure immediately returns `#f`.

`process-exit-status` *process* [Function]

{`gauche.process`} Returns exit status of *process* retrieved by `process-wait`. If this is called before `process-wait` is called on *process*, the result is undefined.

The meaning of exit status depends on the platform. You need to use `sys-wait-exited?` or `sys-wait-signaled?` to see if it is terminated voluntarily or by a signal, and use `sys-wait-exit-status` or `sys-wait-termsig` to extract the exit code or the terminating signal (see Section 6.25.10 [Process management], page 271).

`process-send-signal` *process signal* [Function]

{`gauche.process`} Sends a signal *signal* to the subprocess *process*. *signal* must be an exact integer for signal number. See Section 6.25.7 [Signal], page 260, for predefined variables of signals.

`process-kill` *process* [Function]
`process-stop` *process* [Function]
`process-continue` *process* [Function]

{`gauche.process`} Sends SIGKILL, SIGSTOP and SIGCONT to *process*, respectively.

9.25.4 Process ports

`open-input-process-port` *command :key input error encoding conversion-buffer-size* [Function]

{`gauche.process`} Runs *command* asynchronously in a subprocess. Returns two values, an input port which is connected to the stdout of the running subprocess, and a process object.

Command can be a string or a list.

If it is a string, it is passed to `/bin/sh`. You can use shell metacharacters in this form, such as environment variable interpolation, globbing, and redirections. If you create the command line by concatenating strings, it's your responsibility to ensure escaping special characters if you don't want the shell to interpret them. The `shell-escape-string` function described below might be a help.

If *command* is a list, each element is converted to a string by `x->string` and then passed directly to `sys-exec` (the `car` of the list is used as both the command path and the first element of argv, i.e. `argv[0]`). Use this form if you want to avoid the shell from interfering; i.e. you don't need to escape special characters.

The subprocess's stdin is redirected from `/dev/null`, and its stderr shares the calling process's stderr by default. You can change these by giving file pathnames to *input* and *error* keyword arguments, respectively.

You can also give the *encoding* keyword argument to specify character encoding of the process output. If it differs from the Gauche's internal encoding format, `open-input-process-port` inserts a character encoding conversion port. If *encoding* is given, the *conversion-buffer-size* keyword argument can control the conversion buffer size. See Section 9.4 [Character code conversion], page 339, for the details of character encoding conversions.

```
(receive (port process) (open-input-process-port "ls -l Makefile")
  (begin0 (read-line port)
          (process-wait process)))
 ⇒ "-rw-r--r--   1 shiro    users        1013 Jun 22 21:09 Makefile"

(receive (port process) (open-input-process-port '(ls -l "Makefile"))
  (begin0 (read-line port)
          (process-wait process)))
 ⇒ "-rw-r--r--   1 shiro    users        1013 Jun 22 21:09 Makefile"

(open-input-process-port "command 2>&1")
 ⇒ ;the port reads both stdout and stderr

(open-input-process-port "command 2>&1 1>/dev/null")
 ⇒ ;the port reads stderr
```

The exit status of subprocess is not automatically collected. It is the caller's responsibility to issue `process-wait`, or the subprocess remains in a zombie state. If it bothers you, you can use one of the following functions.

`call-with-input-process` *command proc :key input error encoding conversion-buffer-size on-abnormal-exit* [Function]

{`gauche.process`} Runs *command* in a subprocess and pipes its stdout to an input port, then call *proc* with the port as an argument. When *proc* returns, it collects its exit status, then returns the result *proc* returned. The cleanup is done even if *proc* raises an error.

The keyword argument *on-abnormal-exit* specifies what happens when the child process exits with non-zero status code. It can be either `:error` (default), `:ignore`, or a procedure that takes one argument. If it is `:error`, a `<process-abnormal-exit>` error condition is thrown by non-zero exit status; the `process` slot of the condition holds the process object. If it is `:ignore`, nothing is done for non-zero exit status. If it is a procedure, it is called with a process object; when the procedure returns, `call-with-input-process` returns normally.

The semantics of *command* and other keyword arguments are the same as `open-input-process-port` above.

```
(call-with-input-process "ls -l *"
  (lambda (p) (read-line p)))
```

`with-input-from-process` *command thunk :key input error encoding conversion-buffer-size on-abnormal-exit* [Function]

{`gauche.process`} Runs *command* in a subprocess, and calls *thunk* with its current input port connected to the command's stdout. The command is terminated and its exit status is collected, after *thunk* returns or raises an error.

The semantics of *command* and keyword arguments are the same as `call-with-input-process` above.

```
(with-input-from-process "ls -l *" read-line)
```

`open-output-process-port` *command :key output error encoding conversion-buffer-size* [Function]

{`gauche.process`} Runs `command` in a subprocess asynchronously. Returns two values, an output port which is connected to the stdin of the subprocess. and the process object.

The semantics of *command* is the same as `open-input-process-port`. The semantics of *encoding* and *conversion-buffer-size* are also the same.

The subprocess's stdout is redirected to `/dev/null` by default, and its stderr shares the calling process's stderr. You can change these by giving file pathnames to *output* and *error* keyword arguments, respectively.

The exit status of the subprocess is not automatically collected. The caller should call `process-wait` on the subprocess at appropriate time.

`call-with-output-process` *command proc :key output error encoding conversion-buffer-size on-abnormal-exit* [Function]

{`gauche.process`} Runs `command` in a subprocess, and calls *proc* with an output port which is connected to the stdin of the command. The exit status of the command is collected after either *proc* returns or raises an error.

The semantics of keyword arguments are the same as `open-output-process-port`, except *on-abnormal-exit*, which is the same as described in `call-with-input-process`.

```
(call-with-output-process "/usr/sbin/sendmail"
  (lambda (out) (display mail-body out)))
```

`with-output-to-process` *command thunk :key output error encoding conversion-buffer-size on-abnormal-exit* [Function]

{`gauche.process`} Same as `call-with-output-process`, except that the output port which is connected to the stdin of the command is set to the current output port while executing *thunk*.

`call-with-process-io` *command proc :key error encoding conversion-buffer-size on-abnormal-exit* [Function]

{`gauche.process`} Runs *command* in a subprocess, and calls *proc* with two arguments; the first argument is an input port which is connected to the command's stdout, and the second is an output port connected to the command's stdin. The error output from the command is shared by the calling process's, unless an alternative pathname is given to the *error* keyword argument.

The exit status of the command is collected when *proc* returns or raises an error.

`process-output->string` *command :key error encoding conversion-buffer-size on-abnormal-exit* [Function]

`process-output->string-list` *command :key error encoding conversion-buffer-size on-abnormal-exit* [Function]

{`gauche.process`} Runs *command* and collects its output (to stdout) and returns them. `process-output->string` concatenates all the output from *command* to one string, replacing any sequence of whitespace characters to single space. The action is similar to "command substitution" in shell scripts. `process-output->string-list` collects the output from *command* line-by-line and returns the list of them. Newline characters are stripped.

Internally, *command* is run by `call-with-input-process`, to which keyword arguments are passed.

```
(process-output->string '(uname -smp))
  ⇒ "Linux i686 unknown"

(process-output->string '(ls))
  ⇒ "a.out foo.c foo.c~ foo.o"

(process-output->string-list '(ls))
  ⇒ ("a.out" "foo.c" "foo.c~" "foo.o")
```

`shell-escape-string` *str :optional flavor* [Function]

{`gauche.process`} If *str* contains characters that affects shell's command-line argument parsing, escape *str* to avoid shell's interpretation. Otherwise, returns *str* itself.

The optional *flavor* argument takes a symbol to specify the platform; currently `windows` and `posix` can be specified. The way shell handles the escape and quotation differ a lot between these platforms; the `windows` flavor uses MSVC runtime argument parsing behavior, while the `posix` flavor assumes IEEE Std 1003.1. When omitted, the default value is chosen according to the running platform. (Note: Cygwin is regarded as `posix`.)

Use this procedure when you need to build a command-line string by yourself. (If you pass a command-line argument list, instead of a single command-line string, you don't need to escape them since we bypass the shell.)

`shell-tokenize-string` *str :optional flavor* [Function]

{`gauche.process`} Split a string *str* into arguments as the shell does.

```
(shell-tokenize-string "grep -n -e \"foo bar\" log")
 ⇒ ("grep" "-n" "-e" "foo bar" "log")
```

The optional *flavor* arguments can be a symbol either `windows` or `posix` to specify the syntax. If it's `windows`, we follow MSVC runtime command-line argument parser behavior. If it's `posix`, we follow IEEE Std 1003.1 Shell Command Language. When omitted, the default value is chosen according to the running platform. (Note: Cygwin is regarded as `posix`.)

This procedure does not handle fancier shell features such as variable substitution. If it encounters a metacharacter that requires such interpretation, an error is signaled. In other words, metacharacters must be properly quoted in *str*.

```
(shell-tokenize-string "echo $foo" 'posix)
  ⇒ signals error

(shell-tokenize-string "echo \"$foo\"" 'posix)
  ⇒ still signals error

(shell-tokenize-string "echo '$foo'" 'posix)
  ⇒ ("echo" "$foo")

(shell-tokenize-string "echo \\$foo" 'posix)
  ⇒ ("echo" "$foo")
```

9.25.5 Process connection

`<process-connection>` [Class]

A connection abstraction to communicate with an external processs. Inherits `<connection>`. See Section 9.8 [Connection framework], page 365, for the details of the connection interface.

This is useful to give an external process to the code that expects connection. For example, instead of direct network connection, you can insert a filter process between remote server and your client code.

`make-process-connection` *process-or-spec* [Function]

Run an external process and returns a connection that's connected to standard I/O of the process.

You can pass a list of command and its arguments, or a <*process*> object, to the *process-or-spec* argument. If you it's a `<process>` object, it's stdin and stdout is connected to pipes. If it is a list, it is passed to `run-process` to run a new process.

Shutting down both channels of the connection terminates the process. Most processes that reads from stdin would exits after it reads EOF from input, so we just poll the process exit status for a short period of time. If the process doesn't exit, we send signals (first SIGTERM, then SIGKILL) to ensure the termination of the process.

Merely closing the connection doesn't terminate the process, so that the forked process can keep talking to the process.

9.26 `gauche.record` - Record types

`gauche.record` [Module]

This module provides a facility to define *record types*, user-defined aggregate types. The API is upper compatible to SRFI-9 (Defining Record Types) and SRFI-99 (ERR5RS Records).

Record types are implemented as Gauche's classes, but have different characteristics from the general classes. See Section 9.26.1 [Record types introduction], page 433, for when you want to use record types.

The record API consists of three layers, following SRFI-99 and R6RS design.

The syntactic layer is the `define-record-type` macro that conveniently defines a record type and related procedures (a constructor, a predicate, accessors and modifiers) all at once declaratively. Knowing this macro alone is sufficient for most common usage of records.

The inspection layer defines common procedures to query information to the records and record types.

The procedural layer is a low-level machinery to implement the syntactic layer; you don't usually need to use them in day-to-day programming, but they might be handy to create record types on-the-fly at runtime.

9.26.1 Introduction

Gauche provides a general way for users to define new types as new classes, using object system (see Chapter 7 [Object system], page 279), and indeed record types are implemented as Gauche's classes. However, using record types instead of classes has several advantages.

- It is portable. The API conforms two major record SRFIs, SRFI-9 and SRFI-99, so the code using record types can run on various Scheme systems.
- It is efficient. Record types are less flexibile than classes, but that allows Gauche to optimize more. Hence creating records and accessing/modifying them are much faster than creating instances of general classes and accessing/modifying them. It makes record types preferable

choice when you only need a mechanism to bundle several related values to carry around, and don't need fancier mechanisms such as class redefinitions.

- As Gauche's extension, you can define pseudo record types, which interprets ordinary aggregate types such as vectors and lists as records. (For Common Lisp users; it is like the `:type` option of `defstruct`). This helps flexibility of interface. For example, you can ask your library's users to pass a point in a vector of three numbers, instead of asking users to pack their point data into your custom point record type. Yet inside your library you can treat the passed data as if it is your point record type. See Section 9.26.5 [Pseudo record types], page 438, for more details.

The disadvantage of record types is that they don't obey Gauche's class redefinition protocol (see Section 7.2.5 [Class redefinition], page 292). That is, if you redefine a record with the same name, it creates a new record type unrelated to the old one. The record instances created from the old definition won't be updated according to the new definition.

More importantly, record constructors, accessors and modifiers are tend to be inlined where they are used, to achieve better performance. Since they are inlined, the code that uses those procedures are not affected when the record type is redefined. This means if you redefine a record type, you have to reload (recompile) the sources that uses any of record constructors, accessors or modifiers.

9.26.2 Syntactic Layer

`define-record-type` *type-spec ctor-spec pred-spec field-spec . . .* [Macro]

[SRFI-9][SRFI-99+] {`gauche.record`} Defines a record type, and optionally defines a constructor, a predicate, and field accessors and modifiers.

The *type-spec* argument names the record type, and optionally specifies the supertype (*parent*).

```
type-spec : type-name | (type-name parent)

type-name : identifier
parent : expression
```

The *type-name* identifier will be bound to a *record type descriptor*, or rtd, which can be used for introspection and reflection. See Section 9.26.3 [Record types inspection layer], page 436, and Section 9.26.4 [Record types procedural layer], page 437, for possible operations for record types. In Gauche, a record type descriptor is a `<class>` with a metaclass `<record-meta>`.

The *parent* expression should evaluate to a record type descriptor. If given, the defined record type inherits it; that is, all the slots defined in the parent type are available to the *type-name* as well, and the instance of *type-name* answers `#t` to the predicate of the parent type.

Since a record type is also a class, parent type is also a superclass of the defined record type. However, record types are limited to have single inheritance.

You can give a pseudo record base type as *parent* to define a pseudo record type, which allows you to access ordinary aggregates like vectors as records. See Section 9.26.5 [Pseudo record types], page 438, for more details.

The *ctor-spec* defines the constructor of the record instance.

```
ctor-spec : #f | #t | ctor-name
          | (ctor-name field-name ...)

ctor-name : identifier
field-name : identifier
```

If it is `#f`, no constructor is created. If it is `#t`, a default constructor is created with a name `make-`*`type-name`*. If it is a single identifier *ctor-name*, a default constructor is created with the name. The default constructor takes as many arguments as the number of fields of the record, including inherited ones if any. When called, it allocates an instance of the record, and initialize its fields with the given arguments in the order (inherited fields comes first), and returns the record.

The last variation of *ctor-spec* creates a custom constructor with the name *ctor-name*. The custom constructor takes as many arguments as the given *field-names*, and initializes the named fields. If the inherited record type has a field of the same name as the ancestor record type, only the inherited ones are initialized. In Gauche, uninitialized fields remains unbound until some value is set to it.

The *pred-spec* defines the predicate of the record instance, which takes one argument and returns `#t` iff it is an instance of the defined record type or its descendants.

```
pred-spec : #f | #t | pred-name

pred-name : identifier
```

If it is `#f`, no predicate is created. If it is `#t`, a predicate is created with a name *type-name*`?`. If it is a single identifier, a predicate is created with the given name.

The rest of the arguments specify fields (slots) of the record.

```
field-spec
  : field-name    ; immutable, with default accessor
  | (field-name)  ; mutable, with default accessor/modifier
  | (field-name accessor-name); immutable
  | (field-name accessor-name modifier-name); mutable

field-name    : identifier
accessor-name : identifier
modifier-name : identifier
```

The first and the third forms define immutable fields, which can only be initialized by the constructor but cannot be modified afterwards (thus such fields don't have modifiers). The second and the fourth forms define mutable fields.

The third and fourth forms explicitly name the accessor and modifier. With the first and second forms, on the other hand, the accessor is named as *type-name-field-name*, and the modifier is named as *type-name-field-name*`-set!`.

Let's see some examples. Here's a definition of a record type `point`.

```
(define-record-type point #t #t
  x y z)
```

The variable `point` is bound to a record type descriptor, which is just a class. But you can take its class and see it is indeed an instance of `<record-meta>` metaclass.

```
point            ⇒ #<class point>
(class-of point) ⇒ #<class <record-meta>>
```

You can create an instance of `point` by the default constructor `make-point`. The predicate is given the default name `point?`, and you can access the fields of the created record by `point-x` etc.

```
(define p (make-point 1 2 3))

(point? p)  ⇒ #t
(point-x p) ⇒ 1
(point-y p) ⇒ 2
(point-z p) ⇒ 3
```

Since we defined all fields immutable, we cannot modify the instance `p`.

Here's a mutable version of point, `mpoint`. You can modify its fields by modifier procedures and generalized `set!`.

```
(define-record-type mpoint #t #t
  (x) (y) (z))

(define p2 (make-mpoint 1 2 3)) ; create an instance

(mpoint-x p2)  ⇒ 1

(mpoint-x-set! p2 4)            ; default modifier
(mpoint-x p2)  ⇒ 4

(set! (mpoint-x p2) 6)          ; generalized set! also works
(mpoint-x p2)  ⇒ 6
```

Next one is an example of inheritance. Note that the default constructor takes arguments for fields of the parent record as well.

```
(define-record-type (qpoint mpoint) #t #t
  (w))

(define p3 (make-qpoint 1 2 3 4))

(qpoint? p3)  ⇒ #t       ; p3 is a qpoint
(mpoint? p3)  ⇒ #t       ; ... and also an mpoint

(mpoint-x p3) ⇒ 1        ; accessing inherited field
(mpoint-y p3) ⇒ 2
(mpoint-z p3) ⇒ 3
(qpoint-w p3) ⇒ 4
```

A small caveat: Accessors and modifiers for inherited fields (e.g. `qpoint-x` etc.) are not created.

Gauche's convention is to enclose class name by `<>`. You can follow the convention and still explicitly gives simpler names (instead of `make-<point>` or `<point>-x`):

```
(define-record-type <point> make-point point?
  (x point-x)
  (y point-y)
  (z point-z))
```

9.26.3 Inspection layer

This layer provides common procedures that operates on record type descriptors and record instances.

Note that a record type descriptor is a class in Gauche, so you can also use operators on classes (e.g. `class-name`, `class-slots` etc.) on record type descriptors as well. However, these procedures are more portable.

`record?` *obj* [Function]

[SRFI-99][R6RS] {`gauche.record`} Returns `#t` iff *obj* is an instance of record type, `#f` otherwise.

`record-rtd` *record* [Function]

[SRFI-99][R6RS] {`gauche.record`} Returns the record type descriptor of the record instance.

`rtd-name` *rtd* [Function]
[SRFI-99] {`gauche.record`} Returns the name of the record type descriptor *rtd*.

`rtd-parent` *rtd* [Function]
[SRFI-99] {`gauche.record`} Returns the parent type of the record type descriptor *rtd*. If *rtd* doesn't have a parent, `#f` is returned.

`rtd-field-names` *rtd* [Function]
[SRFI-99] {`gauche.record`} Returns a vector of symbols, each of which is the names of the direct fields of the record represented by *rtd*. The result doesn't include inherited fields.

`rtd-all-field-names` *rtd* [Function]
[SRFI-99] {`gauche.record`} Returns a vector of symbols, each of which is the names of the fields of the record represented by *rtd*. The result includes all inherited fields.

`rtd-field-mutable?` *rtd field-name* [Function]
[SRFI-99] {`gauche.record`} Returns `#t` iff the field with the name *field-name* of a record represented by *rtd* is mutable.

9.26.4 Procedural layer

These procedures are low-level machinery on top of which `define-record-type` is implemented. They can be used to create a new record type at runtime.

`make-rtd` *name field-specs :optional parent* [Function]
[SRFI-99] {`gauche.record`} Creates and returns a new record type descriptor with name *name* and having fields specified by *field-specs*. If *parent* is given, it must be a record type descriptor or `#f`. If it is a record type descriptor, the created record type inherits from it.

The *field-specs* argument must be a vector, each element of which is a *field specifier*. A field specifier can be a symbol, a list `(mutable symbol)`, or a list `(immutable symbol)`. The *symbol* names the field. A single symbol or `(mutable symbol)` format makes the field mutable, and `(immutable symbol)` format makes the field immutable.

Note: Gauche does not implement the extension suggested in SRFI-99 yet, which is `sealed`, `opaque` and `uid` arguments.

`rtd?` *obj* [Function]
[SRFI-99] {`gauche.record`} Returns `#t` if *obj* is a record type descriptor, `#f` otherwise.

`rtd-constructor` *rtd :optional field-specs* [Function]
[SRFI-99] {`gauche.record`} Returns a procedure that creates an instance record of the record type represented by *rtd*. Without *field-specs*, it returns the default constructor, which takes as many arguments as the number of fields of the record to initialize them.

You can give a vector of symbols as *field-specs*. The *n*-th symbol specifies which field of the instance should be initialized by the *n*-th argument. The *field-specs* vector cannot contain duplicate names. If the record type defines a field with the same name as the one in the parent record type, the custom constructor can only initialize the field of the derived type's instance.

`rtd-predicate` *rtd* [Function]
[SRFI-99] {`gauche.record`} Returns a predicate to test an object is an instance of *rtd*.

If *rtd* is a pseudo record type, the predicate merely tests the given object is in an appropriate type and has enough size to hold the contents. See Section 9.26.5 [Pseudo record types], page 438, for the details.

`rtd-accessor` *rtd field-name* [Function]

[SRFI-99] {`gauche.record`} Returns a procedure that takes one argument, an instance of *rtd*, and returns the value of the *field-name* of the instance.

An error is signaled if the record type doesn't have the field of name *field-name*.

If *rtd* is inherits other record types, and it defines a field of the same name as inherited ones, then the accessor returned by this procedure retrieves the value of the field of the derived record.

`rtd-mutator` *rtd field-name* [Function]

[SRFI-99] {`gauche.record`} Returns a procedure that takes two arguments, an instance of *rtd* and a value, and sets the latter as the value of the *field-name* of the instance.

An error is signaled if the record type doesn't have the field of name *field-name*, or the named field is immutable.

Like `rtd-accessor`, if the record has a field with the same name as inherited one, the modifier returned by this procedure only modifies the field of the derived record.

9.26.5 Pseudo record types

A pseudo record type is a record type that does not create an instance of its own type. Instead it treats an object of other collection types, such as a vector, as if it had named fields. It's easier to understand by an example:

```
(define-record-type (vpoint (pseudo-rtd <vector>)) #t #t
  (x) (y) (z))

(make-vpoint 1 2 3)  ⇒ #(1 2 3)
(vpoint-x '#(1 2 3)) ⇒ 1

(rlet1 v (make-vpoint 1 2 3)
  (set! (vpoint-y v) -1))
 ⇒ #(1 -1 3)
```

To create a pseudo record type, specify another pseudo record type as a parent. The procedure `pseudo-rtd` can be used to obtain a base pseudo record type of the suitable instance class.

`pseudo-rtd` *instance-class* [Function]

{`gauche.record`} Returns a pseudo rtd suitable to use *instance-class* as a pseudo record.

Currently, `<list>`, `<vector>`, and uniform vector classes (`<u8vector>` etc.) are supported as *instance-class*.

The predicates of a pseudo record return `#t` if the given object can be interpreted as the pseudo record. In the above example of `vpoint` record, the predicate `vpoint?` returns `#t` iff the given object is a vector with 3 or more elements:

```
(vpoint? '#(0 0 0))   ⇒ #t
(vpoint? '#(0 0))     ⇒ #f
(vpoint? '(0 0 0))    ⇒ #f
(vpoint? '#(0 0 0 0)) ⇒ #t
```

We allow more elements so that the pseudo record can be used to interpret the header part of the longer data.

9.27 gauche.reload - Reloading modules

`gauche.reload` [Module]

In the development cycle, you often have to reload modules frequently. This module supports it.

Note that some part of semantics of the program depends on the order of loading modules, so reloading arbitrary modules may change the program behavior unexpectedly. This module is for developers who knows what they are doing.

Redefinition rules: Reloading a module resets all the binding in the module by default. Sometimes it is not desirable, however. For example, you might want to keep an intermediate results in some variable. You can specify rules for the reloading procedure to determine which binding to keep.

The rule is described in the following syntax.

```
<module-rules> : (<module-rule> ...)
<module-rule>  : (<module-pattern> <rule> ...)
<module-pattern> : a symbol module name, or a symbol containing glob pattern
<rule>         : procedure | symbol | regexp
               | (and <rule> ...)
               | (or  <rule> ...)
               | (not <rule>)
```

`<module-rules>` is the global rule to determine per-module rules. `<module-pattern>` is either a symbol module name or a symbol that contains glob pattern (e.g. `mylib.*`). If `<rule>` is a procedure, it is used as a predicate and the bindings whose value satisfies the predicate are kept from redefinition. If `<rule>` is a symbol, the binding of the variable whose name is the symbol is kept. If `<rule>` is a regexp, the bindings of the variable whose name matches the regexp are kept.

Note that the mechanism to prevent redefinition is kind of ad-hoc hack and semantically unclean. Especially, the right-hand expressions of `define`s are still evaluated, so any side effects they have will be in effect (e.g. `define-class` would still redefine a class). It's just for your convenience. Take a look at the code if you want to know the exact behavior.

`reload` *module-name :optional rule . . .* [Function]

{`gauche.reload`} Reloads the specified module. You can optionally specify redefinition rules by *rule* . . ., where each *rule* is the term `<rule>` defined above.

`reload-modified-modules` *:optional module-rules* [Function]

{`gauche.reload`} Reloads module(s) that have been modified since they are loaded last time. If optional *module-rules* is given, it is used to determine the redefinition rules for reloaded modules. If *module-rules* is omitted, the current rules are used. The default of current rules is empty. You can set the current rules by `module-reload-rules`.

`module-reload-rules` *:optional module-rules* [Function]

{`gauche.reload`} This is a parameter (see Section 9.22 [Parameters], page 411) that keeps the default module rules for `reload-modified-modules`. If called without arguments, returns the current module rules. If called with *module-rules*, sets the argument to the current module rules.

`reload-verbose` *:optional flag* [Function]

{`gauche.reload`} This is a parameter to control verbosity of the reloading procedures. If called without arguments, returns the current verbosity flag. If called with *flag*, it is set to the current verbosity flag.

9.28 gauche.selector - Simple dispatcher

`gauche.selector` [Module]

This module provides a simple interface to dispatch I/O events to registered handlers, based on `sys-select` (see Section 6.25.11 [I/O multiplexing], page 274).

`<selector>` [Class]

{`gauche.selector`} A dispatcher instance that keeps watching I/O ports with associated handlers. A new instance can be created by `make` method.

`selector-add!` (*self* <*selector*>) *port-or-fd proc flags* [Method]

{`gauche.selector`} Add a handler *proc* to the selector. *proc* is called when *port-or-fd*, which should be a port object or an integer that specifies a system file descriptor, meets a certain condition specified by *flags*. *flags* must be a list of one or more of the following symbols.

`r` Calls *proc* when data is available at *port-or-fd* to read.

`w` Calls *proc* when *port-or-fd* is ready to be written.

`x` Calls *proc* when an exceptional condition occurs on *port-or-fd*.

proc is called with two arguments. The first one is *port-or-fd* itself, and the second one is a symbol `r`, `w` or `x`, indicating the condition.

If a handler is already associated with *port-or-fd* under the same condition, the previous handler is replaced by *proc*.

`selector-delete!` (*self* <*selector*>) *port-or-fd proc flags* [Method]

{`gauche.selector`} Deletes the handler entries that matches *port-or-fd*, *proc* and *flags*. One or more of the arguments may be `#f`, meaning "don't care". For example,

```
(selector-delete! selector the-port #f #f)
```

deletes all the handlers associated to *the-port*, and

```
(selector-delete! selector #f #f '(w))
```

deletes all the handlers waiting for writable condition.

`selector-select` (*self* <*selector*>) *:optional* (*timeout #f*) [Method]

{`gauche.selector`} Dispatcher body. Waits for the conditions registered in *self*, and when it occurs, calls the associated handler. If the *timeout* argument is omitted or false, this method waits indefinitely. Alternatively you can give a timeout value, that can be a real number in microseconds, or a list of two integers that represents seconds and microseconds.

Returns the number of handlers called. Zero means the selector has been timed out.

It is safe to modify *self* inside handler. The change will be effective from the next call of `selector-select`

This is a simple example of "echo" server:

```
(use gauche.net)
(use gauche.selector)
(use gauche.uvector)

(define (echo-server port)
  (let ((selector (make <selector>))
        (server   (make-server-socket 'inet port :reuse-addr? #t)))

    (define (accept-handler sock flag)
      (let* ((client (socket-accept server))
```

```
                (output (socket-output-port client)))
           (selector-add! selector
                          (socket-input-port client :buffering #f)
                          (lambda (input flag)
                            (echo client input output))
                          '(r))))

       (define (echo client input output)
         (let ((str (read-uvector <u8vector> 4096 input)))
           (if (eof-object? str)
               (begin (selector-delete! selector input #f #f)
                      (socket-close client))
               (begin (write-uvector str output)
                      (flush output)))))

       (selector-add! selector
                      (socket-fd server)
                      accept-handler
                      '(r))
       (do () (#f) (selector-select selector))))
```

9.29 gauche.sequence - Sequence framework

gauche.sequence [Module]

Provides a generic operations on *sequences*. A sequence is a collection with ordered elements. Besides all the operations applicable on collections, you can associate integer index to each element, and apply order-aware operations on the elements.

This module inherits gauche.collection (see Section 9.5 [Collection framework], page 344). All the collection generic operations can be applied to a sequence as well.

Among Gauche builtin class, lists, vectors and strings are sequences and the specialized methods are defined for them. Other extension types, such as uniform vectors, have the methods as well.

9.29.1 Fundamental sequence accessors

ref (*seq* <*sequence*>) *index :optional fallback* [Method]

{gauche.sequence} Returns *index*-th element of the sequence *seq*. This method enables uniform access for any sequence types.

When *index* is less than zero, or greater than or equal to the size of the sequence, *fallback* is returned if provided, or an error is signaled if not.

```
(ref '(a b c) 1)  ⇒ b
(ref '#(a b c) 1) ⇒ b
(ref "abc" 1)     ⇒ #\b
```

(setter ref) (*seq* <*sequence*>) *index value* [Method]

{gauche.sequence} Sets *value* to the *index*-th element of the sequence *seq*. This is the uniform sequence modifier.

Note: Some sequences may not support arbitrary modification by index. For example, if you have a sequence representing a set of sorted integers, you cannot modify *i*-th element with arbitrary value. Yet such sequence may provide other means of modification, such as inserting or deleting elements.

```
(let ((x (list 'a 'b 'c))
```

```
    (set! (ref x 1) 'z)
    x) ⇒ (a z c)

  (let ((x (vector 'a 'b 'c)))
    (set! (ref x 1) 'z)
    x) ⇒ #(a z c)

  (let ((x (string #\a #\b #\c)))
    (set! (ref x 1) #\z)
    x) ⇒ "azc"
```

`referencer` (*seq* <*sequence*>) [Method]
{`gauche.sequence`}

`modifier` (*seq* <*sequence*>) [Method]
{`gauche.sequence`}

9.29.2 Slicing sequence

`subseq` (*seq* <*sequence*>) *:optional start end* [Method]
{`gauche.sequence`} Retrieve a subsequence of the sequence *seq*, from *start*-th element (inclusive) to *end*-th element (exclusive). If *end* is omitted, up to the end of sequence is taken. The type of the returned sequence is the same as *seq*.

```
(subseq '(a b c d e) 1 4)   ⇒ (b c d)
(subseq '#(a b c d e) 1 4)  ⇒ #(b c d)
(subseq "abcde" 1 4)        ⇒ "bcd"

(subseq '(a b c d e) 3)     ⇒ (d e)
```

`(setter subseq)` (*seq* <*sequence*>) *start end value-seq* [Method]
`(setter subseq)` (*seq* <*sequence*>) *start value-seq* [Method]
{`gauche.sequence`} Sets the elements of *value-seq* from the *start*-th element (inclusive) to the *end*-th element (exclusive) of the sequence *seq*. *Value-seq* can be any sequence, but its size must be larger than (*end* - *start*).

In the second form, *end* is figured out by the length of *value-seq*.

```
(define s (vector 'a 'b 'c 'd 'e))
(set! (subseq s 1 4) '(4 5 6))
s ⇒ #(a 4 5 6 e)
(set! (subseq s 0)   "ab")
s ⇒ #(#\a #\b 5 6 e)
```

9.29.3 Mapping over sequences

You can use extended `fold`, `map`, `for-each` and other generic functions on sequences, since a sequence is also a collection. However, sometimes you want to have index as well as the element itself during iteration. There are several generic functions for it.

`fold-with-index` *kons knil* (*seq* <*sequence*>) *...* [Method]
{`gauche.sequence`} Like generic `fold`, except *kons* is given the index within *seq*, as the first argument, as well as each element from *seq*s and the accrued value.

```
(fold-with-index acons '() '(a b c))
  ⇒ ((2 . c) (1 . b) (0 . a))
```

`map-with-index` *proc* (*seq* <*sequence*>) . . . [Method]
`map-to-with-index` *class proc* (*seq* <*sequence*>) . . . [Method]
`for-each-with-index` *proc* (*seq* <*sequence*>) . . . [Method]

{`gauche.sequence`} Like `map`, `map-to` and `for-each`, except *proc* receives the index as the first argument.

```
(map-with-index list '(a b c d) '(e f g h))
  ⇒ ((0 a e) (1 b f) (2 c g) (3 d h))

(map-to-with-index <vector> cons '(a b c d))
  ⇒ #((0 . a) (1 . b) (2 . c) (3 . d))
```

`find-with-index` *pred* (*seq* <*sequence*>) [Method]

{`gauche.sequence`} Finds the first element in *seq* that satisfies *pred* like `find`, but returns two values, the index of the element and the element itself. If no element satisfies *pred*, two `#f`'s are returned.

```
(find-with-index char-upper-case? "abraCadabra")
  ⇒ 4 and #\C

(find-with-index char-numeric? "abraCadabra")
  ⇒ #f and #f
```

`find-index` *pred* (*seq* <*sequence*>) [Method]

{`gauche.sequence`} Like `find`, but returns the index of the first element that satisfies *pred* in *seq*, instead of the element itself. If no element in *seq* satisfies *pred*, `#f` is returned.

```
(find-index char-upper-case? "abraCadabra")
  ⇒ 4

(find-index char-numeric? "abraCadabra")
  ⇒ #f
```

See also `list-index` in `scheme.list` (see Section 10.3.1 [R7RS lists], page 512).

`fold-right` *kons knil* (*seq* <*sequence*>) . . . [Method]

{`gauche.sequence`} Generalization of `fold-right` on lists. Like `fold`, this method applies a higher-order function *kons* over given sequence(s), passing the "seed" value whose default is *knil*. The difference between `fold` and `fold-right` is the associative order of elements on which *kons* is applied.

When we have one sequence, `[E0, E1, ..., En]`, `fold` and `fold-right` work as follows, respectively.

```
fold:
  (kons En (kons En-1 (kons ... (kons E1 (kons E1 knil)) ...)))

fold-right
  (kons E0 (kons E1 (kons ... (kons En-1 (kons En knil)) ...)))
```

This method isn't defined on `<collection>`, since collections don't care the order of elements.

9.29.4 Other operations over sequences

Selection and searching

`sequence-contains` *haystack needle :key test* [Generic function]

{`gauche.sequence`} Both *needle* and *haystack* must be sequences. Searches *needle* from *haystack* from the beginning of *haystack*. If *needle* is found, the index in *haystack* where it begins is returned. Otherwise `#f` is returned. The keyword argument *test* is used to compare elements; its default is `eqv?`.

```
(sequence-contains '#(a b r a c a d a b r a) '#(b r a))
  ⇒ 1

(sequence-contains '#(a b r a c a d a b r a) '#(c r a))
  ⇒ #f
```

This can be regarded as generalization of `string-contains` in srfi-13 (see Section 11.5.7 [SRFI-13 String searching], page 596).

`break-list-by-sequence` *list needle :key test* [Function]
`break-list-by-sequence!` *list needle :key test* [Function]

{`gauche.sequence`} Searches a sequence *needle* from *list*, and if found, breaks *list* to two parts—the prefix of *list* up to right before *needle* begins, and the rest—and returns them. *List* must be a list, but *needle* can be any sequence. Elements are compared by *test*, defaulted to `eqv?`.

```
(break-list-by-sequence '(a b r a c a d a b r a) '(c a d))
  ⇒ (a b r a) and (c a d a b r a)
```

If *needle* isn't found in *list*, it returns *list* itself and `()`. This behavior is aligned to `span` and `break` (see Section 10.3.1 [R7RS lists], page 512), which split a list by predicate but returns the whole list if split condition isn't met.

```
(break-list-by-sequence '(a b r a c a d a b r c a) '(c a z))
  ⇒ (a b r a c a d a b r c a) and ()
```

The linear update version `break-list-by-sequence!` modifies *list* to create the return value if necessary, so *list* must be mutable. The caller must use the return value instead of relying on side-effects, though, for *list* may not be modified.

`sequence->kmp-stepper` *needle :key test* [Function]

{`gauche.sequence`} This is an internal routine to search subsequence (*needle*) inside larger sequence, using Knuth-Morris-Pratt (KMP) algorithm. It is used in `sequence-contains`, `break-list-by-sequence` and `break-list-by-sequence!`.

Returns a procedure that performs one step of KMP search. The procedure takes two arguments, an element *elt* and an index *k*. It compares *elt* with `(~ needle k)`, and returns two values—the next index and a flag indicating the match is completed. When the match is completed, the next index is equal to the length of *needle*.

As an edge case, if *needle* is an empty sequence, `sequence->kmp-stepper` returns `#f`.

Elements are compared using *test*, which is defaulted to `eqv?`.

The following is a skeleton of searcher using `sequence->kmp-stepper`. Here we assume *haystack* is a list, and we just return whether the needle is found or not, or needle is empty; you might want to carry around other info in the loop (e.g. `sequence-contains` tracks the current index of *haystack* in order to return the found index.)

```
(if-let1 stepper (sequence->kmp-stepper needle)
  (let loop ([haystack haystack]
             [k 0])
    (if (null? haystack)
      'not-found
      (receive (k found) (stepper (car haystack) k) ; KMP step
        (if found
          'found
          (loop (cdr haystack) k)))))
  'needle-is-empty)
```

Note that selection and searching methods for collections can also be applied to sequences. See Section 9.5.2 [Selection and searching in collection], page 347.

Grouping

`group-sequence` *seq :key key test* [Generic function]

{`gauche.sequence`} Groups consecutive elements in a sequence *seq* which have the common key value. A key value of an element is obtained by applying the procedure *key* to the element; the default procedure is `identity`. For each element in *seq*, *key* is applied exactly once. The equal-ness of keys are compared by *test* procedure, whose default is `eqv?`.

```
(group-sequence '(1 1 1 2 3 4 4 2 2 3 1 1 3))
  ⇒ ((1 1 1) (2) (3) (4 4) (2 2) (3) (1 1) (3))

(group-sequence '(1 1 1 2 3 4 4 2 2 3 1 1 3)
                :key (cut modulo <> 2)))
  ⇒ ((1 1 1) (2) (3) (4 4 2 2) (3 1 1 3))

(group-sequence '#("a" "a" "b" "b" "c" "d" "d")
                :test string=?)
  ⇒ (("a" "a") ("b" "b") ("c") ("d" "d"))

(group-sequence "aabbcdd"
                :test char=?)
  ⇒ ((#\a #\a) (#\b #\b) (#\c) (#\d #\d))
```

This method is similar to Haskell's `group`. If you want to group elements that are not adjacent, use `group-collection` (see Section 9.5.2 [Selection and searching in collection], page 347).

If you simply need to reduce each group for one instance, that is, removing adjacent duplicated elements, you can use `delete-neighbor-dups` below.

`group-contiguous-sequence` *seq :key key next test squeeze* [Generic function]

{`gauche.sequence`} Group contiguous elements in *seq*.

```
(group-contiguous-sequence '(1 2 3 4 7 8 9 11 13 14 16))
  ⇒ ((1 2 3 4) (7 8 9) (11) (13 14) (16))
```

If the keyword argument *squeeze* is true, each subsequence is represented with its first and last elements, except when the subsequence has only one element.

```
(group-contiguous-sequence '(1 2 3 4 7 8 9 11 13 14 16) :squeeze #t)
  ⇒ ((1 4) (7 9) (11) (13 14) (16))
```

The keyword argument *key* must be a procedure taking one argument, and it is applied to every element in the sequence once, to construct the result. Its default is `identity`.

The keyword argument *next* must be a procedure taking one argument, which is the key value (whatever *key* procedure returns) and must return the "next" key value. Its default is `(^n (+ n 1))`.

The *test* argument must be a procedure taking two argument and used to compare two key values. Its default is `eqv?`.

```
(group-contiguous-sequence "AbCdFgH"
  :key char-upcase :next (^c (integer->char (+ 1 (char->integer c)))))
  ⇒ ((#\A #\B #\C #\D) (#\F #\G #\H))
```

`delete-neighbor-dups` *seq :key key test start end* [Generic function]

{`gauche.sequence`} Returns a sequence of the same type as *seq*, in which elements in *seq* are included in the original order, except duplicate adjacent elements. The type of *seq* must has a builder.

```
(delete-neighbor-dups '(1 1 1 2 3 4 4 2 2 3 1 1 3))
  ⇒ (1 2 3 4 2 3 1 3)

(delete-neighbor-dups '#(1 1 1 2 3 4 4 2 2 3 1 1 3))
  ⇒ #(1 2 3 4 2 3 1 3)

(delete-neighbor-dups "1112344223113")
  ⇒ "12342313"
```

Elements are compared with `eqv?` by default. You can pass alternative procedure to *test* keyword argument; it is always called as `(test x y)`, where *x* and *y* are the contiguous elements in *seq*. If elements are compared equal, the first one is kept:

```
(delete-neighbor-dups "AaaAbBBbCCcc" :test char-ci=?)
  ⇒ "AbC"
```

If *key* is provided, it must be a procedure that takes one arguments. It is applied to each element of *seq* at most once, and each resulting value is used for the comparison instead of elements themselves.

```
(delete-neighbor-dups
  '((1 . "a") (1 . "b") (2 . "c") (2 . "d"))
  :key car)
  ⇒ ((1 . "a") (2 . "c"))
```

The *start* and *end* arguments specify indexes in the *seq* to limit the range to look at. Where *start* is inclusive, *end* is exclusive.

```
(delete-neighbor-dups "1112344223113" :start 3 :end 9)
  ⇒ "2342"
```

`delete-neighbor-dups!` *seq :key key test start end* [Generic function]

{`gauche.sequence`} Scan *seq* from left to right, dropping consecutive duplicated elements. The result is stored into *seq*, packed to left. Note *seq* must be modifiable by index, i.e. `modifier` method must be defined. The rest of *seq* will be untouched. Returns the next index after the last modified entry.

```
(let1 v (vector 1 1 2 2 3 3 2 2 4 4)
  (list (delete-neighbor-dups! v)
        v))
  ⇒ (5 #(1 2 3 2 4 3 2 2 4 4))
```

The semantics of keyword arguments *key*, *test*, *start* and *end* are the same as `delete-neighbor-dups`.

```
(let1 v (vector 1 1 2 2 3 3 2 2 4 4)
  (list (delete-neighbor-dups! v :start 2)
        v))
  ⇒ (6 #(1 1 2 3 2 4 2 2 4 4))
```

Note: This method works on any sequence with `modifier` method, but it's not necessarily more efficient than `delete-neighbor-dups`, which creates a new sequence. If *seq* is a list or a string, each modification by index takes O(n) time (for a string even it costs O(n) extra storage), so the total cost is O(n^2), whereas `delete-neighbor-dups` needs O(n) time and storage. This works best for vectors and alike, with which it doesn't cost extra allocation and runs in O(n) time.

`delete-neighbor-dups-squeeze!` *seq :key key test start end* [Generic function]

{`gauche.sequence`} Operates like `delete-neighbor-dups` but reuses storage of *seq*, which will be resized by dropping duplicated elements. Returns the sequence after dupes are removed.

Not all sequences are resizable, so this method won't be defined for such sequences. The `gauche.sequence` module only provides this method for `<list>`, in which dropping the middle of the sequence is very efficient as it is just a single `set-cdr!`.

```
(delete-neighbor-dups-squeeze! (list 1 1 1 2 2 2 3 3 3))
  ⇒ (1 2 3)

(delete-neighbor-dups-squeeze! (list 1 1 1 2 2 2 3 3 3 4 4)
                                :start 3 :end 9)
  ⇒ (2 3)
```

The semantics of keyword arguments *key*, *test*, *start* and *end* are the same as `delete-neighbor-dups`.

Prefix

`common-prefix` (*a* <*sequence*>) (*b* <*sequence*>) *:key key test* [Generic function]

{`gauche.sequence`} Returns a new sequence of the same type of *a* which contains the common prefix of sequences *a* and *b*. The types of *a* and *b* doesn't need to match. The type of *a* must have a builder.

For each corresponding element in *a* and *b*, the *key* procedure is applied (default `identity`), then compared with *test* procedure (default `eqv?`).

```
(common-prefix '(a b c d e) '(a b c e f))
  ⇒ (a b c)

(common-prefix "abcef" '#(#\a #\b #\c #\d #\e))
  ⇒ "abc"
```

For strings, `srfi-13` has a specific function with related feature: `string-prefix-length` (see Section 11.5.6 [SRFI-13 String Prefixes & Suffixes], page 595).

`common-prefix-to` (*class* <*class*>) (*a* <*sequence*>) (*b* <*sequence*>) *:key key test* [Generic function]

{`gauche.sequence`} Returns a new sequence of the type *class* which contains the common prefix of sequences *a* and *b*. The types of *a* and *b* doesn't need to match, and neither needs to have a builder. The *class* must be a sequence class with a builder.

The meanings of keyword arguments are the same as `common-prefix`.

```
(common-prefix-to <list> "abcde" "ABCEF" :test char-ci=?)
  ⇒ '(#\a #\b #\c)
```

Permutation and shuffling

`permute` (*src* <*sequence*>) (*permuter* <*sequence*>) *:optional fallback* [Generic function]

{`gauche.sequence`} Returns a newly created sequence of the same type as *src*, in which the elements are permuted from *src* according to *permuter*.

Permuter is a sequence of exact integers. When the *k*-th element of *permuter* is *i*, the *k*-th element of the result is `(ref src i)`. Therefore, the size of the result sequence is the same as the size of *permuter*. *Permuter* can be any kind of sequence, unrelated to the type of *src*.

It is allowed that the same index *i* can appear more than once in *permuter*.

```
(permute '(a b c d) '(3 2 0 1))      ⇒ (d c a b)
```

```
(permute '(a b c d) '(0 2))         ⇒ (a c)
(permute '(a b c d) '(0 0 1 1 2 2)) ⇒ (a a b b c c)
```

If an integer in *permuter* is out of the valid range as the index of *src*, then an error is signaled unless *fallback* is given. If *fallback* is given, what value is used depends on the result of `(ref src i fallback)`—which usually returns *fallback* for the out-of-range index *i*.

```
(permute '#(a b c) '(3 2 1 0) 'foo) ⇒ #(foo c b a)

(permute "!,HWdelor" #(2 5 6 6 7 1 -1 3 7 8 6 4 0) #\space)
  ⇒ "Hello, World!"
```

`permute-to` (*class* <*class*>) (*src* <*sequence*>) (*permuter* <*sequence*>) *:optional fallback* [Generic function]

{`gauche.sequence`} Like `permute`, but the result will be an instance of the given *class* instead of the class of *src*.

```
(permute-to <string> '(#\a #\b #\c #\d #\r)
             '(0 1 4 0 2 0 3 0 1 4 0))
  ⇒ "abracadabra"
```

`permute!` (*src* <*sequence*>) (*permuter* <*sequence*>) *:optional fallback* [Generic function]

{`gauche.sequence`} Also like `permute`, but the result is stored back to *src*. *Src* must be a mutable sequence, and the length of *src* and *permuter* must be the same.

`shuffle` (*src* <*sequence*>) *:optional random-source* [Generic function]

{`gauche.sequence`} Returns a new sequence of the same type and size as *src*, in which elements are randomly permuted.

```
(shuffle '(a b c d e))  ⇒ (e b d c a)
(shuffle "abcde")       ⇒ "bacde"
```

This generic function uses `srfi-27` (see Section 11.8 [Sources of random bits], page 605). By default it uses `default-random-source`, but you can pass an alternative random source by the optional argument.

`shuffle-to` (*class* <*class*>) (*src* <*sequence*>) *:optional random-source* [Generic function]

{`gauche.sequence`} Like `shuffle`, except that the result will be an instance of *class* instead of the class of *src*.

`shuffle!` (*src* <*sequence*>) *:optional random-source* [Generic function]

{`gauche.sequence`} Like `shuffle`, but the result is stored back to *src*. *Src* must be a mutable sequence.

9.29.5 Implementing sequence

9.30 `gauche.syslog` - Syslog

`gauche.syslog` [Module]

This module provides syslog(3) system logger interface.

For the common applications, you might find `gauche.logger` module easier to use (see Section 9.16 [User-level logging], page 393). This module is for those who need direct access to the syslog API.

The procedures are only defined if the underlying system supports them.

`sys-openlog` *ident option facility* [Function]

[POSIX] {`gauche.syslog`} Opens a connection to the system logger. A string argument *ident* is used for the prefix of the log, and usually is the program name. *Option* is an integer flag to control the behavior of logging, and *facility* is an integer that specify the type of the program.

The flag for *option* can be composed by `logior`-ing one or more of the following integer constants: `LOG_CONS`, `LOG_NDELAY`, `LOG_NOWAIT`, `LOG_ODELAY`, `LOG_PERROR` and `LOG_PID`. (Some of the constants may not be defined if the underlying system doesn't support them).

The *facility* argument can be one of the following integer constants: `LOG_AUTH`, `LOG_AUTHPRIV`, `LOG_CRON`, `LOG_DAEMON`, `LOG_FTP`, `LOG_KERN`, `LOG_LOCAL0` through `LOG_LOCAL7`, `LOG_LPR`, `LOG_MAIL`, `LOG_NEWS`, `LOG_SYSLOG`, `LOG_USER` and `LOG_UUCP`. (Some of the constants may not be defined if the underlying system doesn't support them).

See your system's manpage of openlog(3) for detail description about these constants.

`sys-syslog` *priority message* [Function]

[POSIX] {`gauche.syslog`} Log the string *message*. Unlike syslog(3), this procedure doesn't do formatting—you can use `format` (see Section 6.22.8 [Output], page 231) to create a formatted message, or use higher-level routine `log-format` (see Section 9.16 [User-level logging], page 393).

An integer argument *priority* can be composed by `logior`-ing one of the *facility* constants described above and the *level* constants: `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

`sys-closelog` [Function]

[POSIX] {`gauche.syslog`} Closes the connection to the logging system.

`sys-setlogmask` *mask* [Function]

[POSIX] {`gauche.syslog`} Sets the process's log priority mask that determines which calls to `sys-syslog` may be logged. An priority *mask* can be composed by `logior`-ing bitmasks corresponding to the *level* argument of `sys-syslog`. You can use `sys-logmask` below to obtain a bitmask from the level.

`sys-logmask` *level* [Function]

[POSIX] {`gauche.syslog`} Returns an integer bitmask for `sys-setlogmask` from the log level *level*.

9.31 `gauche.termios` - Terminal control

`gauche.termios` [Module]

This module provides procedures to control terminals. On Unix platforms, the low-level API provides POSIX termios interface as the module name suggests. This module also provides pseudo tty interface, if the system supports it.

On Windows native platforms, POSIX termios interface is not available. It is too different from Windows console API to provide a meaningful emulation. The low-level Windows console API is available in the `os.windows` module (see Section 12.28 [Windows support], page 708). You can still use high-level terminal control procedures in this module.

9.31.1 Posix termios interface

These procedures are available when the feature identifier `gauche.os.windows` is *not* defined. See `cond-expand` in Section 4.12 [Feature conditional], page 68, for how to switch code using feature identifiers.

`<sys-termios>` [Builtin Class]

{`gauche.termios`} POSIX termios(7) structure.

`iflag` [Instance Variable of `<sys-termios>`]

`oflag` [Instance Variable of `<sys-termios>`]

`cflag` [Instance Variable of `<sys-termios>`]

`lflag` [Instance Variable of `<sys-termios>`]

`cc` [Instance Variable of `<sys-termios>`]

The slots `iflag`, `oflag`, `cflag` and `lflag` contains non-negative integers representing bitmasks.

The slot `cc` contains a *copy* of `c_cc` array of `struct termios`, as an u8vector (see Section 9.36 [Uniform vectors], page 476, for the details about u8vector). Since `cc` slot is a copy of the internal structure, you have to `set!` an u8vector to the slot explicitly to make changes to the `c_cc` array.

Throughout this section, argument *port-or-fd* refers to either a port object or a small integer representing system's file descriptor. If *port* is not associated to the system terminal, an error is signaled. (You can check if *port* has an associated terminal by `sys-isatty?`. see Section 6.25.4.5 [Other file operations], page 257).

`sys-tcgetattr` *port-or-fd* [Function]

{`gauche.termios`} Returns terminal parameters in a `<sys-termios>` object, associated to *port-or-fd*.

`sys-tcsetattr` *port-or-fd when termios* [Function]

{`gauche.termios`} Sets terminal parameters associated to *port-or-fd* by *termios*, which must be an instance of `<sys-termios>`.

An integer argument *when* specifies when the changes take effect. Three variables are predefined for the argument:

`TCSANOW` The change is reflected immediately.

`TCSADRAIN`
The change is reflected after all pending output is flushed.

`TCSAFLUSH`
The change is reflected after all pending output is flushed, and all pending input is discarded.

`sys-tcsendbreak` *port-or-fd duration* [Function]

{`gauche.termios`} Transmits a zero stream for the specified duration to the terminal associated to *port-or-fd*. The unit of duration depends on the system; see man tcsendbreak(3) of your system for details.

`sys-tcdrain` *port-or-fd* [Function]

{`gauche.termios`} Waits until all output written to *port-or-fd* is transmitted.

`sys-tcflush` *port-or-fd queue* [Function]

{`gauche.termios`} Discards data in the buffer of *port-or-fd*, specified by *queue*, which may be one of the following values.

`TCIFLUSH` Discards data received but not read.

`TCOFLUSH` Discards data written but not transmitted.

`TCIOFLUSH`
Do both `TCIFLUSH` and `TCOFLUSH` action.

`sys-tcflow` *port-or-fd action* [Function]

{`gauche.termios`} Controls data flow of *port-or-fd* by *action*, which may be one of the following values:

`TCOOFF` Suspends output transmission.

`TCOON` Restarts output transmission.

`TCIOFF` Transmits a STOP character to make the terminal device stop transmitting data to the system.

`TCION` Transmits a START character to make the terminal device resume transmitting data to the system.

`sys-tcgetpgrp` *port-or-fd* [Function]

{`gauche.termios`} Returns process group ID of the terminal associated to *port-or-fd*.

`sys-tcsetpgrp` *port-or-fd pgrp* [Function]

{`gauche.termios`} Sets process group ID of the terminal associated to *port-or-fd* to *pgrp*.

`sys-cfgetispeed` *termios* [Function]
`sys-cfsetispeed` *termios speed* [Function]
`sys-cfgetospeed` *termios* [Function]
`sys-cfsetospeed` *termios speed* [Function]

{`gauche.termios`} Gets/sets input/output speed (baud rate) parameter stored in *termios* object. Speed is represented by the following predefined numbers: `B0`, `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, `B38400`.

Some system may support higher baud rate, such as `B57600`, `B115200` or `B230400`. You can use `symbol-bound?` to check these options are defined. `B0` is used to terminate the connection.

`sys-openpty` *:optional term* [Function]

{`gauche.termios`} Opens a pair of pseudo ttys, one for master and the other for slave, then returns two integers which are their file descriptors. An optional argument *term* must be, if passed, a `<sys-termios>` object; it sets the slave pty's parameters.

You can use `open-input-fd-port` and/or `open-output-fd-port` to create a port around the returned file descriptor (see Section 6.22.4 [File ports], page 221). To obtain pseudo tty's name, use `sys-ttyname` (see Section 6.25.4.5 [Other file operations], page 257).

This function is available only if the system supports `openpty(3)`.

`sys-forkpty` *:optional term* [Function]

{`gauche.termios`} Opens a pair of pseudo ttys, one for master and the other for slave, sets the slave pty suitable for login terminal, then fork(2).

Returns two integers; the first value is a child pid for the parent process, and 0 for the child process. The second value is a file descriptor of the master pty.

An optional argument *term* must be, if passed, a `<sys-termios>` object; it sets the slave pty's parameters.

This function is available only if the system supports `forkpty(3)`.

Note: `sys-forkpty` has the same MT hazard as `sys-fork` (see Section 6.25.10 [Process management], page 271, for details). If you're running multiple threads, use `sys-forkpty-and-exec` below.

`sys-forkpty-and-exec` *command args :key iomap term sigmask* [Function]

{`gauche.termios`} Does `sys-forkpty`, and lets the child process immediately `exec`s the specified *command* with arguments *args*. This function doesn't have the hazard in multi-thread environment.

The meanings of arguments *command*, *args*, *iomap* and *sigmask* are the same as `sys-exec` (see Section 6.25.10 [Process management], page 271). If the keyword argument *term* is given, it is used to initialize the slave pty.

9.31.2 Common high-level terminal control

`without-echoing` *iport proc* [Function]

{`gauche.termios`} If *iport* is an input port connected to a terminal, sets the terminal mode non-echoing and call *proc* with *iport* as an argument. Before returning from `without-echoing`, or throwing an error, the terminal mode is reset to the original state when this procedure is called. The procedure returns whatever value(s) *proc* returns.

You can also pass `#f` to *iport*. In that case, this procedure tries to open a console (`/dev/tty` on Unix, `CON` on Windows) and set the console mode, then calls *proc* with the opened input port. An error is thrown if the procedure can not open a console.

If *iport* is other than above, this procedure simply calls *proc* with *iport*. This allows the caller to read password from redirected input, for example.

Note: Because of an implementation issue, on Windows native platforms this procedure always changes console mode of the standard input handle when `iport` is either `#f` or a terminal input port.

`has-windows-console?` [Function]

{`gauche.termios`} Returns `#t` iff the running Gauche is Windows-native and the process has attached console. On POSIX platforms this procedure always returns `#f`.

The reason that `cond-expand` isn't enough is that on Windows the program may start without console, but you can attach console afterwards. See Section 12.28.2 [Windows console API], page 709, for the details.

9.32 `gauche.test` - Unit Testing

`gauche.test` [Module]

Defines a set of functions to write test scripts. A test script will look like this:

```
(use gauche.test)
(test-start "my feature")
(load "my-feature")   ; load your program
(import my-feature)   ; if your program defines a module.

(test-module 'my-feature) ; tests consistency in your module.

(test-section "feature group 1")
(test "feature 1-1" EXPECT (lambda () TEST-BODY))
(test "feature 1-2" EXPECT (lambda () TEST-BODY))
 ...

(test-section "feature group 2")
(define test-data ...)
(test "feature 2-1" EXPECT (lambda () TEST-BODY))
(test "feature 2-2" (test-error) (lambda () TEST-THAT-SIGNALS-ERROR))
 ...

(test-end :exit-on-failure #t)
```

With this convention, you can run test both interactively or in batch. To run a test interactively, just load the file and it reports a result of each test, as well as the summary of failed test at the end. To run a test in batch, it is convenient to redirect the stdout to some file If stdout is redirected to other than tty, all the verbose logs will go there, and only a small amount of messages go to stderr.

It is recommended to have a "check" target always in Makefile of your module/program, so that the user of your program can run a test easily. The rule may look like this:

```
check :
        gosh my-feature-test.scm > test.log
```

Structuring a test file

`test-start` *module-name* [Function]

{`gauche.test`} Initializes internal state and prints a log header. This should be called before any tests. *Module-name* is used only for logging purpose.

`test-section` *section-name* [Function]

{`gauche.test`} Marks beginning of the group of tests. This is just for logging.

`test-log` *fmtstr args . . .* [Function]

{`gauche.test`} This is also just for logging. Creates a formatted string with *fmrstr* and *args* just like `format`, then write it to the current output port, with prefix `;;` and newline at the end.

With the typical Makefile settings, where you redirect stdout of test scripts to a log file, the message only goes to the log file.

Using this, you can dump information that can't be automatically tested but may be useful for troubleshooting. For example, you get a mysterious test failure reports you can't reproduce on your machine, and suspect some aspects of the running systems may unpredictably affect the test result. You can put `test-log` in the test code to dump such parameters, and ask the reporter to run the test again and analyze the log.

`test-end` *:key exit-on-failure* [Function]

{`gauche.test`} Prints out list of failed tests. If *exit-on-failure* is `#f` or omitted, this procedure returns the number of failed tests.

Otherwise, this function terminates the `gosh` process by `exit`. If a fixnum is given to `exit-on-failure` it becomes the process's exit status; if other true value is given, the exit status will be 1.

`test-record-file` *file* [Function]

{`gauche.test`} Suppose you have several test scripts. Normally you run them as a group and what you want to know is a concise summary of the whole results, instead of each result of individual test files.

A *test record file* is an auxiliary file used to gather summary of the result. It holds a one-line summary of tests like this:

```
Total:  9939 tests,  9939 passed,     0 failed,     0 aborted.
```

When a test record file exists, `test-start` reads and parses it, and remembers the numbers. Then `test-end` adds the count of the results and writes them back to the same test record file.

If you writes the `check` target in your makefile as follows, you will get the final one-line summary every time you run `make check`, assuming that `test1.scm`, `test2.scm`, and `test3.scm` all has `(test-record-file "test.record")` before a call to `test-start`.

```
check:
        @rm -f test.record test.log
```

```
gosh test1.scm >> test.log
gosh test2.scm >> test.log
gosh test3.scm >> test.log
@cat test.record
```

Note that to make `test-record-file` work, it must be placed before the call to `test-start`.

Alternatively, you can use the environment variable `GAUCHE_TEST_RECORD_FILE` to specify the test record file.

`GAUCHE_TEST_RECORD_FILE` [Environment Variable]

If this environment variable is set when the test script is run, its value is used as the name of the test record file.

If the test script calls `test-record-file`, it takes precedence and this environment variable is ignored.

`test-summary-check` [Function]

{`gauche.test`} If the test record file is set (either by `test-record-file` or the environment variable `GAUCHE_TEST_RECORD_FILE`), read it, and then exit with status 1 if the record has nonzero failure count and/or nonzero abort count. If the test record file isn't set, this procedure does nothing.

This is useful when you have multiple test scripts and you want to let `make` fail if any of tests fails, but not before all test script is run. If you make every test script use `:exit-on-failure` of `test-end`, then `make` stops immediately after the script that fails. Instead, you avoid using `:exit-on-failure`, but use the test record file and for the last thing you can call this function:

```
check:
   rm -f $GAUCHE_TEST_RECORD_FILE test.log
   gosh test1.scm >> test.log
   gosh test2.scm >> test.log
   cat $GAUCHE_TEST_RECORD_FILE /dev/null
   gosh -ugauche.test -Etest-summary-check -Eexit
```

By this, `make` will run all the test script no matter how many of them fails (since `gosh` exits with status 0), but detect an error since the last line of `gosh` call exits with status 1 if there has been any failure.

Individual tests

`test*` *name expected expr :optional check* [Macro]

{`gauche.test`} A convenience macro that wraps *expr* by lambda.

```
(test* name expected expr)
  ≡ (test name expected (lambda () expr))
```

`test` *name expected thunk :optional check* [Function]

{`gauche.test`} Calls *thunk*, and checks its result fits *expected* using a procedure *check*, which is called as follows:

```
(check expected result-of-thunk)
```

It should return `#t` if the given result agrees with the expected value, or `#f` otherwise. The default check procedure is `test-check`, explained below. It compares *expected* and *result-of-thunk* with `equal?`, except when *expected* is some of special case test objects. (See "testing ambiguous results" and "testing abnormal cases" paragraphs below for this special treatment.)

One typical usage of the custom check procedure is to compare inexact numbers tolerating small error.

```
(test "test 1" (/ 3.141592653589 4)
      (lambda () (atan 1))
      (lambda (expected result)
        (< (abs (- expected result)) 1.0e-10)))
```

Name is a name of the test, for the logging purpose.

When *thunk* signals an uncaptured error, it is caught and yields a special error object `<test-error>`. You can check it with another error object created by `test-error` function to see if it is an expected type of error. See the entry of `test-error` below for the details.

`test-check` *expected result :optional fallback* [Function]

{`gauche.test`} The default procedure `test` and `test*` use to check the result of the test expression conforms the expected value. By default, `test-check` just compares *expected* and *result* with a procedure *fallback*, which is defaulted to `equal?`. `test-check` behaves differently if *expected* is one of special test objects described below.

Testing ambiguous results

`test-one-of` *choice ...* [Function]

{`gauche.test`} Sometimes the result of test expression depends on various external environment, and you cannot put an exact expected value. This procedure supports to write such tests conveniently.

Returns a special object representing *either one of the choices*. The default check procedure, `test-check`, recognizes the object when it is passed in the *expected* argument, and returns true if any one of *choice* ... passes the check against the result.

For example, the following test passes if `proc` returns either 1 or 2.

```
(test* "proc returns either 1 or 2" (test-one-of 1 2) (proc))
```

`test-none-of` *choice ...* [Function]

{`gauche.test`} Similar to `test-one-of`, but creates a special object representing *none of the choices*. The test suceeds if the test expression evaluates to a value that don't match any of *choice*s.

Testing abnormal cases

`test-error` *:optional* (*condition-type* <*error*>) (*message #f*) [Function]

{`gauche.test`} Returns a new `<test-error>` object that matches with other `<test-error>` object with the given *condition-type*.

The `test-check` procedure treats `<test-error>` objects specially. When `err-expected` and `err-actual` are `<test-error>` objects, `(test-check err-expected err-actual)` returns `#t` if `err-expected`'s condition type is the same as or supertype of `err-actual`'s.

For example, if you want to test a call to `foo` raises an `<io-error>` (or its condition subtype), you can write as the following example:

```
(test "see if foo raises <io-error>" (test-error <io-error>) (foo))
```

Another optional argument *message* can be used to check if the raised error has a message of expected pattern. The argument may be a string, a regexp or `#f` (default). If it is a string, `test-check` checks if the message of the raised error exactly match the string. If it is a regexp, `test-check` checks the message of the raised error matches that regexp. If it is `#f`, the message is not checked.

`*test-error*` [Variable]

{`gauche.test`} (Deprecated) Bounded to an instance of `<test-error>` with condition type `<error>`. This is only provided for the backward compatibility; new code should use `test-error` procedure above.

`*test-report-error*` [Variable]

{`gauche.test`} If this variable is true, the `test` routine prints stack trace to the current error port when it captures an error. It is useful when you got an unexpected test-error object and want to check out where the error is occurring.

This variable is initialized by the environment variable `GAUCHE_TEST_REPORT_ERROR` when the `gauche.test` module is loaded. For example, you can use the environment variable to check out an unexpected error from your test script as follows (the value of the environment variable doesn't matter).

```
env GAUCHE_TEST_REPORT_ERROR=1 gosh mytest.scm
```

Quasi-static checks

Scheme is dynamically typed, which is convenient for incremental and experimental development on REPL, but it tends to delay error detection until the code is actually run. It is very annoying that you run your program for a while only to see it barfs on simple typo of variable name.

Gauche addresses this issue by checking certain types of errors at the test phase. It isn't purely a static check (we need to load a module or a script, which evaluates toplevel expressions), nor exhaustive (we can't catch inconsistencies that span over multiple modules or about information that can be added at runtime). Nevertheless it can often catch some common mistakes, such as incorrect variable names or calling procedures with wrong number of arguments.

The two procedures, `test-module` and `test-script`, load the named module and the script files respectively (which compiles the Scheme code to VM instructions), then scan the compiled VM code to perform the following tests:

1. See if the global variables referenced within functions are all defined (either in the module, or in one of imported modules).
2. If a global variable is used as a function, see if the number of arguments given to it is consistent to the actual function.
3. See if the symbols set as autoload in the code can be resolved.
4. While testing module, see if the symbols declared in the export list are actually defined.

The check is somewhat heuristic and we may miss some errors and/or can have false positives. For false positives, you can enumerate symbols to be excluded from the test.

`test-module` *module :key allow-undefined bypass-arity-check* [Function]

{`gauche.test`} Loads the module and runs the quasi-static consistency check. *Module* must be a symbol module name or a module.

Sometimes you have a global variable that may not be defined depending on compiler options or platforms, and you check its existence at runtime before using it. The undefined variable reference check by `test-module` doesn't follow such logic, and reports an error whenever it finds your code referring to undefined variable. In such case, you can give a list of symbols to the *allow-undefined* keyword argument; the test will excludes them from the check.

The arity check may also raise false positives, if the module count on a behavior of global procedures that will be modified after the module is loaded (e.g. a method with different number of arguments can be added to a generic function after the module is loaded, which would make the code valid.) If you know you're doing right thing and need to suppress the false positives, pass a list of names of the functions to `bypass-arity-check` keyword arguments.

`test-script` *filename :key allow-undefined bypass-arity-check compile-only* [Function]

{`gauche.test`} Loads the script named by *filename* into a fresh anonymous module and runs the quasi-static consistency check. *Filename* must be a string that names the script file.

The meaning of keyword arguments is the same as `test-module`.

Note that the toplevel forms in *filename* are evaluated, so scripts that relies on the actions of toplevel forms could cause unwanted side-effects. This check works best for the scripts written in `srfi-22` convention, that is, calling actions from `main` procedure instead of toplevel forms. R7RS scripts relies on actions in toplevel forms and can't be tested with this procedure.

Scripts that relies on being loaded into `user` module also won't work well with this check, which loads the forms into anonymous module.

If you need to test a script with toplevel side-effecting forms and you can't change it, you may want to pass true value to the `compile-only` keyword argument. Then `test-script` just compiles each toplevel form before running static checking, instead of `load`ing (which not only compiles but executes each of toplevel forms).

9.33 `gauche.threads` - Threads

If enabled at compilation time, Gauche can use threads built on top of either POSIX threads (pthreads) or Windows threads.

`gauche.threads` [Module]

Provides thread API. You can 'use' this module regardless whether the thread support is compiled in or not; if threads are not supported, many thread-related procedures simply signals a "not supported" error.

If you want to switch code depending on whether pthreads are available or not, you can use a feature identifier `gauche.sys.threads` with `cond-expand` form (see Section 4.12 [Feature conditional], page 68).

```
(cond-expand
 [gauche.sys.threads
   ;; Code that uses thread API (gauche.threads is automatically
   ;; loaded at this moment).
  ]
 [else
   ;; Code that doesn't use thread API
  ])
```

There are also feature identifiers `gauche.sys.pthreads` and `gauche.sys.wthreads` defined for pthreads and Windows threads platforms, respectively. In Scheme level, however, you hardly need to distinguish the underlying implementations. It is recommended to use `gauche.sys.threads` to switch the code according to thread availability.

To check if threads are available at runtime, instead of compile time, use the following procedure.

`gauche-thread-type` [Function]

{`gauche.threads`} Returns a symbol that indicates the supported thread type. It can be one of the following symbols.

`none` Threads are not supported.

`pthread` Threads are built on top of POSIX pthreads.

`win32` Threads are built on top of Win32 threads.

(Note: On pthreads platforms, it should return `pthreads` instead of `pthread`; then the returned symbol would correspond to the value given to `--enable-threads` option at configuration time. It's a historical overlook, stuck for the backward compatibility.)

Scheme-level thread API conforms SRFI-18, "Multithreading support" ([[SRFI-18]], page 830), wrapped around Gauche's object interface.

9.33.1 Thread programming tips

What's Gauche threads for

Although the surface API of threads looks simple and portable, you need to know how the threads are implemented in order to utilize the feature's potential. Some languages support threads as language's built-in construct and encourage programmers to express the calculation in terms of threads. However, it should be noted that in many cases there are alternative ways than threads to implement the desired algorithm, and you need to compare advantages and disadvantages of using threads depending on how the threads are realized in the underlying system.

In Gauche, the primary purpose of threads is to write programs that *require* preemptive scheduling, therefore are difficult to express in other ways. Preemptive threads may be required, for example, when you have to call a module that does blocking I/O which you can't intercept, or may spend nondeterministic amount of calculation time that you want to interrupt.

For each Gauche's thread, an individual VM is allocated and it is run by the dedicated POSIX thread. Thus the cost of context switch is the same as the native thread, but the creation of threads costs much higher than, say, lightweight threads built on top of call/cc. So Gauche's preemptive threads are *not* designed for applications that want to create thousands of threads for fine-grained calculation.

The recommended usage is the technique so called "thread pool", that you create a set of threads and keep them around for long time and dispatch jobs to them as needed. Gauche provides a thread pool implementation in `control.thread-pool` module (see Section 12.6 [Thread pools], page 646).

Preemptive threads have other difficulties (e.g. see [FairThreads], page 829), and sometimes the alternatives may be a better fit than the native preemptive threads.

- If what you need is just a concurrent calculation, you might be able to use cooperative thread technique built on top of `call/cc`. Creating call/cc-based threads is much faster than creating native threads.
- If what you need is to deal with blocking I/O, and you have all your code at hand, it is sometimes easier to use good old `select`-based dispatching (See Section 9.28 [Simple dispatcher], page 440, for example).
- If what you need is to control the resource consumption in the subsystem, and the subsystem works fairly independently from the main system, you may be able to use Unix processes instead of threads. It may sound to go backward, but Unix process does provide higher "shield" between the subsystem and the main system (e.g. the main system can keep running even if subsystem segfaults).

Of course, these technique are not mutually exclusive with native threads. You can use dispatcher with "thread pool" technique, for example. Just keep it in your mind that the native threads are not only but one of the ways to realize those features.

Uncaught errors in a thread body

When you run a single-thread program that raises an unexpected (unhandled) error, Gauche prints an error message and a stack trace by default. So sometimes it perplexes programmers when a thread doesn't print anything when it dies because of an unhandled error.

What's happening is this: An unhandled error in a thread body would cause the thread to terminate, and the error itself will propagate to the thread who's expecting the result of the terminated thread. So, you get the error (wrapped by `<uncaught-exception>`) when you call `thread-join!` on a thread which is terminated because of an unhandled error. The behavior is defined in SRFI-18.

If you fire a thread for one-shot calculation, expecting to receive the result by `thread-join!`, then this is reasonable—you can handle the error situation in the "parent" thread. However, if you run a thread to loop indefinitely to process something and not expect to retrieve its result via `thread-join!`, this becomes a pitfall; the thread may die unexpectedly but you wouldn't know it. (If such a thread is garbage-collected, a warning is printed. However you wouldn't know when that happens so you can't count on it.)

For such threads, you should always wrap the body of such thread with `guard`, and handles the error explicitly. You can call `report-error` to display the default error message and a stack trace.

```
(thread-start!
 (make-thread (^[] (guard (e [else (report-error e) #f])
                     ... thread body ...))))
```

See Section 9.33.4 [Thread exceptions], page 467, for the details of thread exception handling.

Note: As of 0.9.5, Gauche has a known bug that the tail call of error handling clauses of `guard` doesn't become a proper tail call. So, the following code, which should run safely in Scheme, could eat up a stack:

```
(thread-start!
 (make-thread (^[] (let loop ()
                     (guard (e [else (report-error e) (loop)])
                       ... thread body ...)))))
```

For the time being, you can lift the call to loop outside of `guard` as workaround.

```
(thread-start!
 (make-thread (^[] (let loop ()
                     (guard (e [else (report-error e)])
                       ... thread body ...)
                     (loop)))))
```

9.33.2 Thread procedures

`<thread>` [Builtin Class]

A thread. Each thread has an associated thunk which is evaluated by a POSIX thread. When thunk returns normally, the result is stored in the internal 'result' slot, and can be retrieved by `thread-join!`. When thunk terminates abnormally, either by raising an exception or terminated by `thread-terminate!`, the exception condition is stored in their internal 'result exception' slot, and will be passed to the thread calling `thread-join!` on the terminated thread.

Each thread has its own dynamic environment and dynamic handler stack. When a thread is created, its dynamic environment is initialized by the creator's dynamic environment. The thread's dynamic handler stack is initially empty.

A thread is in one of the following four states at a time. You can query the thread state by the `thread-state` procedure.

`new` A thread hasn't started yet. A thread returned from `make-thread` is in this state. Once a thread is started it will never be in this state again. At this point, no POSIX thread has been created; `thread-start!` creates a POSIX thread to run the Gauche thread.

`runnable` When a thread is started by `thread-start!`, it becomes to this state. Note that a thread blocked by a system call is still in `runnable` state.

`stopped` A thread becomes in this state when it is stopped by `thread-stop!`. A thread in this state can go back to `runnable` state by `thread-cont!`, resuming execution from the point when it is stopped.

`terminated`
When the thread finished executing associated code, or is terminated by `thread-terminate!`, it becomes in this state. Once a thread is in this state, the state can no longer be changed.

Access to the resources shared by multiple threads must be protected explicitly by synchronization primitives. See Section 9.33.3 [Synchronization primitives], page 462.

Access to ports are serialized by Gauche. If multiple threads attempt to write to a port, their output may be interleaved but no output will be lost, and the state of the port is kept consistent. If multiple threads attempt to read from a port, a single read primitive (e.g. `read`, `read-char` or `read-line`) works atomically.

Signal handlers are shared by all threads, but each thread has its own signal mask. See Section 6.25.7.5 [Signals and threads], page 265, for details.

A thread object has the following external slots.

`name` [Instance Variable of `<thread>`]
A name can be associated to a thread. This is just for the convenience of the application. The primordial thread has the name `"root"`.

`specific` [Instance Variable of `<thread>`]
A thread-local slot for use of the application.

`current-thread` [Function]
[SRFI-18], [SRFI-21] {`gauche.threads`} Returns the current thread.

`thread?` *obj* [Function]
[SRFI-18], [SRFI-21] {`gauche.threads`} Returns `#t` if *obj* is a thread, `#f` otherwise.

`make-thread` *thunk :optional name* [Function]
[SRFI-18], [SRFI-21] {`gauche.threads`} Creates and returns a new thread to execute *thunk*. To run the thread, you need to call `thread-start!`. The result of *thunk* may be retrieved by calling *thread-join!*.

You can provide the name of the thread by the optional argument *name*.

The created thread inherits the signal mask of the calling thread (see Section 6.25.7.5 [Signals and threads], page 265), and has a copy of parameters of the calling thread at the time of creation (see Section 9.22 [Parameters], page 411).

Other than those initial setups, there will be no relationship between the new thread and the calling thread; there's no parent-child relationship like Unix process. Any thread can call `thread-join!` on any other thread to receive the result. If nobody issues `thread-join!` and nobody holds a reference to the created thread, it will be garbage collected after the execution of the thread terminates.

If a thread execution is terminated because of uncaught exception, and its result is never retrieved by `thread-join!`, a warning will be printed to the standard error port notifying "thread dies a lonely death": It usually indicates some coding error. If you don't collect the result of threads, you have to make sure that all the exceptions are captured and handled within *thunk*.

Internally, this procedure just allocates and initializes a Scheme thread object; the POSIX thread is not created until `thread-start!` is called.

thread-state *thread* [Function]

{gauche.threads} Returns one of symbols new, runnable, stopped or terminated, indicating the state of *thread*.

thread-name *thread* [Function]

[SRFI-18], [SRFI-21] {gauche.threads} Returns the value of *name* slot of *thread*.

thread-specific *thread* [Function]

thread-specific-set! *thread value* [Function]

[SRFI-18], [SRFI-21] {gauche.threads} Gets/sets the value of the *thread*'s specific slot.

thread-start! *thread* [Function]

[SRFI-18], [SRFI-21] {gauche.threads} Starts the *thread*. It is an error if *thread* is already started. Returns *thread*.

thread-yield! [Function]

[SRFI-18], [SRFI-21] {gauche.threads} Suspends the execution of the calling thread and yields CPU to other waiting runnable threads, if any.

thread-sleep! *timeout* [Function]

[SRFI-18], [SRFI-21] {gauche.threads} Suspends the calling thread for the period specified by *timeout*, which must be either a <time> object (see Section 6.25.9 [Time], page 269) that specifies absolute point of time, or a real number that specifies relative point of time from the time this procedure is called in number of seconds.

After the specified time passes, thread-sleep! returns with unspecified value.

If *timeout* points a past time, thread-sleep! returns immediately.

thread-stop! *thread :optional timeout timeout-val* [Function]

{gauche.threads} Stops execution of the target *thread* temporarily. You can resume the execution of the *thread* by thread-cont!.

The stop request is handled synchronously; that is, Gauche VM only checks the request at the "safe" point of the VM and stops itself. It means if the *thread* is blocked by a system call, it won't become stopped state until the system call returns.

By default, thread-stop! returns after the target thread stops. Since it may take indefinitely, you can give optional *timeout* argument to specify timeout. The *timeout* argument can be #f, which means no timeout, or a <time> object that specifies an absolute point of time, or a real number specifying the number of seconds to wait.

The return value of thread-stop! is *thread* if it could successfully stop the target, or *timeout-val* if timeout reached. When *timeout-val* is omitted, #f is assumed.

If the target *thread* has already been stopped by the caller thread, this procedure returns *thread* immediately.

When *thread-stop!* is timed out, the request remains effective even after thread-stop! returns. That is, the target thread may stop at some point in future. The caller thread is expected to call thread-stop! again to complete the stop operation.

An error is signaled if the target thread has already been stopped by another thread (including the "pending" stop request issued by other threads), or the target thread is in neither runnable nor stopped state.

thread-cont! *thread* [Function]

{gauche.threads} Resumes execution of *thread* which has been stopped by thread-stop!. An error is raised if *thread* is not in stopped state, or it is stopped by another thread.

If the caller thread has already requested to stop the target thread but timed out, calling thread-cont! cancels the request.

`thread-terminate!` *thread* [Function]

[SRFI-18], [SRFI-21] {`gauche.threads`} Terminates the specified thread *thread*. The *thread* is terminated and an instance of `<terminated-thread-exception>` is stored in the result exception field of *thread*.

If *thread* is the same as the calling thread, this procedure won't return. Otherwise, this procedure returns unspecified value.

This procedure should be used with care, since *thread* won't have a chance to call cleanup procedures (such as 'after' thunks of `dynamic-wind`). If *thread* is in a critical section, it can leave some state inconsistent. However, once a thread is terminated, any mutex that the thread has kept becomes 'abandoned' state, and an attempt to lock such a mutex by other thread raises an 'abandoned mutex exception', so that you will know the situation. See Section 9.33.3 [Synchronization primitives], page 462.

`thread-join!` *thread :optional timeout timeout-val* [Function]

[SRFI-18], [SRFI-21] {`gauche.threads`} Waits termination of *thread*, or until the timeout is reached if *timeout* is given.

Timeout must be either a `<time>` object (see Section 6.25.9 [Time], page 269) that specifies absolute point of time, or a real number that specifies relative point of time from the time this procedure is called in number of seconds, or `#f` that indicates no timeout (default).

If *thread* terminates normally, `thread-join!` returns a value which is stored in the result field of *thread*. If *thread* terminates abnormally, `thread-join!` raises an exception which is stored in the result exception field of *thread*. It can be either a `<terminated-thread-exception>` or `<uncaught-exception>`.

If the timeout is reached, *thread-join!* returns *timeout-val* if given, or raises `<join-timeout-exception>`.

See Section 9.33.4 [Thread exceptions], page 467, for the details of these exceptions.

9.33.3 Synchronization primitives

Mutexes and condition variables are the low-level synchronization devices. Defined in srfi-18 and srfi-21, they are portable across Scheme implementations that supports one of those srfis. However, in most cases you want to use following higher-level synchronization utilities:

Atoms
: An atom is a wrapper of arbitrary Scheme object, and allows synchronized access to it somewhat like Java's `synchronized` blocks. Atoms are explained in this section.

MT-Queues
: Thread-safe queues (`<mtqueue>`) are provided in `data.queue` module (see Section 12.12 [Queue], page 656), which works as a synchronized channel and suitable to implement producer-consumer pattern.

Mutex

`<mutex>` [Builtin Class]

{`gauche.threads`} A primitive synchronization device. It can take one of four states: locked/owned, locked/not-owned, unlocked/abandoned and unlocked/not-abandoned. A mutex can be locked (by `mutex-lock!`) only if it is in unlocked state. An 'owned' mutex keeps a thread that owns it. Typically an owner thread is the one that locked the mutex, but you can make a thread other than the locking thread own a mutex. A mutex becomes unlocked either by `mutex-unlock!` or the owner thread terminates. In the former case, a mutex becomes unlocked/not-abandoned state. In the latter case, a mutex becomes unlocked/abandoned state.

A mutex has the following external slots.

`name` [Instance Variable of `<mutex>`]
The name of the mutex.

`state` [Instance Variable of `<mutex>`]
The state of the mutex. This is a read-only slot. See the description of `mutex-state` below.

`specific` [Instance Variable of `<mutex>`]
A slot an application can keep arbitrary data. For example, an application can implement a 'recursive' mutex using the specific field.

`mutex?` *obj* [Function]
[SRFI-18], [SRFI-21] {`gauche.threads`} Returns `#t` if *obj* is a mutex, `#f` otherwise.

`make-mutex` *:optional name* [Function]
[SRFI-18], [SRFI-21] {`gauche.threads`} Creates and returns a new mutex object. When created, the mutex is in unlocked/not-abandoned state. Optionally, you can give a name to the mutex.

`mutex-name` *mutex* [Function]
[SRFI-18], [SRFI-21] {`gauche.threads`} Returns the name of the mutex.

`mutex-specific` *mutex* [Function]
`mutex-specific-set!` *mutex value* [Function]
[SRFI-18], [SRFI-21] {`gauche.threads`} Gets/sets the specific value of the mutex.

`mutex-state` *mutex* [Function]
[SRFI-18], [SRFI-21] {`gauche.threads`} Returns the state of *mutex*, which may be one of the followings:

a thread The mutex is locked/owned, and the owner is the returned thread.

symbol `not-owned`
The mutex is locked/not-owned.

symbol `abandoned`
The mutex is unlocked/abandoned.

symbol `not-abandoned`
The mutex is unlocked/not-abandoned.

`mutex-lock!` *mutex :optional timeout thread* [Function]
[SRFI-18], [SRFI-21] {`gauche.threads`} Locks *mutex*. If *mutex* is in unlocked/not-abandoned state, this procedure changes its state to locked state exclusively. By default, *mutex* becomes locked/owned state, owned by the calling thread. You can give other owner thread as *thread* argument. If *thread* argument is given and `#f`, the mutex becomes locked/not-owned state.

If *mutex* is in unlocked/abandoned state, that is, some other thread has been terminated without unlocking it, this procedure signals 'abandoned mutex exception' (see Section 9.33.4 [Thread exceptions], page 467) after changing the state of *mutex*.

If *mutex* is in locked state and *timeout* is omitted or `#f`, this procedure blocks until *mutex* becomes unlocked. If *timeout* is specified, `mutex-lock!` returns when the specified time reaches in case it couldn't obtain a lock. You can give *timeout* an absolute point of time (by `<time>` object, see Section 6.25.9 [Time], page 269), or a relative time (by a real number).

`Mutex-lock!` returns `#t` if *mutex* is successfully locked, or `#f` if timeout reached.

Note that *mutex* itself doesn't implements a 'recursive lock' feature; that is, if a thread that has locked *mutex* tries to lock *mutex* again, the thread blocks. It is not difficult, however, to implement a recursive lock semantics on top of this mutex. The following example is taken from SRFI-18 document:

```
(define (mutex-lock-recursively! mutex)
  (if (eq? (mutex-state mutex) (current-thread))
      (let ((n (mutex-specific mutex)))
        (mutex-specific-set! mutex (+ n 1)))
      (begin
        (mutex-lock! mutex)
        (mutex-specific-set! mutex 0))))

(define (mutex-unlock-recursively! mutex)
  (let ((n (mutex-specific mutex)))
    (if (= n 0)
        (mutex-unlock! mutex)
        (mutex-specific-set! mutex (- n 1)))))
```

`mutex-unlock!` *mutex :optional condition-variable timeout* [Function]

[SRFI-18], [SRFI-21] {`gauche.threads`} Unlocks *mutex*. The state of *mutex* becomes unlocked/not-abandoned. It is allowed to unlock a mutex that is not owned by the calling thread.

If optional *condition-variable* is given, `mutex-unlock!` serves the "condition variable wait" operation (e.g. `pthread_cond_wait` in POSIX threads). The current thread atomically wait on *condition-variable* and unlocks *mutex*. The thread will be unblocked when other thread signals on *condition-variable* (see `condition-variable-signal!` and `condition-variable-broadcast!` below), or *timeout* reaches if it is supplied. The *timeout* argument can be either a `<time>` object to represent an absolute time point (see Section 6.25.9 [Time], page 269), a real number to represent a relative time in seconds, or `#f` which means never. The calling thread may be unblocked prematurely, so it should reacquire the lock of *mutex* and checks the condition, as in the following example (it is taken from SRFI-18 document):

```
(let loop ()
  (mutex-lock! m)
  (if (condition-is-true?)
      (begin
        (do-something-when-condition-is-true)
        (mutex-unlock! m))
      (begin
        (mutex-unlock! m cv)
        (loop))))
```

The return value of `mutex-unlock!` is `#f` when it returns because of timeout, and `#t` otherwise.

`mutex-locker` *mutex* [Function]
`mutex-unlocker` *mutex* [Function]

{`gauche.threads`} Returns `(lambda () (mutex-lock! mutex))` and `(lambda () (mutex-unlock! mutex))`, respectively. Each closure is created at most once per *mutex*, thus it is lighter than using literal lambda forms in a tight loop.

`with-locking-mutex` *mutex thunk* [Function]

{`gauche.threads`} Calls *thunk* with locking a mutex *mutex*. This is defined as follows.

```
(define (with-locking-mutex mutex thunk)
```

```
(dynamic-wind
 (mutex-locker mutex)
 thunk
 (mutex-unlocker mutex)))
```

Condition variable

`<condition-variable>` [Builtin Class]

{`gauche.threads`} A condition variable keeps a set of threads that are waiting for a certain condition to be true. When a thread modifies the state of the concerned condition, it can call `condition-variable-signal!` or `condition-variable-broadcast!`, which unblock one or more waiting threads so that they can check if the condition is satisfied.

A condition variable object has the following slots.

`name` [Instance Variable of `<condition-variable>`]

The name of the condition variable.

`specific` [Instance Variable of `<condition-variable>`]

A slot an application can keep arbitrary data.

Note that SRFI-18 doesn't have a routine equivalent to pthreads' `pthread_cont_wait`. If you want to wait on condition variable, you can pass a condition variable to `mutex-unlock!` as an optional argument (see above), then acquire mutex again by `mutex-lock!`. This design is for flexibility; see SRFI-18 document for the details.

This is the common usage of pthreads' condition variable:

```
while (some_condition != TRUE) {
  pthread_cond_wait(condition_variable, mutex);
}
```

And it can be translated to SRFI-18 as follows:

```
(let loop ()
  (unless some-condition
    (mutex-unlock! mutex condition-variable)
    (mutex-lock! mutex)
    (loop)))
```

`condition-variable?` *obj* [Function]

[SRFI-18], [SRFI-21] {`gauche.threads`} Returns `#t` if *obj* is a condition variable, `#f` otherwise.

`make-condition-variable` *:optional name* [Function]

[SRFI-18], [SRFI-21] {`gauche.threads`} Returns a new condition variable. You can give its name by optional *name* argument.

`condition-variable-name` *cv* [Function]

[SRFI-18], [SRFI-21] {`gauche.threads`} Returns the name of the condition variable.

`condition-variable-specific` *cv* [Function]
`condition-variable-specific-set!` *cv value* [Function]

[SRFI-18][SRFI-21] {`gauche.threads`} Gets/sets the specific value of the condition variable.

`condition-variable-signal!` *cv* [Function]

[SRFI-18][SRFI-21] {`gauche.threads`} If there are threads waiting on *cv*, causes the scheduler to select one of them and to make it runnable.

`condition-variable-broadcast!` *cv* [Function]

[SRFI-18][SRFI-21] {`gauche.threads`} Unblocks all the threads waiting on *cv*.

Atom

An atom is a convenient wrapper to make operations on a given set of objects thread-safe. Instead of defining thread-safe counterparts of every structure, you can easily wrap an existing data structures to make it thread-safe.

`atom` *val* . . . [Function]

{`gauche.threads`} Creates and returns an atom object with *val* . . . as the initial values.

`atom?` *obj* [Function]

{`gauche.threads`} Returns `#t` iff *obj* is an atom.

The following procedures can be used to *atomically* access and update the content of an atom. They commonly take optional *timeout* and *timeout-val* arguments, both are defaulted to `#f`. In some cases, the procedure takes more than one *timeout-val* arguments. With the default value `#f` as *timeout* argument, those procedures blocks until they acquire a lock.

The timeout arguments can be used to modify the behavior when the lock cannot be acquired in timely manner. *timeout* may be a `<time>` object (see Section 6.25.9 [Time], page 269) to specify an absolute point of time, or a real number to specify the relative time in seconds. If timeout is expired, those procedures give up acquiring the lock, and the value given to *timeout-val* is returned. In `atomic` and `atomic-update!`, you can make them return multiple timeout values, by giving more than one *timeout-val* arguments.

`atom-ref` *atom :optional index timeout timeout-val* [Function]

{`gauche.threads`} Returns *index*-th value of *atom*. See above for *timeout* and *timeout-val* arguments.

```
(define a (atom 'a 'b))

(atom-ref a 0) ⇒ a
(atom-ref a 1) ⇒ b
```

`atomic` *atom proc :optional timeout timeout-val timeout-val2 . . .* [Function]

{`gauche.threads`} Calls *proc* with the current values in *atom*, while locking *atom*. *proc* must take as many arguments as the number of values *atom* has.

The returned value(s) of *proc* is the result of `atomic`, unless timeout occurs. See above for *timeout* and *timeout-val* arguments.

For example, the `ref/count` procedure in the following example counts the number of times the hashtable is referenced in thread-safe way.

```
(define a (atom (make-hash-table 'eq?) (list 0)))

(define (ref/count a key)
  (atomic a
   (lambda (ht count-cell)
     (inc! (car count-cell))
     (hash-table-get h key))))
```

`atomic-update!` *atom proc :optional timeout timeout-val timeout-val2 . . .* [Function]

{`gauche.threads`} Calls *proc* with the current values in *atom* while locking *atom*, and updates the values in *atom* by the returned values from *proc*. *proc* must take as many arguments as the number of values *atom* has, and must return the same number of values.

The returned value(s) of `atomic-update!` is what *proc* returns, unless timeout occurs. See above for *timeout* and *timeout-val* arguments.

The following example shows a thread-safe counter.

```
(define a (atom 0))

(atomic-update! a (cut + 1 <>))
```

Note: The term *atom* in historical Lisps meant objects that are not a cons cell (pair). Back then cons cells were the only aggregated datatype and there were few other datatypes (numbers and symbols), so having a complementary term to cells made sense.

Although it still appears in introductory Lisp tutorials, modern Lisps, including Scheme, has so many datatypes and it makes little sense to have a specific term for non-aggregate types.

Clojure adopted the term *atom* for thread-safe (atomic) primitive data, and we followed it.

Note: The constructor of atom is not `make-atom` but `atom`, following the convention of `list`/`make-list`, `vector`/`make-vector`, and `string`/`make-string`; that is, the name without `make-` takes its elements as variable number of arguments.

9.33.4 Thread exceptions

Some types of exceptions may be thrown from thread-related procedures. These exceptions can be handled by Gauche's exception mechanism (see Section 6.20 [Exceptions], page 204).

`<thread-exception>` [Builtin Class]

{`gauche.threads`} A base class of thread-related exceptions. Inherits `<exception>` class. It has one slot.

`thread` [Instance Variable of `<thread-exception>`]

A thread that threw this exception.

`<join-timeout-exception>` [Builtin Class]

{`gauche.threads`} An exception thrown by `thread-join!` when a timeout reaches before the waited thread returns. Inherits `<thread-exception>`.

`<abandoned-mutex-exception>` [Builtin Class]

{`gauche.threads`} An exception thrown by `mutex-lock!` when a *mutex* to be locked is in unlocked/abandoned state. Inherits `<thread-exception>`. It has one additional slot.

`mutex` [Instance Variable of `<abandoned-mutex-exception>`]

A mutex that caused this exception.

`<terminated-thread-exception>` [Builtin Class]

{`gauche.threads`} An exception thrown by `thread-join!` when the waited thread is terminated abnormally (by `thread-terminate!`). Inherits `<thread-exception>`. It has one additional slot.

`terminator` [Instance Variable of `<terminated-thread-exception>`]

A thread that terminated the thread that causes this exception.

`<uncaught-exception>` [Builtin Class]

{`gauche.threads`} An exception thrown by `thread-join!` when the waited thread is terminated by an uncaught exception. Inherits `<thread-exception>`. It has one additional slot.

`reason` [Instance Variable of `<uncaught-exception>`]

An exception that caused the termination of the thread.

`join-timeout-exception?` *obj* [Function]
`abandoned-mutex-exception?` *obj* [Function]
`terminated-thread-exception?` *obj* [Function]
`uncaught-exception?` *obj* [Function]

[SRFI-18], [SRFI-21] {`gauche.threads`} These procedures checks if *obj* is a certain type of exception. Provided for the compatibility to SRFI-18.

`uncaught-exception-reason` *exc* [Function]

[SRFI-18], [SRFI-21] {`gauche.threads`} Returns the value of `reason` slot of `<uncaught-exception>` object. Provided for the compatibility to SRFI-18.

9.34 `gauche.time` - Measure timings

`gauche.time` [Module]

Provides three ways to measure execution time of Scheme code. A macro `time`, which is convenient for interactive use, a set of procedures for benchmarking, and `<time-counter>` objects which are useful to be embedded in the program.

Interactive measurement of execution time

Note: The `time` macro is pre-defined to autoload `gauche.time` for the convenience; you don't need to say `(use gauche.time)` to use the `time` macro.

`time` *expr expr2 . . .* [Macro]

{`gauche.time`} Evaluates *expr expr2 . . .* sequentially, as `begin`, and returns the result(s) of the last expression. Before returning the value(s), the macro reports the elapsed (real) time and CPU times in the user space and the kernel space to the current error port, much like the bourne shell's time command.

The current version uses `sys-gettimeofday` (see Section 6.25.9 [Time], page 269) to calculate the elapsed time, and `sys-times` (see Section 6.25.8 [System inquiry], page 266) to calculate user and system CPU times. So the resolution of these numbers depends on these underlying system calls. Usually the CPU time has 10ms resolution, while the elapsed time might have higher resolution. On the systems that doesn't have gettimeofday(2) support, however, the elapsed time resolution can be as bad as a second.

```
gosh> (time (length (sort (call-with-input-file "/usr/share/dict/words"
                                                port->string-list))))
;(time (length (sort (call-with-input-file "/usr/share/dict/words" port- ...
; real   0.357
; user   0.350
; sys    0.000
45427
```

Benchmarking

It is not unusual that the routine you want to measure takes only a fraction of second, so you have to run it many times for better measurement. It is also common that you want to compare results of measurement of two or more implementation strategies. Here are useful procedures to do so.

The name and behavior of those benchmarking routines are inspired by Perl's Benchmark module.

`time-this` *how thunk* [Function]

{`gauche.time`} Calls *thunk* many times and measure its execution time. The argument *how* can be one of the following forms.

`integer` It calls *thunk* as many times as the given number.

(cpu *real*)
: It calls *thunk* as many times as the total cpu time exceeds the given number of seconds.

It also runs an empty loop as the same times and subtract the time took for the empty loop from the measured time, to get more accurate result.

The result is returned in a `<time-result>` record, described below. Here are some examples:

```
;; Run the thunk 1,000,000 times
(time-this 1000000 (lambda () (expt 100 30)))
  ⇒ #<time-result 1000000 times/  1.030 real/  1.040 user/  0.000 sys>

;; Run the thunk at least 5.0 cpu seconds
(time-this '(cpu 5.0) (lambda () (expt 100 30)))
  ⇒ #<time-result 4903854 times/  5.090 real/  5.050 user/  0.010 sys>
```

`<time-result>` [Record]

{`gauche.time`} A record to hold the benchmark result. Following slots are defined.

`count` [Instance Variable of `<time-result>`]

The number of times the thunk was run. This slot is also accessed by a procedure `time-result-count`.

`real` [Instance Variable of `<time-result>`]

The total real (elapsed) time running the thunk took. This slot is also accessed by a procedure `time-result-real`.

`user` [Instance Variable of `<time-result>`]

The total user cpu time running the thunk took. This slot is also accessed by a procedure `time-result-user`.

`sys` [Instance Variable of `<time-result>`]

The total system cpu time running the thunk took. This slot is also accessed by a procedure `time-result-sys`.

`make-time-result` *count real user sys* [Function]

{`gauche.time`} The constructor of `<time-result>` records.

`time-result?` *obj* [Function]

{`gauche.time`} The predicate of `<time-result>` records.

`time-result+` *t1 t2 :key* (*with-count #f*) [Function]
`time-result-` *t1 t2 :key* (*with-count #f*) [Function]

{`gauche.time`} Add or subtract two `<time-result>` records and returns a new record.

If *with-count* is false, only the real, user and sys slots are added or subtracted, and the result's count slot is set to the same as *t1*'s count slot. It is supposed to be used to calculate on measurement from different chunk of code.

If *with-count* is true, then the values of count slot is also added or subtracted. It is supposed to calculate on multiple benchmark results of the same code.

`time-these` *how alist* [Function]
`time-these/report` *how alist* [Function]

{`gauche.time`} These procedures benchmarks multiple chunks of code to compare.

The *alist* argument must be the form of `((key . thunk) ...)`, where `key` is a symbol and *thunk* is a procedure taking no arguments.

The *how* argument is the same as `time-this`; that is, either an integer for number of iterations, or a list `(cpu x)` to indicate *x* seconds of cpu time.

`time-these` runs benchmarks for each thunk in *alist* using `time-this`, and returns the result in a list of the form `(how (key1 . result1) (key2 . result2) ...)`, where each *result* is a `<time-result>` object.

`time-these/report` outputs the benchmark results and comparison matrix in human readable way to the current output port.

```
gosh> (time-these/report '(cpu 3.0)
        `((real1 . ,(cut expt 100 20))
          (real2 . ,(cut %expt 100 20))
          (imag  . ,(cut expt +100i 20))))
Benchmark: ran real1, real2, imag, each for at least 3.0 cpu seconds.
  real1: 3.312 real, 3.320 cpu (3.320 user + 0.000 sys)@ 1694277.11/s n=5625000
  real2: 2.996 real, 3.010 cpu (3.010 user + 0.000 sys)@35595634.55/s n=107142860
   imag: 3.213 real, 3.190 cpu (3.190 user + 0.000 sys)@  862068.97/s n=2750000

              Rate  real1 real2   imag
  real1  1694277/s     -- 0.048  1.965
  real2 35595635/s 21.009    -- 41.291
   imag   862069/s  0.509 0.024     --
```

The first part of the report shows, for each thunks, the real (elapsed) time, the cpu time used (and its breakdown of user and system time), the rate of iteration per second, and the total number of iterations.

The second part compares the speed between each pair of the benchmarks. For example, its first row tells that the benchmark `real1` is 0.048 times faster than `real2` and 1.965 times faster than `imag`.

`report-time-results` *result* [Function]

{`gauche.time`} This is a utility procedure to create a report from the result of `time-these`. Actually, `time-these/report` is just a combination of `time-these` and this procedure:

```
(define (time-these/report how samples)
  (report-time-results (time-these how samples)))
```

Finer measurement

`<time-counter>` [Class]

{`gauche.time`} An abstract class of time counters. Time counter is a kind of timer whose value is incremented as the time passes. The counting can be started and stopped any number of times. The value of the counter can be read when the timer is stopping. You can have multiple time counters. It is useful, for example, to measure the time in two parts inside a loop independently.

The concrete subclass determines which time it is counting. You have to instantiate one of those subclasses described below to use the time counter.

`<real-time-counter>` [Class]
`<user-time-counter>` [Class]
`<system-time-counter>` [Class]
`<process-time-counter>` [Class]

{`gauche.time`} Classes for time counters that count real (elapsed) time, user-space CPU time, kernel-space CPU time, and total CPU time (user + system), respectively.

`time-counter-start!` (*counter* <*time-counter*>) [Method]
`time-counter-stop!` (*counter* <*time-counter*>) [Method]

{`gauche.time`} Starts and stops the *counter*. The time during the counter is running is accumulated to the counter value when the counter is stopped.

Start/stop pairs can be nested, but only the outermost pair takes the effect. That is, if you call `time-counter-start!` on the counter that is already started, it doesn't have any effect except that to stop such a counter you have to call `time-counter-stop!` one more time. It is useful when you want to measure the time spent in the larger block that may already contain timer start/stop pairs.

Calling `time-counter-stop!` on the already stopped counter has no effect.

`time-counter-reset!` (*counter* <*time-counter*>) [Method]

{`gauche.time`} Resets the value of *counter*. If *counter* is already running, it is forced to stop before being reset.

`time-counter-value` (*counter* <*time-counter*>) [Method]

{`gauche.time`} Returns the current value of the counter as the number of seconds, in a real number. The resolution depends on the source of the counter.

`with-time-counter` *counter expr* . . . [Macro]

{`gauche.time`} A convenience macro to run the *counter* while *expr* . . . are evaluated. Returns the result(s) of the last expression. It is defined as follows.

```
(define-syntax with-time-counter
  (syntax-rules ()
    ((_ counter . exprs)
     (dynamic-wind
      (lambda () (time-counter-start! counter))
      (lambda () . exprs)
      (lambda () (time-counter-stop! counter))))
    ))
```

The following example measures approximate times spend in process-A and process-B inside a loop.

```
(let ((ta (make <real-time-counter>))
      (tb (make <real-time-counter>)))
  (dotimes (i 100000)
    (with-time-counter ta
      (process-A))
    (with-time-counter tb
      (process-B)))
  (format #t "Time spent in process-A: ~s\n" (time-counter-value ta))
  (format #t "Time spent in process-B: ~s\n" (time-counter-value tb))
  )
```

9.35 `gauche.unicode` - Unicode utilities

`gauche.unicode` [Module]

This module provides various operations on a sequence of Unicode codepoints.

Gauche can be compiled with a native encoding other than Unicode, and the full Unicode-compatible behavior on characters and strings may not be available on such systems. So we provide most operations in two flavors: Operations on characters and strings, or operations on codepoints represented as a sequence of integers.

If Gauche is compiled with its native encoding being `none`, `euc-jp` or `sjis`, character-and-string operations are likely to be partial functions of the operations defined in Unicode standard. That is, if the operation can yield a character that are not supported in the native encoding, it may be remapped to an alternative character. Each manual entry explains the detailed behavior.

The codepoint operations are independent from Gauche's native encoding and supports full spec as defined in Unicode standard. If Gauche is compiled with the `utf-8` native encoding, the operations are essentially the same as character-and-string flavors when you convert codepoints and characters by `char->integer` and `integer->char`. The codepoint operations are handy when you need to support the algorithms described in Unicode standard fully, no matter what the running Gauche's native encoding is.

9.35.1 Unicode transfer encodings

The procedures in this group operate on codepoints represented as integers. In the following descriptions, 'octets' refers to an integer between 0 to 255, inclusive.

They take optional *strictness* argument. It specifies what to do when the procedure encounters a datum outside of the defined domain. Its value can be either one of the following symbols:

`strict` Raises an error when the procedure encounters such input. This is the default behavior.

`permissive`
: Whenever possible, treat the date as if it is a valid value. For example, codepoint value beyond `#x10ffff` is invalid in Unicode standard, but it may be useful for some other purpose that just want to use UTF-8 as an encoding scheme of binary data.

`ignore` Whenver possible, treat the invalid input as if they do not exist.

The procedure may still raise an error in `permissive` or `ignore` strictness mode, if there can't be a sensible way to handle the input data.

`ucs4->utf8` *codepoint :optional strictness* [Function]

{`gauche.unicode`} Takes an integer codepoint and returns a list of octets that encodes the input in UTF-8.

```
(ucs4->utf8 #x3bb)  ⇒ (206 187)
(ucs4->utf8 #x3042) ⇒ (227 129 130)
```

If *strictness* is `strict` (default), input codepoint between `#xd800` to `#xdfff`, and beyond `#x110000`, are rejected. If *strictness* is `permissive`, it accepts input between `0` and `#x7fffffff`, inclusive; it may produce 5 or 6 octets if the input is large (as the original UTF-8 definition). If *strictness* is `ignore`, it returns an empty list for invalid codepoints.

`utf8-length` *octet :optional strictness* [Function]

{`gauche.unicode`} Takes *octet* as the first octet of UTF-8 sequence, and returns the number of total octets required to decode the codepoint.

If *octet* is not an exact integer between 0 and 255 (inclusive), an error is thrown, regardless of *strictness* argument.

If *strictness* is `strict` (default), this procedure returns either 1, 2, 3 or 4. An error is thrown if *octet* cannot be a leading octet of a proper UTF-8 encoded Unicode codepoint.

If *strictness* is `permissive`, this procedure may return an integer between 0 and 6, inclusive. If the input is from `#xf8` to `#xfd`, inclusive, this returns 5 or 6, according to the original utf-8 spec (these values corresponds to the codepoint range `#x110000` to `#x7fffffff`). If the input is in the range between `#x80` and `#xbf`, inclusive, or *#xfe* or `#xff`, this procedure returns 1–it's up to the application how to treat these illegal octets.

If *strictness* is `ignore`, this procedure returns `0` when it would raise an error if *strictness* is `strict`. Other than that, it works the same as the default case.

`utf8->ucs4` *octet-list :optional strictness* [Function]

{`gauche.unicode`} Takes a list of octets, and decodes it as a utf-8 sequence. Returns two values: The decoded ucs4 codepoint, and the rest of the input list.

If it finds a value other than exact integer between 0 and 255 in the input list, an error is thrown regardless of the value of *strictness*.

An invalid utf8 sequence causes an error if *strictness* is `strict`, or skipped if it is `ignore`. If *strictness* is `permissive`, the procedure accepts the original utf-8 sequence which can produce surrogated pair range (between `#xd800` and `#dfff`) and the range between `#x110000` to `#x7fffffff`. The invalid octet sequence is still an error with `permissive` mode.

`utf8->string` *u8vector :optional start end* [Function]

[R7RS base] {`gauche.unicode`} Converts a sequence of utf8 octets in *u8vector* to a string. Optional *start* and/or *end* argument(s) will limit the range of the input.

If Gauche's native encoding is utf8, `u8vector->string` (see Section 9.36.2 [Uvector conversion operations], page 484) will do the job faster; but this routine can be used regardless of Gauche's native encoding, and it raises an error if *u8vector* contains octet sequences illegal as utf8.

`string->utf8` *string :optional start end* [Function]

[R7RS base] {`gauche.unicode`} Converts a string to a u8vector of utf8 octets. Optional *start* and/or *end* argument(s) will limit the range of the input.

If Gauche's native encoding is utf8, `string->u8vector` (see Section 9.36.2 [Uvector conversion operations], page 484) will do the job faster; but this routine can be used regardless of Gauche's native encoding.

`ucs4->utf16` *codepoint :optional strictness* [Function]

{`gauche.unicode`} Takes an integer codepoint and returns a list of integers that encodes the input in UTF-16.

If *strictness* is `strict` (default), the input must be either between `0` and `#xd7ff` or between `#xe000` and `#x10ffff`. An error is thrown otherwise. The 'hole' is the codepoint reserved for surrogates, and there's no valid mapping from them to utf-16 is defined.

If *strictness* is `permissive`, it accepts high surrogates and low surrogates, in which case the result is single element list of input. An error is still thrown for negative input and input greater than or equal to `#x110000`.

If *strictness* is `ignore`, an empty list is returned for an invalid codepoint (including surrogates).

Note: We can encode values larger than `#x10ffff` in utf-8 in the permissive mode, but not in utf-16.

`utf16-length` *code :optional strictness* [Function]

{`gauche.unicode`} *Code* must be an exact integer between 0 and 65535, inclusive. Returns 1 if *code* is BMP character codepoint, or 2 if *code* is a high surrogate.

If *strictness* is `strict` (default), an error is signalled if *code* is a low surrogate, or it is out of range. If *strictness* is `permissive`, 1 is returned for low surrogates, but an error is signalled for out of range arguments. If *strictness* is `ignore`, 0 is returned for low surrogates and out of range arguments.

`utf16->ucs4` *code-list :optional strictness* [Function]

{`gauche.unicode`} Takes a list of exact integers and decodes it as a utf-16 sequence. Returns two values: The decoded ucs4 codepoint, and the rest of input list.

If *strictness* is `strict` (default), an invalid utf-16 sequence and out-of-range integer raise an error. If *strictness* is `permissive`, an out-of-range integer causes an error, but a lone surrogate is allowed and returned as is. If *strictness* is `ignore`, lone surrogates and out-of-range integers are just ignored.

9.35.2 Unicode text segmentation

These procedures implements grapheme-cluster and word breaking algorithms defined in UAX #29: Unicode Text Segmentation.

`string->words` *string* [Function]
`codepoints->words` *sequence* [Function]

{`gauche.unicode`} From given string or codepoint sequence (a `<sequence>` object containing list of codepoints), returns a list of words. Each cluster is represented as a string, or a sequence of the same type as input, respectively.

```
(string->words "That's it.")
 ⇒ ("That's" " " "it" ".")
(codepoints->words '(84 104 97 116 39 115 32 105 116 46)
 ⇒ ((84 104 97 116 39 115) (32) (105 116) (46))
```

In the second example, the list is a list of codepoints of characters in "That's it."

`string->grapheme-clusters` *string* [Function]
`codepoints->grapheme-clusters` *sequence* [Function]

{`gauche.unicode`} From given string or codepoint sequence (a `<sequence>` object containing list of codepoints), returns a list of grapheme clusters. Each cluster is represented as a string, or a sequence of the same type as input, respectively.

The following procedures are low-level building blocks to build the above `string->words` etc. A *generator* argument is a procedure with no arguments, and returns a value (or some values) at at time for every call, until it returns EOF.

`make-word-breaker` *generator* [Function]
`make-grapheme-cluster-breaker` *generator* [Function]

{`gauche.unicode`} From given *generator* is a generator of characters or codepoints, returns a generator that returns two values: The first value is the character or codepoint generated from the original generator, and the second value is a boolean flag, which is `#t` if a word or a grapheme cluster breaks before the character/codepoint, and `#f` otherwise.

Suppose a generator *g* returns characters in a string `That's it.`, one at a time. Then the created generator will work as follows:

```
(define brk (make-word-breaker g))
(brk)  ⇒  #\T     and #t
(brk)  ⇒  #\h     and #f
(brk)  ⇒  #\a     and #f
(brk)  ⇒  #\t     and #f
(brk)  ⇒  #\'     and #f
(brk)  ⇒  #\s     and #f
(brk)  ⇒  #\space and #t
(brk)  ⇒  #\i     and #t
(brk)  ⇒  #\t     and #f
(brk)  ⇒  #\.     and #t
(brk)  ⇒  #<eof>  and #t
```

It shows the word breaks at those character boundaries shown by the caret `^` below (for clearity, I use `_` to indicate the space).

```
 T h a t ' s _ i t .
^           ^ ^   ^ ^
```

`make-word-reader` *generator return* [Function]
`make-grapheme-cluster-reader` *generator return* [Function]

{`gauche.unicode`} The input *generator* is a generator of characters or codepoints, and *return* is a procedure that takes a list of characters or codepoints, and returns an object. These procedures creates a generator that returns an object at at time, each consists of a word or a grapheme cluster, respectively.

Suppose a generator *g* returns characters in a string `That's it.`, one at a time, again. Then the created generator works as follows:

```
(define brk (make-word-reader g list->string))
(brk)  ⇒  "That's"
(brk)  ⇒  " "
(brk)  ⇒  "it"
(brk)  ⇒  "."
(brk)  ⇒  #<eof>
```

9.35.3 Full string case conversion

`string-upcase` *string* [Function]
`string-downcase` *string* [Function]
`string-titlecase` *string* [Function]
`string-foldcase` *string* [Function]

[R6RS][R7RS char] {`gauche.unicode`} Converts given *string* to upper case, using language-independent full case folding defined by Unicode standard. They differ from srfi-13's procedures with the same names (see Section 11.5.8 [SRFI-13 String case mapping], page 596), which simply uses character-by-character case mapping. Notably, the length of resulting string may differ from the source string, and some conversions are sensitive to whether the character is at the word boundary or not. The word boundaries are determined according to UAX #29 text segmentation rules.

```
(string-upcase "straße")
 ⇒ "STRASSE"
(string-downcase "ΧΑΟΣΧΑΟΣ.ΧΑΟΣ. Σ.")
 ⇒ "χαοσχαοσ.χαος. σ."
(string-titlecase "You're talking about R6RS, right?")
 ⇒ "You're Talking About R6rs, Right?"
(string-foldcase "straße")
 ⇒ "strasse"
(string-foldcase "ΧΑΟΣΣ")
 ⇒ "χαοσσ"
```

Note that `string-titlecase` isn't included in R7RS `scheme.char` module.

`codepoints-upcase` *sequence* [Function]
`codepoints-downcase` *sequence* [Function]
`codepoints-titlecase` *sequence* [Function]
`codepoints-foldcase` *sequence* [Function]

{`gauche.unicode`} Like `string-upcase` etc, but these work on a sequence of codepoints instead. Returns a sequence of the same type of the input.

```
(codepoints-upcase '#(115 116 114 97 223 101))
 ⇒ #(83 84 82 65 83 83 69)
```

`string-ci=?` *string1 string2 string3 . . .* [Function]
`string-ci<?` *string1 string2 string3 . . .* [Function]
`string-ci<=?` *string1 string2 string3 . . .* [Function]

`string-ci>?` *string1 string2 string3 . . .* [Function]
`string-ci>=?` *string1 string2 string3 . . .* [Function]

[R7RS char] {`gauche.unicode`} Case-insensitive string comparison, using full-string case conversion.

Note that Gauche has builtin `string-ci=?` etc., which use character-wise case folding (see Section 6.12.6 [String Comparison], page 158). These are different procedures.

```
(string-ci=? "\u00df" "SS") ⇒ #t
```

9.35.4 East asian width property

`char-east-asian-width` *char-or-codepoint* [Function]

{`gauche.unicode`} The argument may be a character or a nonnegative integer of Unicode codepoint. Returns one of the symbols `N` (neutral), `F` (fullwidth), `H` (halfwidth), `W` (wide), `Na` (narrow), and `A` (ambiguous).

The meaning of this property is explained in Unicode standard annex #11, `http://unicode.org/reports/tr11/`.

9.36 `gauche.uvector` - Uniform vectors

`gauche.uvector` [Module]

Provides procedures that work on vectors whose elements are of the same numeric type, as defined in srfi-160 (formerly srfi-4), which has become R7RS large (as `scheme.vector.@`.

The `@` part is actually one of the following tags, indicating the type of elements:

`u8` Unsigned 8-bit integer - an exact integer between 0 and 255.

`s8` Signed 8-bit integer - an exact integer between -128 and 127.

`u16` Unsigned 16-bit integer - an exact integer between 0 and 65535.

`s16` Signed 16-bit integer - an exact integer between -32678 and 32767.

`u32` Unsigned 32-bit integer - an exact integer between 0 and 2^32 - 1.

`s32` Signed 32-bit integer - an exact integer between -(2^31) and 2^31 - 1.

`u64` Unsigned 64-bit integer - an exact integer between 0 and 2^64 - 1.

`s64` Signed 64-bit integer - an exact integer between -(2^63) and 2^63 - 1.

`f16` 16-bit floating point number (10-bit mantissa and 5-bit exponent), as inexact real.

`f32` IEEE single-precision floating point number as inexact real.

`f64` IEEE double-precision floating point number as inexact real.

`c32` Inexact complex, consists of a pair of 16-bit floating point numbers.

`c64` Inexact complex, consists of a pair of IEEE single-precision floating point numbers.

`c128` Inexact complex, consists of a pair of IEEE double-precision floating point numbers.

There are some advantages of using uniform vectors over normal (heterogeneous) vectors. It may be more compact than the normal vectors. Some operations (especially Gauche's extension of vector arithmetic operations) can bypass type check and conversion of individual elements, thus be more efficient. And it is much easier and efficient to communicate with external libraries that require homogeneous array of numbers; for example, OpenGL binding of Gauche uses uniform vectors extensively.

Gauche's implementation is a superset of srfi-160 in a few ways:

- Support of `f16vector` and `c32vector`, using 16-bit floating point numbers as used in high-dynamic range image format.
- Efficient element-wise arithmetic procedures, e.g. `@vector-add`.
- Implements the collection framework (see Section 9.5 [Collection framework], page 344) and the sequence framework (see Section 9.29 [Sequence framework], page 441). So the methods like `map`, `for-each`, `ref` or `subseq` can be used.
- Some routines takes optional parameters: `@vector-ref` takes optional fallback value.

When you try to store a number out of the range of the vector type, an error is signaled by default. However, some procedures take an optional argument *clamp* that specifies alternative behavior in such a case. *Clamp* argument may take one of the following values.

`#f` Default behavior (signals an error).

`high` Clamps high bound; i.e. if the value to be stored is beyond the higher bound of the range, the maximum value is stored instead.

`low` Clamps low bound; i.e. if the value to be stored is below the lower bound of the range, the minimum value is stored instead.

`both` Clamps both sides; does both `high` and `low`.

```
(list->u8vector '(-1))          ⇒ error
(list->u8vector '(-1) 'low)     ⇒ #u8(0)
(list->u8vector '(-1) 'high)    ⇒ error
(list->u8vector '(3000) 'high) ⇒ #u8(255)
(list->u8vector '(-100 20 300) 'both) ⇒ #u8(0 20 255)
```

In the following description, `@` can be replaced for any of `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f16`, `f32`, `f64`, `c32`, `c64` or `c128`.

Note: R7RS-large provides separate library for each type, and you should import them individually, for example, `(use scheme.vector.u8)` (Gauche way) or `(import (scheme vector u8))` (R7RS way).

On the other hand, using `gauche.uvector` imports all the bindings.

9.36.1 Uvector basic operations

`<@vector>` [Builtin Class]

{`gauche.uvector`} A class for @vector. It inherits `<sequence>`.

`#@(`*n* `...)` [Reader Syntax]

Denotes a literal homogeneous vector.

```
#s8(3 -2 4)
#u32(4154 88357 2 323)
#f32(3.14 0.554525 -3.342)
```

`@vector?` *obj* [Function]

[R7RS vector.@] {`gauche.uvector`} Returns `#t` iff *obj* is a @vector, `#f` otherwise.

`uvector?` *obj* [Function]

{`gauche.uvector`} Returns `#t` iff *obj* is of any uniform vector type.

`@?` *obj* [Function]

[R7RS vector.@] {`gauche.uvector`} Returns `#t` iff *obj* can be an element of `@vector`.

@vector-empty? *obj* [Function]

[R7RS vector.@] {gauche.uvector} The argument must be a @vector. Returns #t iff it is empty.

@vector *x* . . . [Function]

[R7RS vector.@] {gauche.uvector} Constructs @vector whose elements are numbers *x* . . . . The numbers must be exact integer for exact integer vectors, and in the valid range of the vector.

```
(s8vector 1 2 3) ⇒ #s8(1 2 3)
```

make-@vector *len :optional fill* [Function]

[R7RS vector.@] {gauche.uvector} Constructs a @vector of length *len*. The elements are initialized by a number *fill*. For exact integer vectors, *fill* must be an exact integer and in the valid range. If *fill* is omitted, the content of the vector is undefined.

```
(make-u8vector 4 0) ⇒ #u8(0 0 0 0)
```

make-uvector class len *:optional* **fill** [Function]

{gauche.uvector} This is a Gauche extension; instead of using separate constructor for each uvector type, you can pass the class of desired uvector.

```
(make-uvector <u8vector> 3)    ⇒ #u8(0 0 0)
(make-uvector <s8vector> 5 -1) ⇒ #s8(-1 -1 -1 -1 -1)
```

@vector-unfold *f len seed* [Function]
@vector-unfold-right *f len seed* [Function]

{gauche.uvector} Construct a @vector of length *len*, with each element as a result of `(f seed)`, `(f (f seed))`, `(f (f (f seed)))`, ... @vector-unfold fills the element from left to right, while @vector-unfold-right from right to left.

```
(u8vector-unfold (cut + 2 <>) 5 0)
 ⇒ #u8(2 4 6 8 10)
(u8vector-unfold-right (pa$ + 2) 5 0)
 ⇒ #u8(10 8 6 4 2)
```

@vector-length *vec* [Function]

[R7RS vector.@] {gauche.uvector} Returns the length of the @vector *vec*.

Note that the generic function `size-of` can be used to obtain the length of *vec* as well, if you import `gauche.collection` (see Section 9.5 [Collection framework], page 344).

```
(s16vector-length '#s16(111 222 333)) ⇒ 3

(use gauche.collection)
(size-of '#s16(111 222 333)) ⇒ 3
```

uvector-length *uvector* [Function]

{gauche.uvector} This is a generic version of `@vector-length`; you can pass any instance of uniform vector, and it returns the number of its elements.

uvector-size *uvector :optional start end* [Function]

{gauche.uvector} This function can be applied to any type of uniform vectors, and returns the raw size of the *uvector* in number of octets.

When *start* and/or *end* is/are given, the size of data between those indices are calculated. The special value `-1` for *end* indicates the end of the vector. The returned value matches the number of octets to be written out by `(write-uvector uvector port start end)`.

(Do not confuse this with `uvector-length`, which returns the number of elements.)

```
(uvector-size '#u8(1 2 3))        ⇒ 3
```

```
(uvector-size '#u64(1 2 3))        ⇒ 24

(uvector-size '#u32(0 1 2 3) 2)    ⇒ 8
(uvector-size '#u32(0 1 2 3) 0 1) ⇒ 4
```

`uvector-class-element-size` *class* [Function]

{`gauche.uvector`} Returns the size of an element of a uvector of the given class, in bytes. An error is raised when *class* is not a uvector class.

```
(uvector-class-element-size <u8vector>)  ⇒ 1
(uvector-class-element-size <s64vector>) ⇒ 8
```

`@vector-ref` *vec k :optional fallback* [Function]

[R7RS vector.@] {`gauche.uvector`} Returns the *k*-th element of @vector *vec*.

If the index *k* is out of the valid range, an error is signaled unless an optional argument *fallback* is given; in that case, *fallback* is returned.

Note that the generic function `ref` can be used as well, if you import `gauche.collection`.

```
(u16vector-ref '#u16(111 222 333) 1) ⇒ 222

(use gauche.collection)
(ref '#u16(111 222 333) 1) ⇒ 222
```

`uvector-ref` *vec k :optional fallback* [Function]

{`gauche.uvector`} Generic version of `@vector-ref`. It can take any kind of uniform vector to *vec*, and returns its *k*-th element. If the index *k* is out of the valid range, an error is signaled unless an optional argument *fallback* is given; in that case, *fallback* is returned.

This is handy to write a generic code that works on any kind of uniform vector, but this is slower than the specific versions. Gauche's compiler recognizes the specific versions of referencer and generate very efficient code for them, while this generic version becomes a normal procedure call. In inner-loop it can make a big difference.

`(setter uvector-ref)` is `uvector-set!`.

`@vector-set!` *vec k n :optional clamp* [Function]

[R7RS vector.@] {`gauche.uvector`} Sets a number *n* to the *k*-th element of @vector *vec*. Optional *clamp* argument specifies the behavior when *n* is out of valid range. Default is to signal an error.

Note that the setter of the generic function `ref` can be used as well, if you import `gauche.collection`.

```
(let ((v (s32vector -439 852 8933)))
  (s32vector-set! v 1 4)
  v)
 ⇒ #s32vector(-439 4 8933)

(use gauche.collection)
(let ((v (s32vector -439 852 8933)))
  (set! (ref v 1) 4)
  v)
 ⇒ #s32vector(-439 4 8933)
```

`uvector-set!` *vec k val* [Function]

{`gauche.uvector`} Generic version of `@vector-set!`. It can handle any kind of uniform vectors, but a bit slower than the specific versions.

`@vector-swap!` *vec i j* [Function]
[R7RS vector.@] {`gauche.uvector`} Interchanges *i*th and *j*th elements of the uvector *vec*. Return value is not specified.

`@vector-fill!` *vec fill :optional start end* [Function]
{`gauche.uvector`} Stores *fill* in every element of *vec*, ranging from *start* to *end* of *vec*, if they are given. Return value is not specified.

`@vector=` *vec1 ...* [Function]
[R7RS vector.@] {`gauche.uvector`} All arguments must be @vectors. Returns `#t` iff all arguments have the same length and has the same values (in terms of `=`) at the corresponding position. Zero arguments return *#t*.

Note that in Gauche you can compare uvectors with `equal?`as well.

`@vector=?` *vec1 vec2* [Function]
[SRFI-66] {`gauche.uvector`} Note: This is provided only for the srfi-66 compatibility. Use `@vector=` instead.

Both arguments must be a @vector. Returns `#t` if *vec1* and *vec2* are equal to each other, `#f` otherwise.

`@vector-compare` *vec1 vec2* [Function]
[SRFI-66] {`gauche.uvector`} Both arguments must be a @vector. Returns `-1` if *vec1* is smaller than *vec2*, `0` if both are equal to each other, and `1` if *vec1* is greater than *vec2*.

Shorter vector is smaller than longer vectors. If the lengths of both vectors are the same, elements are compared from left to right.

Note that you can compare uvectors with `compare` in Gauche. These are provided because SRFI-66 defines `u8vector-compare`. You can also use them to indicate arguments are vectors of the specific type.

`@vector-copy` *vec :optional start end* [Function]
[R7RS vector.@] {`gauche.uvector`} Returns a fresh copy of uniform vector *vec*. If *start* and/or *end* are given, they limit the range of *vec* to be copied.

```
(u8vector-copy '#u8(1 2 3 4))      ⇒ #u8(1 2 3 4)
(u8vector-copy '#u8(1 2 3 4) 2)    ⇒ #u8(3 4)
(u8vector-copy '#u8(1 2 3 4) 1 3) ⇒ #u8(2 3)
```

`uvector-copy` *vec :optional start end* [Function]
{`gauche.uvector`} This is a generic version of `@vector-copy`. You can give any type of uvector to *vec*, and get its copy (or copy of its part, depending on *start*/*end* argument).

`@vector-reverse-copy` *vec :optional start end* [Function]
[R7RS vector.@] {`gauche.uvector`} Copies *vec* between *strat* and *end* index, but reversing it.

```
(u8vector-reverse-copy '#u8(1 2 3 4 5))
  ⇒ #u8(5 4 3 2 1)

(u8vector-reverse-copy '#u8(1 2 3 4 5) 1 4)
  ⇒ #u8(4 3 2)
```

`@vector-copy!` *target tstart source :optional sstart send* [Function]
[R7RS vector.@] {`gauche.uvector`} Both *target* and *source* must be @vectors, and *target* must be mutable. This procedure copies the elements of *source*, beginning from index *sstart*

(inclusive) and up to *send*, into *target*, beginning from index *tstart*. *sstart* and *send* may be omitted, and in that case 0 and the length of *source* are assumed, respectively.

```
(let ((target (u8vector 0 1 2 3 4 5 6)))
  (u8vector-copy! target 2 '#u8(10 11 12 13 14) 1 4)
  target)
 ⇒ #u8(0 1 11 12 13 6)
```

If the number of elements in the source vector between *sstart* and *send* is larger than the target vector beginning from *tstart*, the excess elements are silently discarded.

It is ok to pass the same vector to *target* and *source*; it always works even if the regions of source and destination are overlapping.

Note: This procedure used to take just two uniform vectors, *target* and *source*, and just copies contents of *source* to *target*. Both vectors had to be the same type and same length. The API is revised according to srfi-160. The old interface is still supported for the backward compatibility, but it is deprecated and will be gone in the future releases.

Also note that SRFI-66 provides `uvector-copy!` with different argument order (see Section 11.16 [Octet vectors], page 618).

`@vector-multi-copy!` *target tstart tstride source :optional sstart ssize sstride count* [Function]

{`gauche.uvector`} This procedure allows different parts of the source uvector *source* into various parts of the target uvector *target*, all at once.

When *ssize* is omitted or zero, this procedure does the following:

```
;; For each i from 0 to count:
(u8vector-copy! target (+ tstart (* i tstride))
                source sstart)
```

That is, it copies the content of *source* (offset by *sstart*, which defaults to 0) into the *target* repeatedly, advancing index with *tstride*. If either the target index reaches the end or *count* copies are made, the procedure returns. See the example:

```
(define t (make-u8vector 10 0))
(u8vector-multi-copy! t 0 4 '#u8(1 2 3))

t ⇒ #u8(1 2 3 0 1 2 3 0 1 2)
```

If *ssize* is given and positive, the source is also splitted as follows:

```
;; For each i from 0 to count:
(u8vector-copy! target (+ tstart (* i tstride))
                source (+ sstart (* i sstride))
                       (+ sstart (* i sstride) ssize))
```

That is, each *ssize* slice from *source*, is copied into *target*, advancing source index by *sstride* and the destination index by *dstride*. In this case, *sstride* defaults to *ssize* if omitted.

```
(define t (make-u8vector 12 0))
(u8vector-multi-copy! t 0 4 '#u8(1 2 3 4 5 6 7 8 9) 0 3)

t ⇒ #u8(1 2 3 0 4 5 6 0 7 8 9 0)
```

The operation ends when either *count* slices are copied, or destination index or source index reaches the end.

Hint: If you want to copy a part of the source vector repeatedly (instead of to its end), you can specify 0 to *sstride*:

```
(define t (make-u8vector 12 0))
(u8vector-multi-copy! t 0 4 '#u8(1 2 3 4 5 6 7 8 9) 2 4 0)
```

```
t ⇒ #u8(3 4 5 6 3 4 5 6 3 4 5 6)
```

Using collection and sequence framework, you can perform various operations on the homogeneous vectors.

```
(use gauche.collection)
(use gauche.sequence)

(fold + 0 '#s32(1 2 3 4)) ⇒ 10

(map-to <f32vector> * '#f32(3.2 1.1 4.3) '#f32(-4.3 2.2 9.4))
  ⇒ #f32(-13.760001 2.420000 40.420002)

(subseq #u32(1 4 3 4 5) 2 4) ⇒ #u32(3 4)
```

`uvector-copy!` *target tstart source :optional sstart send* [Function]

{`gauche.uvector`} This is a generic version of `@vector-copy!`. The destination *target* and the source *source* can be any type of uniform vectors, and they don't need to match. The copy is done bit-by-bit. So if you copy to a different type of uvector, the result depends on how the numbers are represented internally. This is mainly to manipulate binary data.

Tstart is interpreted according to the type of *target*, and *sstart* and *send* are interpreted according to the type of *source*.

```
(rlet1 v (make-u8vector 6 0)
  (uvector-copy! v 1 '#u32(0 #x01020304 0) 1 2))
 ⇒ #u8(0 1 2 3 4 0) or #u8(0 4 3 2 1 0)
```

`@vector-append` *vec . . .* [Function]

[R7RS vector.@] {`gauche.uvector`} All arguments must be @vectors. Returns a fresh vector whose contents are concatenation of the given vectors. (It returns a fresh vector even there's only one argument).

```
(u8vector-append '#u8(1 2 3) '#u8(4 5) '#u8() '#u8(6 7 8))
  ⇒ #u8(1 2 3 4 5 6 7 8)
```

`@vector-concatenate` *vecs* [Function]

[R7RS vector.@] {`gauche.uvector`} Returns a new @vector which is concatenation of the list of @vectors *vecs*.

```
(u8vector-concatenate '(#u8(1 2 3) #u8(4 5 6)))
  ⇒ #u8(1 2 3 4 5 6)
```

`@vector-append-subvectors` *:optional vec start end . . .* [Function]

[R7RS vector.@] {`gauche.uvector`} Returns a new @vector which is concatenation of the subvectors of given *vecs*, using accompanied *start* and *end* index.

```
(u8vector-append-subvectors '#u8(1 2 3 4) 1 3 '#u8(5 6 7 8) 0 2)
  ⇒ #u8(2 3 5 6)
```

`@vector-comparator` [Variable]

[R7RS vector.@] {`gauche.uvector`} Bound to comparators that can compare two `@vectors` and to hash a `@vector`. See Section 6.2.4 [Basic comparators], page 103, for the details of comparators. These comparators both provides ordering predicate and hash function.

`uvector-binary-search` *uvector key :optional start end skip rounding* [Function]

{`gauche.uvector`} The `uvector` must contain values in increasing order. This procedure finds the index of an element that is equal to *key*, using binary search. If such element can't be found, `#f` is returned.

```
(uvector-binary-search '#u8(0 5 19 32 58 96) 32)
  ⇒ 3

(uvector-binary-search '#u8(0 5 19 32 58 96) 33)
  ⇒ #f
```

The optional *start* and *end* arguments limits the portion of *uvector* to search; *start* specifies starting index (inclusive) and *end* specifies ending index (exclusive). Passing `#f` indicates the default value (0 for *start*, the length of the vector for *end*). The returned index is the actual index of the vector, but the elements outside of *start*-*end* range don't need to be sorted.

```
(uvector-binary-search '#u8(99 99 19 32 58 99) 32 2 5)
  ⇒ 3

(uvector-binary-search '#u8(99 99 19 32 58 99) 99 2 5)
  ⇒ #f
```

The optional *skip* argument must be a nonnegative exact integer or `#f`. If it is a positive integer, the number of elements after every key in the *uvector* is ignored. For example, if *skip* is 2 and *uvector* is `#u8(3 100 101 5 102 103 13 104 105)`, only `3`, `5` and `13` are subject to search, and elements inbetween are ignored. This allows the caller to store *payload*, or associated value to each key, in the *uvector* itself. If *skip* is positive integer, the length of the searched portion of *uvector* must be a multiple of the record size (*skip*+1).

```
(uvector-binary-search '#u8(3 100 101 5 102 103 13 104 105) 13 #f #f 2)
  ⇒ 6

(uvector-binary-search '#u8(3 100 101 5 102 103 13 104) 13 #f #f 2)
  ⇒ ; Error: uvector size (8) isn't multiple of record size (3)
```

Finally, *rounding* argument adjusts the behavior when the exact match isn't found. It can be either one of the following values:

`#f`
: This is the default. The procedure searches the element that is equal to *key*, and returns `#f` if such element isn't found.

a symbol `floor`
: When the exact match isn't found, the procedure returns an index of the element that's closest to but not greater than *key*. If *key* is smaller than all the elements, `#f` is returned.

a symbol `ceiling`
: When the exact match isn't found, the procedure returns an index of the element that's closest to but not smaller than *key*. If *key* is greater than all the elements, `#f` is returned.

```
(uvector-binary-search '#u32(1 10 100 1000 10000) 3757)
  ⇒ #f

(uvector-binary-search '#u32(1 10 100 1000 10000) 3757 #f #f #f 'floor)
  ⇒ 3

(uvector-binary-search '#u32(1 10 100 1000 10000) 3757 #f #f #f 'ceiling)
  ⇒ 4
```

Note: SRFI-133 has `vector-binary-search`, which is quite similar to this procedure (see Section 11.30 [Vector library], page 632) but it requires comparison procedure, for it needs to compare general Scheme values. And it does not support *skip* and *rounding* arguments.

9.36.2 Uvector conversion operations

`@vector->list` *vec :optional start end* [Function]

[R7RS vector.@] {`gauche.uvector`} Converts @vector *vec* to a list. If *start* and/or *end* are given, they limit the range of *vec* to be extracted.

Note that the generic function `coerce-to` can be used as well, if you import `gauche.collection`.

```
(u32vector->list '#u32(9 2 5)) ⇒ (9 2 5)

(use gauche.collection)
(coerce-to <list> '#u32(9 2 5)) ⇒ (9 2 5)
```

`uvector->list` *uvec :optional start end* [Function]

{`gauche.uvector`} This is a generic version of `@vector->list`. It can take any kind of uvector as *uvec*. The meaning of optional arguments are the same as `@vector->list`.

`@vector->vector` *vec :optional start end* [Function]

[R7RS vector.@] {`gauche.uvector`} Converts @vector *vec* to a vector. If *start* and/or *end* are given, they limit the range of *vec* to be copied.

Note that the generic function `coerce-to` can be used as well, if you import `gauche.collection`.

```
(f32vector->vector '#f32(9.3 2.2 5.5))   ⇒ #(9.3 2.2 5.5)
(f32vector->vector '#f32(9.3 2.2 5.5) 2) ⇒ #(5.5)

(use gauche.collection)
(coerce-to <vector> '#f32(9.3 2.2 5.5)) ⇒ #(9.3 2.2 5.5)
```

`uvector->vector` *uvec :optional start end* [Function]

{`gauche.uvector`} This is a generic version of `@vector->vector`. It can take any kind of uvector as *uvec*. The meaning of optional arguments are the same as `@vector->vector`.

`list->@vector` *list :optional clamp* [Function]

[R7RS vector.@] {`gauche.uvector`} Converts a list *list* to a @vector. Optional argument *clamp* specifies the behavior when the element of *list* is out of the valid range. (The *clamp* argument is Gauche's extension.)

Note that the generic function `coerce-to` can be used as well, if you import `gauche.collection`.

```
(list->s64vector '(9 2 5)) ⇒ #s64(9 2 5)

(use gauche.collection)
(coerce-to <s64vector> '(9 2 5)) ⇒ #s64(9 2 5)
```

`reverse-list->@vector` *list :optional clamp* [Function]

[R7RS vector.@] {`gauche.uvector`} Create a new `@vector` with the elements of *list* in reverse order. Optional argument *clamp* specifies the behavior when the element of *list* is out of the valid range. (The *clamp* argument is Gauche's extension.)

`vector->@vector` *vec :optional start end clamp* [Function]

[R7RS vector.@] {`gauche.uvector`} Converts a vector *vec* to a @vector. If *start* and/or *end* are given, they limit the range of *vec* to be copied. Optional argument *clamp* specifies the behavior when the element of *vec* is out of the valid range. (The *clamp* argument is Gauche's extension.)

Note that the generic function `coerce-to` can be used as well, if you import `gauche.collection`.

```
(vector->f64vector '#(3.1 5.4 3.2)) ⇒ #f64(3.1 5.4 3.2)

(use gauche.collection)
(coerce-to <f64vector> '#(3.1 5.4 3.2)) ⇒ #f64(3.1 5.4 3.2)
```

`string->s8vector` *string :optional start end immutable?* [Function]
`string->u8vector` *string :optional start end immutable?* [Function]

{`gauche.uvector`} Returns an s8vector or u8vector whose byte sequence is the same as the internal representation of the given string. Optional range arguments *start* and *end* specifies the *character position* (not the byte position) inside *string* to be converted.

By default, the content of the string is copied to a newly created mutable uvector. However, if a true value is given to the optional *immutable?* argument, the result is an immutable uvector, and it may avoid copying the string body (note that in Gauche, the body of string is immutable; `string-set!` creates a new body, so changing the original string won't affect the uvector created by `string->u8vector` with `immutable?` flag.)

These procedures are useful when you want to access byte sequence of the string randomly.

```
(string->u8vector "abc") ⇒ #u8(97 98 99)

(string->u8vector "very large string .... " 0 -1 #t)
  ⇒ #u8(...)  ; immutable, sharing content with the original string
```

`string->s8vector!` *target tstart string :optional start end* [Function]
`string->u8vector!` *target tstart string :optional start end* [Function]

{`gauche.uvector`} *Target* must be an s8vector or a u8vector, respectively. *Target* must be mutable. Like copies the raw byte representation of *string* into *target* beginning from index *tstart*.

Returns *target*.

```
(let ((target (make-u8vector 10 0)))
  (string->u8vector! target 3 "abcde"))
 ⇒ #u8(0 0 0 97 98 99 100 101 0 0)
```

`s8vector->string` *vec :optional start end terminator* [Function]
`u8vector->string` *vec :optional start end terminator* [Function]

{`gauche.uvector`} Converts a byte sequence in s8vector or u8vector to a string that has the same byte sequence. Optional range arguments *start* and *end* specifies the byte position in *vec* to be converted.

The optional *terminator* argument can be an exact integer or `#f` (default). If it is an exact integer, and it appears in *vec*, the string terminates right before it. For example, you can give 0 as *terminator* to read a NUL-terminated string from a buffer.

```
(u8vector->string '#u8(65 66 0 67 68) 0 5)   ⇒ "AB\0CD"
(u8vector->string '#u8(65 66 0 67 68) 0 5 0) ⇒ "AB"
```

Note that these procedure may result an incomplete string if *vec* contains a byte sequence invalid as the internal encoding of the string.

`string->s32vector` *string :optional start end* [Function]
`string->u32vector` *string :optional start end* [Function]

{`gauche.uvector`} Returns an s32vector or u32vector whose elements are the internal codes of the characters in the string. Optional range arguments *start* and *end* specifies the *character position* inside *string* to be converted.

These procedures are useful when you want to access the characters in the string randomly.

`string->s32vector!` *target tstart string :optional start end* [Function]
`string->u32vector!` *target tstart string :optional start end* [Function]

{`gauche.uvector`} *Target* must be a mutable s32vector or u32vector, respectively. Fill the *target* from position *tstart* with the codepoint of each character of *string*, until either *string* is exhausted or *target* is filled to the end.

Optional range arguments *start* and *end* specifies the *character position* inside *string* to be considered.

`s32vector->string` *vec :optional start end terminator* [Function]
`u32vector->string` *vec :optional start end terminator* [Function]

{`gauche.uvector`} Without *start* and *end*, these procedures work like this:

```
(lambda (vec) (map-to <string> integer->char vec)))
```

Optional range arguments *start* and *end* limits the range of conversion between them.

The optional *terminator* argument must be an exact integer or `#f` (default). If an integer is given, and the integer is found in the input, the output string terminates right before it.

```
(u32vector->string '#u32(65 66 0 67 68) 0 5 0) ⇒ "AB"
```

`uvector-alias` *uvector-class vec :optional start end* [Function]

{`gauche.uvector`} This procedure creates an uvector of class *uvector-class* that shares the storage of the given uniform vector *vec*. If optional *start* and *end* arguments are given, only the specified range of *vec* is used for the new vector. Since the storage is shared, modification of the original vector can be seen from the new vector, or vice versa.

The class *uvector-class* must be either one of the uniform vector class, but is not necessary match the class of the source vector *vec*. In such case, the new vector looks at the same region of *vec*'s memory, but interprets it differently. For example, the following code determines whether Gauche is running on big-endian or little-endian machine:

```
(let ((u8v (uvector-alias <u8vector> #u32(1))))
  (if (zero? (u8vector-ref u8v 0))
      'big-endian
      'little-endian))
```

If the *uvector-class* is other than s8vector or u8vector, the region the new vector points has to meet the alignment requirement. You can assume the beginning of the source vector is aligned suitable for any uniform vectors. So, for example, if you're creating u32vector from u8vector, the *start* and *end* must be multiple of 4 (or, if they're omitted, the length of the original u8vector must be multiple of 4). An error is signaled when the given parameters doesn't satisfy alignment constraint.

9.36.3 Uvector numeric operations

These are Gauche extension that allows faster arithmetic over uniform vectors, than extracting and calculating element-wise values.

Most procedures comes with two flavors, a functional version (without `!` in the name) and a linear-update version (with `!` in the name).

A functional version assumes the caller treats the arguments and results immutable objects; mutating them later could have unexpected consequences. (Notably, the functional version may return one of its arguments as is, or returns a pre-computed value, so you shouldn't assume the return values are freshly allocated objects, unless it is noted so explicitly.)

A linear update version may reuse the storage of the designated argument to produce the return value. Gauche tries to reuse the argument as much as possible, but you should always use the return value and shouldn't assume the argument itself is modified in-place. In fact, after calling linear-updating procedure, you can't use the argument that may be modified, since you can't assume the state of the object after calling the procedure.

`@vector-add` *vec val :optional clamp* [Function]
`@vector-add!` *vec val :optional clamp* [Function]
`@vector-sub` *vec val :optional clamp* [Function]
`@vector-sub!` *vec val :optional clamp* [Function]
`@vector-mul` *vec val :optional clamp* [Function]
`@vector-mul!` *vec val :optional clamp* [Function]

{`gauche.uvector`} Element-wise arithmetic. *Vec* must be a @vector, and *val* must be either a @vector, a vector, or a list of the same length as *vec*, or a number (an exact integer for integer vectors, and a real number for f32- and f64-vectors).

If *val* is a @vector, its elements are added to, subtracted from, or multiplied by the corresponding elements of *vec*, respectively, and the results are gathered to a @vector and returned. The linear-update version (those have bang '!' in the name) reuses *vec* to store the result, and also returns it. If the result of calculation goes out of the range of @vector's element, the behavior is specified by *clamp* optional argument. (For f32vector and f64vector, *clamp* argument is ignored and the result may contain infinity).

If *val* is a number, it is added to, subtracted from, or multiplied by each element of *vec*, respectively.

```
(s8vector-add '#s8(1 2 3 4) '#s8(5 6 7 8)) ⇒ #s8(6 8 10 12)
(u8vector-sub '#u8(1 2 3 4) '#u8(2 2 2 2)) ⇒ error
(u8vector-sub '#u8(1 2 3 4) '#u8(2 2 2 2) 'both) ⇒ #u8(0 0 1 2)

(f32vector-mul '#f32(3.0 2.0 1.0) 1.5) ⇒ #f32(4.5 3.0 1.5)
```

`@vector-div` *vec val* [Function]
`@vector-div!` *vec val* [Function]

{`gauche.uvector`} Element-wise division of flonum vectors. These are only defined for f16, f32 and f64vector. *val* must be a @vector, a vector or a list of the same length as *vec*, or a real number.

```
(f32vector-div '#f32(1.0 2.0 3.0) 2.0) ⇒ #f32(0.5 1.0 1.5)
```

`@vector-and` *vec val* [Function]
`@vector-and!` *vec val* [Function]
`@vector-ior` *vec val* [Function]
`@vector-ior!` *vec val* [Function]
`@vector-xor` *vec val* [Function]
`@vector-xor!` *vec val* [Function]

{`gauche.uvector`} Element-wise logical (bitwise) operation. These procedures are only defined for integral vectors. *val* must be a @vector, a vector or a list of the same length as *vec*, or an exact integer. Bitwise and, inclusive or or exclusive or is calculated between each element in *vec* and the corresponding element of *val* (when *val* is a non-scalar value), or *val* itself (when *val* is an integer). The result is returned in a @vector. The linear-update version reuses *vec* to store the result, and also returns it.

`@vector-dot` *vec0 vec1* [Function]

{`gauche.uvector`} Calculates the dot product of two @vectors. The length of *vec0* and *vec1* must be the same.

`@vector-range-check` *vec min max* [Function]

{`gauche.uvector`} *Vec* must be a @vector, and each of *min* and *max* must be either a @vector, a vector or a list of the same length as *vec*, or a number, or `#f`.

For each element in *vec*, this procedure checks if the value is between *minval* and *maxval* inclusive, where *minval* and *maxval* are the corresponding values of *min* and *max* (when *min* and/or *max* is/are non-scalar value) or *min* and *max* themselves (when *min* and/or *max* is/are a number). When *min* is `#f`, negative infinity is assumed. When *max* is `#f`, positive infinity is assumed.

If all the elements in *vec* are within the range, `#f` is returned. Otherwise, the index of the leftmost element of *vec* that is out of range is returned.

```
(u8vector-range-check '#u8(3 1 0 2) 0 3)  ⇒ #f
(u8vector-range-check '#u8(3 1 0 2) 1 3)  ⇒ 2

(u8vector-range-check '#u8(4 32 64 98) 0 '#u8(10 40 70 90))
  ⇒ 3

;; Range check in a program
(cond
 ((u8vector-range-check u8v 1 31)
  => (lambda (i)
      (errorf "~sth vector element is out of range: ~s"
              i (u8vector-ref u8v i))))
 (else (do-something u8v)))
```

`@vector-clamp` *vec min max* [Function]
`@vector-clamp!` *vec min max* [Function]

{`gauche.uvector`} *Vec* must be a @vector, and each of *min* and *max* must be either a @vector, a vector or a list of the same length as *vec*, or a number, or `#f`.

Like @vector-range-check, these procedures check if each element of *vec* are within the range between *minval* and *maxval* inclusive, which are derived from *min* and *max*. If the value is less than *minval*, it is replaced by *minval*. If the value is grater than *maxval*, it is replaced by *maxval*.

@vector-clamp creates a copy of *vec* and do clamp operation on it, while @vector-clamp! modifies *vec*. Both return the clamped vector.

```
(s8vector-clamp '#s8(8 14 -3 -22 0) -10 10) ⇒ #s8(8 10 -3 -10 0)
```

9.36.4 Uvector block I/O

A uniform vector can be seen as an abstraction of a chunk of memory. So you might want to use it for binary I/O. Yes, you can do it.

`read-uvector` *class size :optional iport endian* [Function]

{`gauche.uvector`} Reads *size* elements of uvector of class *class* from *iport*, and returns fleshly created uvector. If *iport* is omitted, the curret input port is used.

For example, you can read input as an octet stream as follows:

```
(with-input-from-string "abcde"
  (^[] (read-uvector <u8vector> 5)))
 ⇒ #u8(97 98 99 100 101)
```

If the input port has already reached EOF, an EOF object is returned. The returned uvector can be shorter than *size* if the input reaches EOF before *size* elements are read.

If the *iport* is a buffered port with 'modest' or 'none' buffering mode (see Section 6.22.4 [File ports], page 221), `read-uvector` may return before *size* elements are read, even if *iport* hasn't reached EOF. The ports connected to a pipe or a network socket behave so by default.

The data is read as a byte stream, so if you give uniform vectors other than s8vector or u8vector, your result may affected by the endianness. If the optional argument *endian* is given, the input is interpreted in that endianness. When omitted, the value of the parameter default-endian is used. See Section 6.3.7 [Endianness], page 123, for more about endian handling.

If the size of the input data is unknown and you need to read everything until EOF, use port->uvector below.

R7RS has read-bytevector; it is the same as passing <u8vector> to read-uvector.

read-uvector! *vec :optional iport start end endian* [Function]

{gauche.uvector} Reads a chunk of data from the given input port *iport*, and stores it to the uniform vector *vec*. You can give any uniform vector. If optional *start* and *end* arguments are given, they specify the index range in *vec* that is to be filled, and the rest of the vector remains untouched. Otherwise, entire vector is used. A special value -1 for *end* indicates the end of *vec*. If *iport* is omitted, the current input port is used.

If the input reached EOF before the required region of *vec* is filled, the rest of the vector is untouched.

If *iport* is already reached EOF when read-uvector! is called, an EOF object is returned. Otherwise, the procedure returns the number of *elements* read (not bytes).

If the *iport* is a buffered port with 'modest' or 'none' buffering mode (see Section 6.22.4 [File ports], page 221), read-uvector! may return before all the elements in *vec* is filled, even if *iport* hasn't reached EOF. The ports connected to a pipe or a network socket behave so by default. If you know there will be enough data arriving and want to make sure *vec* is filled, change the buffering mode of *iport* to 'full'.

The data is read as a byte stream, so if you give uniform vectors other than s8vector or u8vector, your result may affected by the endianness. If the optional argument *endian* is given, the input is interpreted in that endianness. When omitted, the value of the parameter default-endian is used. See Section 6.3.7 [Endianness], page 123, for more about endian handling.

read-block! *vec :optional iport start end endian* [Function]

{gauche.uvector} An old name of read-uvector!. Supported for the backward compatibility, but new code should use read-uvector!.

port->uvector *iport :optional class* [Function]

{gauche.uvector} Read data from the input port *iport* until EOF and store them into a uvector of *class*. If *class* is omitted, <u8vector> is used.

If you specify a class of uvector whose element is more than an octet, the input data is packed with platform's native byteorder.

This procedure is parallel to port->string etc. (see Section 6.22.7.4 [Input utility functions], page 231).

write-uvector *vec :optional oport start end endian* [Function]

{gauche.uvector} Writes out the content of the uniform vector *vec* 'as is' to the output port *oport*. If *oport* is omitted, the current output port is used. If optional *start* and *end* arguments are given, they specify the index range in *vec* to be written out. A special value -1 for *end* indicates the end of *vec*. This procedure returns an unspecified value.

If you write out a uniform vector except s8vector and u8vector, the care should be taken about the endianness, as in read-uvector. The optional argument *endian* specifies the output endian. When it is omitted, the value of the parameter default-endian is used (see Section 6.3.7 [Endianness], page 123).

write-block *vec :optional oport start end endian* [Function]

{gauche.uvector} An old name of write-uvector. Supported for the backward compatibility, but new code should use write-uvector.

9.36.5 Bytevector compatibility

R7RS-small includes bytevectors in its core (`scheme.base`). In Gauche, bytevectors are the same as u8vectors.

The basic R7RS bytevector procedures are provided in this module. Conversion between bytevectors and strings are provided in `gauche.unicode` (see Section 9.35.1 [Unicode transfer encodings], page 472).

`bytevector` *byte . . .* [Function]

{`gauche.uvector`} [R7RS base] Alias of `u8vector`. Returns a fresh bytevector (u8vector) with *byte* . . . as its elements.

`bytevector?` *obj* [Function]

{`gauche.uvector`} [R7RS base] Alias of `u8vector?`. Returns true iff *obj* is a bytevector (u8vector).

`make-bytevector` *len :optional byte* [Function]

{`gauche.uvector`} [R7RS base] Alias of `make-u8vector`. Returns a fresh bytevector (u8vector) of length *len*. All elements are initialized by *byte* if given.

`bytevector-length` *bv* [Function]

{`gauche.uvector`} [R7RS base] Alias of `u8vector-length`. Returns the length of the bytevector (u8vector) *bv*.

`bytevector-u8-ref` *bv k* [Function]
`bytevector-u8-set!` *bv k byte* [Function]

{`gauche.uvector`} [R7RS base] Alias of `u8vector-ref` and `u8vector-set!`. Read and write *k*-th element of a bytevector (u8vector) *bv*.

It is an error to give out-of-bound index.

The return value of `bytevector-u8-set!` is unspecified.

As Gauche's extension, `(setter bytevector-u8-ref)` is `bytevector-u8-set!`.

`bytevector-copy` *bv :optional start end* [Function]

{`gauche.uvector`} [R7RS base] Alias of `u8vector-copy`. Returns a fresh copy of a bytevector (u8vector) *bv*. Optionally you can restrict the range of the source vector by indices *start* (inclusive) and *end* (exclusive).

`bytevector-copy!` *target tstart source :optional sstart send* [Function]

{`gauche.uvector`} [R7RS base] Alias of `u8vector-copy!`. Both *target* and *source* muse be bytevectors (u8vectors), and *target* must be mutable. Copy the content of *source* (optionally restricting the range between indices *start* (inclusive) and *end* (exclusive)) into *target* starting at the index *tstart*.

`bytevector-append` *bv . . .* [Function]

{`gauche.uvector`} [R7RS base] Alias of `u8vector-append`. All arguments must be bytevectors (u8vectors). Returns a fresh bytevector whose elements are the concatenation of elements of *bv* . . . .

`bytevector=?` *bv1 bv2* [Function]

{`gauche.uvector`} Alias of `u8vector=?`. All arguments must be bytevectors (u8vectors). Returns `#t` iff all bytevectors are of the same size and content.

9.37 gauche.version - Comparing version numbers

gauche.version [Module]

This module provides a convenient procedure to compare *version numbers* or *revision numbers*, such as "0.5.1", "3.2-3" or "8.2pl1". Usually each release of software component has a version number, and you can define order between them. For example, version "1.2.3" is newer than "1.2" and older than "2.1". You can compare those version numbers like this:

```
(version<? "2.2.3" "2.2.11")     ⇒ #t
(version<? "2.3.1" "2.3")        ⇒ #f
(version<? "2.3.1-1" "2.3.1-10") ⇒ #t
(version<? "13a" "5b")           ⇒ #f
```

There are no standard way to name versions, so I chose one convention. This won't work for all possible variations, but I think it covers typical cases.

Strictly speaking, you can only define partial order between version numbers, for there can be branches. This module uses simple measure and just assumes the version numbers can be fully ordered.

The version number here is defined by the following syntax.

```
<version> : <principal-release>
          | <version> <post-subrelease>
          | <version> <pre-subrelease>
<principal-release> : <relnum>
<post-subrelease>   : [.-] <relnum>
<pre-subrelease>    : _ <relnum>?
<relnum>            : [0-9A-Za-z]+
```

Typically <relnum> is composed by numeric part and extension part. For example, "23a" is composed by an integer 23 and extension "a". If <relnum> doesn't begins with digits, we assume its numeric part is -1.

Then, the order of <relnum> is defined as follows:

1. If relnum A and relnum B have different numeric part, we ignore the extension and order them numerically, e.g. "3b" < "4a".
2. If relnum A and relnum B have the same numeric part, we compare extension by alphabetically, e.g. "4c" < "4d" and "5" < "5a".

Given the order of <relnum>, the order of version numbers are defined as follows:

1. Decompose each version number into a list of <principal-release> and subsequence subrelease components. We call each element of the list "release components".
2. If the first release component of both lists are the same, remove it from both. Repeat this until the head of the lists differ.
3. Now we have the following cases.
   1. Both lists are empty: versions are the same.
   2. One list (A) is empty and the other list (B) has post-subrelease at head: A is prior to B
   3. One list (A) is empty and the other list (B) has pre-subrelease at head: B is prior to A
   4. List A's head is post-subrelease and list B's head is pre-subrelease: B is prior to A
   5. Both lists have post-subrelease or pre-subrelease at head: compare their relnums.

Here are some examples:

```
"1" < "1.0" < "1.1" < "1.1.1" < "1.1.2" < "1.2" < "1.11"
```

```
"1.2.3" < "1.2.3-1" < "1.2.4"
"1.2.3" < "1.2.3a" < "1.2.3b"
"1.2_" < "1.2_rc0" < "1.2_rc1" < "1.2" < "1.2-pl1" < "1.2-pl2"
"1.1-patch112" < "1.2_alpha"
```

The reason of having <pre-subrelease> is to allow "release candidate" or "pre-release" version.

A trick: If you want "version 1.2 release or later", you can say (version<=? "1.2" v). This excludes prerelease versions such as 1.2_pre3. If you want "version 1.2 release or later", you can say (version<=? "1.2_" v), which includes 1.2_pre1 etc., but excludes anything below, such as 1.1.99999.

It is common if you want to specify acceptable versions with combination of conditions, e.g. "version 1.3 or later, except version 1.4.1" or "greater than version 1.1 and below 1.5". A *version spec* is an S-expression to represent that condition. You can use version-satisfy? to check if given version satisfies the spec.

The syntax of version spec is as follows.

```
<version-spec> : <version>
               | (<op> <version>)
               | (and <version-spec> ...)
               | (or <version-spec> ...)
               | (not <version-spec>)

<version> : version string
<op>      : = | < | <= | > | >=
```

version=? *ver1 ver2* [Function]
version<? *ver1 ver2* [Function]
version<=? *ver1 ver2* [Function]
version>? *ver1 ver2* [Function]
version>=? *ver1 ver2* [Function]

{gauche.version} Returns a boolean value depending on the order of two version number string *ver1* and *ver2*. If the arguments contain invalid strings as the defined version number, an error is signaled.

version-compare *ver1 ver2* [Function]

{gauche.version} Compares two version number strings *ver1* and *ver2*, and returns either -1, 0, or 1, depending whether *ver1* is prior to *ver2*, *ver1* is the same as *ver2*, or *ver1* is after *ver2*, respectively.

relnum-compare *rel1 rel2* [Function]

{gauche.version} This is lower-level procedure of version-compare. Compares two release numbers (relnums) *rel1* and *rel2*, and returns either -1, 0, or 1 depending whether *rel1* is prior to *rel2*, *rel1* is the same as *rel2*, or *rel1* is after *rel2*, respectively.

The following procedures are to check if a given version satisfies a version specification.

valid-version-spec? *spec* [Function]

{gauche.version} This is a syntax checker. Returns #t if *spec* is a valid version specification, #f otherwise. See gauche.version module description for the definition of version specification.

version-satisfy? *spec version* [Function]

{gauche.version} Returns #t if *version* satisfies a version specification *spec*, #f otherwise. See gauche.version module description for the definition of version specification.

9.38 gauche.vport - Virtual ports

`gauche.vport` [Module]

Virtual ports, or procedural ports, are the ports whose behavior can be programmed in Scheme.

This module provides two kinds of virtual ports: Fully virtual ports, in which every I/O operation invokes user-provided procedures, and virtual buffered ports, in which I/O operations are done on an internal buffer and user-provided procedures are called only when the buffer needs to be filled or flushed.

This module also provides virtual buffered ports backed up by a uniform vector, as an example of the feature.

Fully virtual ports

This type of virtual ports are realized by classes `<virtual-input-port>` and `<virtual-output-port>`. You can customize the port behavior by setting appropriate slots with procedures.

`<virtual-input-port>` [Class]

{`gauche.vport`} An instance of this class can be used as an input port. The behavior of the port depends on the settings of the instance slot values.

To work as a meaningful input port, at least either one of `getb` or `getc` slot must be set. Otherwise, the port returns EOF for all input requests.

`getb` [Instance Variable of `<virtual-input-port>`]

If set, the value must be a procedure that takes no arguments. Every time binary input is required, the procedure is called.

The procedure must return an exact integer between 0 and 255 inclusive, or `#f` or an EOF object. If it returns an integer, it becomes the value read from the port. If it returns other values, the port returns EOF.

If the port is requested a character input and it doesn't have the `getc` procedure, the port calls this procedure, possibly multiple times, to construct a whole character.

`getc` [Instance Variable of `<virtual-input-port>`]

If set, the value must be a procedure that takes no arguments. Every time character input is required, the procedure is called.

The procedure must return a character, `#f` or an EOF object. If it returns a character, it becomes the value read from the port. If it returns other values, the port returns EOF.

If the port is requested a binary input and it doesn't have the `getb` procedure, the port calls this procedure, then converts a character into a byte sequence, and use it as the binary value(s) read from the port.

`gets` [Instance Variable of `<virtual-input-port>`]

If set, the value must be a procedure that takes one argument, a positive exact integer. It is called when the block binary input, such as `read-uvector`, is requested. It must return a (maybe incomplete) string up to the specified size, or `#f` or EOF object. If it returns a null string, `#f` or EOF object, the port thinks it reached EOF. If it returns other string, it is used as the result of block read. It shouldn't return a string larger than the given size (Note: you must count size (bytes), not the number of characters). The reason of this procedure is efficiency; if this procedure is not provided, the port calls `getb` procedure repeatedly to prepare the block of data. In some cases, providing block input can be much more efficient (e.g. suppose you're reading from a block of memory chunk).

You can leave this slot unset if you don't need to take such advantage.

`ready` [Instance Variable of `<virtual-input-port>`]

If set, the value must be a procedure that takes one boolean argument. It is called when `char-ready?` or `byte-ready?` is called on the port. The value returned from your procedure will be the result of these procedures.

The boolean argument is `#t` if `char-ready?` is called, or `#f` if `byte-ready?` is called.

If unset, `char-ready?` and `byte-ready?` always return `#t` on the port

`close` [Instance Variable of `<virtual-input-port>`]

If set, the value must be a procedure that takes no arguments. It is called when the port is closed. Return value is discarded. You can leave this unset if you don't need to take an action when the port is closed.

This procedure may be called from a finalizer, so you have to be careful to write it. See the note on finalization below.

`seek` [Instance Variable of `<virtual-input-port>`]

If set, the value must be a procedure that takes two arguments, offset and whence. The meaning of them is the same as the arguments to `port-seek` (see Section 6.22.3 [Common port operations], page 218). The procedure must adjust the port's internal read pointer so that the next read begins from the new pointer. It should return the updated pointer (the byte offset from the beginning of the port).

If unset, call of `port-seek` and `port-tell` on this port will return `#f`.

Note that this procedure may be called for the purpose of merely querying the current position, with 0 as `offset` and `SEEK_CUR` as `whence`. If your port knows the read pointer but cannot move it, you can still provide this procedure, which returns the current pointer position for such queries and returns `#f` for other arguments.

`<virtual-output-port>` [Class]

{`gauche.vport`} An instance of this class can be used as an output port. The behavior of the port depends on the settings of the instance slot values.

To work as an output port, at least either one of `putb` or `putc` slot has to be set.

`putb` [Instance Variable of `<virtual-output-port>`]

If set, the value must be a procedure that takes one argument, a byte value (exact integer between 0 and 255, inclusive). Every time binary output is required, the procedure is called. The return value of the procedure is ignored.

If this slot is not set and binary output is requested, the port may signal an `<io-unit-error>` error.

`putc` [Instance Variable of `<virtual-output-port>`]

If set, the value must be a procedure that takes one argument, a character. Every time character output is required, the procedure is called. The return value of the procedure is ignored.

If this slot is not set but `putb` slot is set, the virtual port decomposes the character into a sequence of bytes then calls `putb` procedures.

`puts` [Instance Variable of `<virtual-output-port>`]

If set, the value must be a procedure that takes a (possibly incomplete) string. The return value of the procedure is ignored.

This is for efficiency. If this slot is not set, the virtual port calls `putb` or `putc` repeatedly to output a chunk of data. But if your code can perform chunked output efficiently, you can provide this procedure.

flush [Instance Variable of `<virtual-output-port>`]

If set, the value must be a procedure that takes no arguments. It is called when flushing a port is required (e.g. `flush` is called on the port, or the port is being closed).

This procedure is useful that your port does some sort of buffering, or needs to keep some state. If your port doesn't do stateful operation, you can leave this unset.

This procedure may be called from a finalizer, and needs a special care. See notes on finalizers below.

close [Instance Variable of `<virtual-output-port>`]

The same as `<virtual-input-port>`'s `close` slot.

seek [Instance Variable of `<virtual-output-port>`]

The same as `<virtual-input-port>`'s `seek` slot.

Virtual buffered ports

This type of virtual ports are realized by classes `<buffered-input-port>` and `<buffered-output-port>`. You can customize the port behavior by setting appropriate slots with procedures.

Those ports have internal buffer and only calls Scheme procedures when the buffer needs to be filled or flushed. Generally it is far more efficient than calling Scheme procedures for every I/O operation. Actually, the internal buffering mechanism is the same as Gauche's file I/O ports.

These ports uses `u8vector` as a buffer. See Section 9.36 [Uniform vectors], page 476, for the details.

<buffered-input-port> [Class]

{`gauche.vport`} An instance of this class behaves as an input port. It has the following instance slots. For a meaningful input port, you have to set at least `fill` slot.

fill [Instance Variable of `<buffered-input-port>`]

If set, it must be a procedure that takes one argument, a `u8vector`. It must fill the data from the beginning of the vector. It doesn't need to fill the entire vector if there's not so many data. However, if there are remaining data, it must fill at least one byte; if the data isn't readily available, it has to wait until some data becomes available.

The procedure must return a number of bytes it actually filled. It may return 0 or an EOF object to indicate the port has reached EOF.

ready [Instance Variable of `<buffered-input-port>`]

If set, it must be a procedure that takes no arguments. The procedure must return a true value if there are some data readily available to read, or `#f` otherwise. Unlike fully virtual ports, you don't need to distinguish binary and character I/O.

If this slot is not set, the port is regarded as it always has data ready.

close [Instance Variable of `<buffered-input-port>`]

If set, it must be a procedure that takes no arguments. The procedure is called when the virtual buffered port is closed. You don't need to set this slot unless you need some cleaning up when the port is closed.

This procedure may be called from a finalizer, and needs special care. See the note on finalization below.

filenum [Instance Variable of `<buffered-input-port>`]

If set, it must be a procedure that returns underlying file descriptor number (exact non-negative integer). The procedure is called when `port-file-number` is called on the port.

If there's no such underlying file descriptor, you can return `#f`, or you can leave this slot unset.

`seek` [Instance Variable of `<buffered-input-port>`]

If set, it must be a procedure that takes two arguments, *offset* and *whence*. It works the same way as `<virtual-input-port>`'s seek procedure; see above.

This procedure may be called from a finalizer, and needs special care. See the note on finalization below.

Besides those slot values, you can pass an exact nonnegative integer as the `:buffer-size` keyword argument to the *make* method to set the size of the port's internal buffer. If `:buffer-size` is omitted, or zero is passed, the system's default buffer size (something like 8K) is used. `:buffer-size` is not an instance slot and you cannot set it after the instance of the buffered port is created. The following example specifies the buffered port to use a buffer of size 64K:

```
(make <buffered-input-port> :buffer-size 65536 :fill my-filler)
```

`<buffered-output-port>` [Class]

{`gauche.vport`} An instance of this class behaves as an output port. It has the following instance slots. You have to set at least `flush` slot.

`flush` [Instance Variable of `<buffered-output-port>`]

If set, it must be a procedure that takes two arguments, an `u8vector` buffer and a flag. The procedure must output data in the buffer to somewhere, and returns the number of bytes actually output.

If the flag is false, the procedure may output less than entire buffer (but at least one byte). If the flag is true, the procedure must output entire buffer.

`close` [Instance Variable of `<buffered-output-port>`]

Same as `<buffered-input-port>`'s `close` slot.

`filenum` [Instance Variable of `<buffered-output-port>`]

Same as `<buffered-input-port>`'s `filenum` slot.

`seek` [Instance Variable of `<buffered-output-port>`]

Same as `<buffered-input-port>`'s `seek` slot.

Besides those slot values, you can pass an exact nonnegative integer as the `:buffer-size` keyword argument to the *make* method to set the size of the port's internal buffer. See the description of `<buffered-input-port>` above for the details.

Uniform vector ports

The following two procedures return a buffered input/output port backed up by a uniform vector. The source or destination vector can be any type of uniform vector, but they will be aliased to `u8vector` (see `uvector-alias` in Section 9.36.2 [Uvector conversion operations], page 484).

If used together with `pack/unpack` (see Section 12.2 [Packing binary data], page 641), it is useful to parse or construct binary data structure. It is also an example of using virtual ports; read `gauche/vport.scm` (or `ext/vport/vport.scm` in the source tree) if you're curious about the implementation.

`open-input-uvector` *uvector* [Function]

{`gauche.vport`} Returns an input port that reads the content of the given uniform vector *uvector* from its beginning. If reading operation reaches the end of *uvector*, EOF is returned. Seek operation is also implemented.

`open-output-uvector` *:optional uvector :key extendable* [Function]

{`gauche.vport`} Returns an output port that uses the given uvector as the storage for the data output to the port.

If *uvector* is completely filled, what happens after that depends on *extendable* - if it is false (default), the rest of data is discarded silently. If it is true, the storage is extended automatically to accommodate more data.

If you give true value to *extendable*, you have to retrieve the result by `get-output-uvector` below, since the uvector you passed in won't contain spilled data.

As a special case, you can omit *uvector* argument; then `u8vector` is used as the storage. In that case you can't specify *extendable* keyword argument, but it is assumed true, since it won't make sense otherwise. Use `get-output-uvector` to retrieve the stored result.

Seek operation is also implemented. Note that the meaning of `SEEK_END` whence differ between extendable and fixed-size uvector ports. For extendable ports, the end whence placed next to the biggest offset of the data ever written; if you open a port and just write one byte, the end whence is the second byte, no matter how big the existing buffer is. On the other hand, for fixed-size uvector ports, end whence is fixed to the next to the end of the given buffer, no matter how much data you've written to it. In the latter case, you can't seek on or past the end (you need to pass negative number along `SEEK_END` to `port-seek`).

`get-output-uvector` *port :key shared* [Function]

{`gauche.vport`} If *port* is a port created by `open-output-uvector`, returns a *uvector* that contains accumulated data. If *port* is not a port created by `open-output-uvector`, `#f` is returned.

The returned uvector is the same type as the one passed to `open-output-uvector`, containing up to actually written data; it may be smaller than the *uvector* passed to `open-output-uvector`; it can be larger if the port is extendable.

If the type of uvector is other than `s8vector` and `u8vector`, and the written data doesn't fill up the whole element won't be in the result. For example, if you use `s32vector` to create the port, then write 7 bytes to it, `get-output-uvector` returns a single element `s32vector`, for the last 3 bytes does not consist a whole 32bit integer.

By default, the returned vector is a fresh copy of the contents. Passing true value to *shared* may avoid copying and allow sharing storage for the one being used by *port*. If you do so, keep in mind that if you seek back and write to *port* subsequently, the content of returned vector may be changed.

List ports

The following procedures allow you to use list of characters or octets as a source of an input port. These are (a kind of) opposite of `port->list` family (see Section 6.22.7.4 [Input utility functions], page 231) or `port->char-lseq` family (see Section 6.19.2 [Lazy sequences], page 200).

`open-input-char-list` *char-list* [Function]
`open-input-byte-list` *byte-list* [Function]

{`gauche.vport`} Creates and returns an input port that uses the given list of characters and bytes as the source.

```
(read (open-input-char-list '(#\a #\b)))
 ⇒ ab
```

`get-remaining-input-list` *port* [Function]

{`gauche.vport`} If *port* is the one created by `open-input-char-list` or `open-input-byte-list`, returns a list of remaining data that hasn't been read yet. If the port already read everything, or the port is not the one created by `open-input-char-list` or `open-input-byte-list`, an empty list is returned.

A caveat: Gauche allows mixing binary input and textual input from the same port. If you read or even peek a byte from a port created from a character list, the port buffers a character and disassembles it to bytes; the disassembled character may not be included in the remaining input list.

Generator ports

The following procedures allow you to use character generators or byte generators as a source of an input port. These are (a kind of) opposite of `port->char-generator` family (see Section 9.11.1 [Generator constructors], page 372).

`open-input-char-generator` *cgen* [Function]
`open-input-byte-generator` *bgen* [Function]

{`gauche.vport`} Creates and returns an input port that uses the given generators as the source. The *cgen* argument must be a generator that yields characters. The *bgen* argument must be a generator that yields bytes (exact integers between 0 and 255, inclusive). An error will be raised if the given generator yields incorrect type of objects.

```
(read (open-input-char-generator (string->generator "foo")))
  ⇒ foo
```

Since the generators are objects relying on side effects, you shouldn't use *cgen* or *bgen* after you pass them to those procedures; if you use them afterwards, the result is undefined.

`get-remaining-input-generator` *port* [Function]

{`gauche.vport`} If *port* is the one created by `open-input-char-generator` or `open-input-byte-generator`, returns a generator that yields the characters or bytes that haven't been read yet. If the port already read everything, an empty generator is returned.

Once you take the remaining input generator, you should no longer read from the input generator ports; they share internal states and mixing them will likely to cause unexpected behaviors. If side-effects safe behavior is desired, use lazy sequence and input list ports.

Note on finalization

If an unclosed virtual port is garbage collected, its close procedure is called (in case of virtual buffered ports, its flush procedure may also be called before close procedure). It is done by a finalizer of the port. Since it is a part of garbage-collection process (although the Scheme procedure itself is called outside of the garbage collector main part), it requires special care.

- It is possible that the object the virtual port has a reference may already be finalized. For example, if a virtual port X holds the only reference to a *sink* port Y, to which the output goes. X's `flush` procedure sends its output to Y. However, if `flush` procedure can be called from a finalizer, it may be possible that Y's finalizer has already been called and Y is closed. So X's `flush` procedure has to check if Y has not been closed.
- You cannot know when and in which thread the finalizer runs. So if the procedure like `close` or `flush` of virtual ports need to lock or access the global resource, it needs to take extra care of avoiding dead lock or conflict of access.

  Even in single thread programs, the finalizer can run anywhere in Scheme programs, so effectively it should be considered as running in a different thread.

10 Library modules - R7RS standard libraries

Gauche predates R7RS, and for the convenience, Gauche makes quite a few procedures as built-in (see Chapter 6 [Core library], page 96). Although the set of Gauche's core features are mostly superset of R7RS, some functions and syntaxes have different names and/or interface from R7RS.

R7RS fully-compatible syntaxes and functions are available in the set of modules described in this chapter. Since R7RS programs and libraries needs to follow a specific format (`import` declaration or `define-library` form), generally there's no ambiguity in whether you're looking at R7RS code or Gauche-specific code. Also, it is totally transparent to load R7RS library into Gauche-specific code or vice versa. However, you need to be aware of which "world" you're in when you code.

If you're familiar with Gauche, take a look at the section Section 10.1 [R7RS integration], page 499, which describes how you can go back and forth between Gauche and R7RS.

10.1 R7RS integration

10.1.1 Traveling between two worlds back and forth

When you start Gauche, either in REPL or as a script, you're in `user` module, which *inherits* `gauche` module. Likewise, when you read a library, the initial module inherits `gauche` module (until you call `select-module`). That's why you can access all the built-in procedures of Gauche without saying `(use something)`. (See Section 4.13.5 [Module inheritance], page 75, for the details about inheriting modules).

On the other hand, R7RS requires to be explicit about which namespaces you'll be using, by `import` form, e.g. `(import (scheme base))`. Besides, R7RS library must be explicitly enclosed by `define-library` form. Before the first `import` form of a program, or outside of `define-library`, is beyond R7RS world—the standard defines nothings about it.

These facts let Gauche to set up appropriate "world", and you can use R7RS code and traditional Gauche code transparently.

NB: As explained in Section 10.1.2 [Three forms of import], page 501, R7RS `import` is rather different from Gauche `import`, so we note the former `r7rs#import` and the latter `gauche#import` in this section for clarity. When you write code don't use prefixes `r7rs#` and `gauche#`; just write `import`.

Loading R7RS libraries

The `define-library` form is defined as a macro in `gauche` module; it sets up R7RS environment before evaluating its contents. So, when you load an R7RS library (either from Gauche code via `use` form, or from R7RS code via `r7rs#import` form), Gauche starts loading the file in `gauche` module, but immediately see `define-library` form, and the rest is handled in R7RS environment.

Suppose you have an R7RS library `(mylib foo)` with the following code:

```
(define-library (mylib foo)
  (import (scheme base))
  (export snoc)
  (begin
    (define (snoc x y) (cons y x))))
```

It should be saved as `mylib/foo.scm` in one of the directories in `*load-path*`.

From R7RS code, this library can be loaded by `r7rs#import`:

```
(import (mylib foo))
```

```
(snoc 1 2) ⇒ (2 . 1)
```

To use this library from Gauche code, concatenate elements of library names by `.` to get a module name, and `use` it:

```
(use mylib.foo)

(snoc 1 2) ⇒ (2 . 1)
```

Loading Gauche libraries

To use Gauche library `foo.bar` from R7RS code, split the module name by `.` to make a list for the name of the library. For example, `gauche.lazy` module can be used from R7RS as follows:

```
(import (gauche lazy))
```

For SRFI modules, R7RS implementations have a convention to name it as `(srfi n)`, and Gauche follows it. The following code loads srfi-1 and srfi-13 from R7RS code:

```
(import (srfi 1) (srfi 13))
```

(It's not that Gauche treat `srfi` name specially; installation of Gauche includes adapter libraries such as `srfi/1.scm`.)

A tip: To use Gauche's built-in features (the bindings that are available by default in Gauche code) from R7RS code, import `(gauche base)` library (see Section 9.2 [Importing gauche builtins], page 322):

```
(import (gauche base))

filter ⇒ #<closure filter>
```

Running R7RS scripts

R7RS scripts always begin with `import` form. However, `r7rs#import` has a different syntax and semantics from `gauche#import`—so we employ a trick.

When `gosh` is started, it loads the given script file in `user` module. We have a separate `user#import` macro, which examines its arguments and if it is R7RS import syntax, switch to the `r7rs.user` module and run the `r7rs#import`. Otherwise, it runs `gauche#import`. See Section 10.1.2 [Three forms of import], page 501, for the details.

An example of R7RS script:

```
(import (scheme base) (scheme write))
(display "Hello, world!\n")
```

If you're already familiar with Gauche scripts, keep in mind that R7RS program doesn't treat `main` procedure specially; it just evaluates toplevel forms from top to bottom. So the following script doesn't output anything:

```
(import (scheme base) (scheme write))
(define (main args)
  (display "Hello, world!\n")
  0)
```

To access the command-line arguments in R7RS scripts, use `command-line` in `(scheme process-context)` library (see Section 10.2.12 [R7RS process context], page 510, also see Section 6.25.2 [Command-line arguments], page 248).

Using R7RS REPL

When `gosh` is invoked with `-r7` option and no script file is given, it enters an R7RS REPL mode. For the convenience, the following modules ("libraries", in R7RS term) are pre-loaded.

```
(scheme base) (scheme case-lambda) (scheme char)
```

```
(scheme complex) (scheme cxr) (scheme eval)
(scheme file) (scheme inexact) (scheme lazy)
(scheme load) (scheme process-context) (scheme read)
(scheme repl) (scheme time) (scheme write)
```

Besides, the history variables `*1`, `*2`, `*3`, `*1+`, `*2+`, `*3+`, `*e` and `*history` are available (See Section 3.2.1 [Working in REPL], page 21, for the details of history variables).

You can know you're in R7RS REPL by looking at the prompt, where `gosh` shows the current module (`r7rs.user`):

```
gosh[r7rs.user]>
```

To switch Gauche REPL from R7RS REPL, import `(gauche base)` and select `user` module using `select-module`:

```
gosh[r7rs.user]> (import (gauche base))
#<undef>
gosh[r7rs.user]> (select-module user)
#<undef>
gosh>
```

(You can `(select-module gauche)` but that's usually not what you want to do—changing `gauche` module can have unwanted side effects.)

When you're working on R7RS code in file and load it into R7RS REPL (for example, if you're using Emacs Scheme mode, C-c C-l does the job), make sure the file is in proper shape as R7RS; that is, the file must start with appropriate `import` declarations, or the file contains `define-library` form(s). If you load file without those forms, it is loaded into Gauche's `user` module no matter what your REPL's current module is, and the definitions won't be visible from `r7rs.user` module by default.

Switching from Gauche REPL

By default, `gosh` enters Gauche REPL when no script file is given. See Section 3.2.1 [Working in REPL], page 21, for detailed explanation of using REPL.

To switch Gauche REPL to R7RS REPL, simply use r7rs-style import; `user#import` knows you want R7RS and make a switch.

```
gosh> (import (scheme base))
#<undef>
gosh[r7rs.user]>
```

If you don't start `gosh` with `-r7` option, however, only the libraries you given to `user#import` are loaded at this moment.

If you want to switch the "vanilla" r7rs environment, that is, even not loading `(scheme base)`, then you can use `r7rs` module and directly select `r7rs.user`:

```
gosh> (use r7rs)
gosh> (select-module r7rs.user)
gosh[r7rs.user]>
```

If you do this, the only bindings visible initially are `import` and `define-library`; even `define` is undefined! You have to manually do `(import (scheme base))` etc. to start writing Scheme in this environment.

10.1.2 Three import forms

For historical reasons, Gauche has three `import` forms; the original Gauche's `import`, R7RS `import`, and the hybrid `import`.

Usually it is clear that the code is written in traditional Gauche or in R7RS, and usage of `import` is typically idiomatic, so there's not much confusion in practice. Only when you talk about `import` outside of code, you might need to specify which one you're talking.

The hybrid `import` is what we described `user#import` in the previous section (see Section 10.1.1 [Traveling between two worlds back and forth], page 499). It understands both of Gauche's `import` and R7RS `import`. So what you really need to know is the first two.

Gauche's module system design is inherited from STk, and we've been used `import` for purely name-space level operation; that is, it assumes the module you import from already exists in memory. Loading a file that defines the module (if necessary) is done by separate primitives, `require`. In most cases one file defines one module, and using that module means `require` it then `import` it (it's so common that Gauche has a macro for it—`use`). However, separating those two sometimes comes handy when you need some nontrivial hacks. See Section 4.13.4 [Using modules], page 73, for the details of Gauche's `import`.

R7RS leaves out the relation between modules (libraries) and files in order to give implementation freedom. If necessary, its `import` must load a file implicitly and transparently. So R7RS's `import` is semantically Gauche's `use`.

The hybrid `import` only appears at the beginning of the Scheme scripts. It finds out whether the script is in the traditional Gauche code or in the R7RS code. See Section 10.1.1 [Traveling between two worlds back and forth], page 499, for the details.

Now we'll explain R7RS `import`:

`import` *import-spec ...* [Special Form]

[R7RS] Imports libraries specified by *import-spec*s. What R7RS calls libraries are what Gauche calls modules; they're the same thing.

R7RS libraries are named by a list of symbols or integers, e.g. `(scheme base)` or `(srfi 1)`. It is translated to Gauche's module name by joining the symbols by periods; so, R7RS `(scheme base)` is Gauche's `scheme.base`. Conversely, Gauche's `data.queue` is available as `(data queue)` in R7RS. To use those two libraries, R7RS program needs this form at the beginning.

```
(import (scheme base)
        (data queue))
```

It works just like Gauche's `use` forms; that is, if the named module doesn't exist in the current process, it loads the file; then the module's exported bindings become visible from the current module.

```
(use scheme.base)
(use data.queue)
```

(You may wonder what if R7RS library uses symbols with periods in them. Frankly, we haven't decided yet. It'll likely be that we use some escaping mechanism; for the time being you'd want to stick with alphanumeric characters and hyphens as possible.)

Just like Gauche's `use`, you can select which symbols to be imported (or not imported), rename specific symbols, or add prefix to all imported symbols. The formal syntax of R7RS import syntax is as follows:

```
<import declaration> : (import <import-set> <import-set> ...)

<import-set> : <library-name>
  | (only <import-set> <identifier> <identifier> ...)
  | (except <import-set> <identifier> <identifier> ...)
  | (prefix <import-set> <identifier>)
  | (rename <import-set>
            (<identifier> <identifier>)
            (<identifier> <identifier>) ...)
```

```
<library-name> : (<identifier-or-base-10-integer>
                  <identifier-or-base-10-integer> ...)
```

10.2 R7RS small language

10.2.1 R7RS library form

R7RS libraries are defined by `define-library` form.

In R7RS view, `define-library` form itself does not belong to a Scheme code—it exists outside of the Scheme world. It defines the boundary of R7RS Scheme; inside `define-library` there is R7RS world, but outside, it's not a business of R7RS. For example, you can't generate `define-library` by a macro, within R7RS specification.

In Gauche, we implement R7RS world inside Gauche world; `define-library` itself is interpreted in the Gauche world. In fact, `define-library` *is* a Gauche macro. However, if you're writing portable R7RS code, you should forget how `define-library` is implemented, and do not put anything outside of `define-library` form.

`define-library` *library-name library-decl . . .* [Macro]

[R7RS] Defines a library *library-name*, which is a list of symbols or base-10 integer:

```
<library-name> : (<identifier-or-base-10-integer>
                  <identifier-or-base-10-integer> ...)
```

Library declarations *library-decl* can be export declarations, import declarations, `begin`-list of Scheme code, include forms, or `cond-expand` forms.

```
<library-decl> : (export <export-spec> ...)
               | <import declaration>
               | (begin <command-or-definition> ...)
               | (include <string> <string2> ...)
               | (include-ci <string> <string2> ...)
               | (include-library-declarations
                      <string> <string2> ...)
               | (cond-expand <cond-expand-clause>
                              <cond-expand-clause2> ...)
               | (cond-expand <cond-expand-clause>
                              <cond-expand-clause2> ...
                              (else <library-decl> ...))
```

The `export` declaration is the same Gauche's `export` form; see Section 4.13.4 [Using modules], page 73.

The `import` declaration is R7RS's `import` form, described in Section 10.1.2 [Three forms of import], page 501.

The `include` and `include-ci` declarations are the same as Gauche's; see Section 4.11 [Inclusions], page 67. Note that Gauche allows any code to be included—the content of the named file is simply wrapped with `begin` and substituted with these forms—but in R7RS definition, what you include must contain only Scheme code (not one of the library declarations or `define-library` form).

The `include-library-declarations` declaration works like `include`, but the content of the read file is interpreted as library declarations instead of Scheme code.

The `cond-expand` declaration is also the same as Gauche's; see Section 4.12 [Feature conditional], page 68. When used directly below `define-library`, it must expands to one of the library declarations.

10.2.2 `scheme.base` - R7RS base library

`scheme.base` [Module]

Exports bindings of R7RS `(scheme base)` library. From R7RS programs, those bindings are available by `(import (scheme base))`.

Bindings common to Gauche's built-ins

The following syntaxes and procedures are the same as Gauche's builtins:

Primitive expression types

```
quote if include include-ci lambda
```

Derived expression types

```
cond case and or when unless cond-expand let let* letrec letrec*
let-values let*-values begin do make-parameter parameterize
guard quasiquote unquote unquote-splicing case-lambda
```

Macros

```
let-syntax letrec-syntax syntax-rules syntax-error define-syntax
```

Variable definitions

```
define define-values
```

Record type definitions

```
define-record-type
```

Equivalence predicates

```
eqv? eq? equal?
```

Numbers

```
number? complex? real? rational? integer? exact? exact-integer?
= < > <= >= zero? positive? negative? odd? even? max min + * - / abs
floor/ floor-quotient floor-remainder
truncate/ truncate-quotient truncate-remainder
quotient modulo remainder gcd lcm numerator denominator
floor ceiling truncate round rationalize square exact-integer-sqrt
expt inexact exact number->string string->number
```

Booleans

```
not boolean? boolean=?
```

Pairs and lists

```
pair? cons car cdr set-car! set-cdr! caar cadr cdar cddr null? list?
make-list list length append reverse list-tail list-ref list-set!
memq memv member assq assv assoc list-copy
```

Symbols

```
symbol? symbol=? symbol->string string->symbol
```

Characters

```
char? char=? char<? char>? char<=? char>=? char->integer integer->char
```

Strings

```
string? make-string string string-length string-ref string-set!
string=? string<? string>? string<=? string>=? substring string-append
string->list list->string string-copy string-copy! string-fill!
```

Vectors

```
vector? make-vector vector vector-length vector-ref vector-set!
vector->list list->vector vector->string string->vector
vector-copy vector-copy! vector-append vector-fill!
```

Control features

```
procedure? apply map call-with-current-continuation call/cc
values call-with-values dynamic-wind
```

Exception

```
error
```

Environments and evaluation

```
scheme-report-environment null-environment
```

Input and output

```
input-port? output-port? port? current-input-port current-output-port
current-error-port close-port close-input-port close-output-port
open-input-string open-output-string get-output-string
read-char peek-char read-line eof-object? eof-object char-ready?
newline write-char
```

Bytevector utilities

R7RS's bytevectors are the same as Gauche's u8vectors.

The following procedures are the same as `gauche.uvector`'s (see Section 9.36.5 [Bytevector compatibility], page 490).

```
bytevector          bytevector?         make-bytevector
bytevector-length   bytevector-u8-ref   bytevector-u8-set!
bytevector-copy     bytevector-copy!    bytevector-append
```

And the following procedures are the same as `gauche.unicode`'s (see Section 9.35.1 [Unicode transfer encodings], page 472).

```
utf8->string        string->utf8
```

Control features

`string-map` *proc str . . .* [Function]
`string-for-each` *proc str . . .* [Function]

[R7RS base] {`scheme.base`} These take different arguments from `string-map` and `string-for-each` in SRFI-13 (see Section 11.5.10 [SRFI-13 String mapping], page 597), so provided only in `scheme.base` module to avoid confusion.

If you pass only one string argunemt, however, it works exactly the same way in both `srfi-13` and `scheme.base`.

`with-exception-handler` *handler thunk* [Function]

[R7RS base] {`scheme.base`} *handler* is evaluated in the dynamic environment of the call to *handler*, except that the current exception handler is that in place for the call to `with-exception-handler`.

If the exception is not continuable, it is reraised after evaluating *handler* and handled by the exception handler in place for the call to `with-exception-handler`.

This slightly differs from Gauche's built-in `with-exception-handler`, which calls *handler* without replacing the current exception handler. See Section 6.20.3 [Handling exceptions], page 208, for the details.

`raise` *obj* [Function]
`raise-continuable` *obj* [Function]

[R7RS base] {`scheme.base`} Gauche's `raise` may return if *obj* isn't a `<serious-condition>`. Distinguishing continuable and noncontinuable exception throw by the procedure has an issue when your exception handler wants to reraise the condition (you don't know if the original condition is raised by `raise` or `raise-continuable`!). Yet R7RS adopted that model, so we compel.

R7RS `raise` is a wrapper of Gauche's `raise`, which throws an error if Gauche's `raise` returns.

R7RS `raise-continuable` is currently just an alias of Gauche's `raise`—as long as you don't pass `<serious-condition>`, it may return. It is not exactly R7RS conformant—it won't return if you pass `<serious-condition>` or object of one of its subclasses (e.g. `<error>`), but it's weired to expect returning from raising `<error>`, isn't it?

`error-object?` *exc* [Function]

[R7RS base] {`scheme.base`} Defined as `(condition-has-type? exc <error>))`

`error-object-message` *exc* [Function]

[R7RS base] {`scheme.base`} If *exc* is a `<message-condition>`, returns its `message-prefix` slot; otherwise, returns an empty string.

`error-object-irritants` *exc* [Function]

[R7RS base] {`scheme.base`} If *exc* is a `<message-condition>`, returns its `message-args` slot; otherwise, returns an empty string.

`read-error?` *exc* [Function]

[R7RS base] {`scheme.base`} Defined as `(condition-has-type? e <read-error>))`.

`file-error?` *exc* [Function]

[R7RS base] {`scheme.base`} At this moment, Gauche doesn't have distinct `<file-error>` condition, but most file errors are thrown as one of `<system-error>`s. This procedure checks error code of `<system-error>` and returns `#t` if the error is likely to be related to the filesystem.

Input and output

`textual-port?` *port* [Function]
`binary-port?` *port* [Function]

[R7RS base] {`scheme.base`} Gauche's port can handle both, so these are equivalent to `port?`.

`input-port-open?` *iport* [Function]
`output-port-open?` *oport* [Function]

[R7RS base] {`scheme.base`} Checks whether *iport*/*oport* is an input/output port *and* it is not closed.

`open-input-bytevector` *u8vector* [Function]
`open-output-bytevector` [Function]
`get-output-bytevector` *port* [Function]

[R7RS base] {`scheme.base`} These are basically the same as `open-input-uvector`, `open-output-uvector` and *get-output-uvector* in `gauche.vport` (see Section 9.38 [Virtual ports], page 493), except that R7RS procedures only deal with `<u8vector>`.

`read-u8` *:optional iport* [Function]
`peek-u8` *:optional iport* [Function]
`u8-ready?` *:optional iport* [Function]

[R7RS base] {`scheme.base`} These are aliases to `read-byte`, `peek-byte` and `byte-ready?`, respectively.

`read-bytevector` *size :optional iport* [Function]

[R7RS base] {`scheme.base`} Equivalent to `(read-uvector <u8vector> size iport)`. See Section 9.36.4 [Uvector block I/O], page 488.

`read-bytevector!` *bv :optional iport start end* [Function]

[R7RS base] {`scheme.base`} An alias to `read-uvector!`. See Section 9.36.4 [Uvector block I/O], page 488.

`write-u8` [Function]

[R7RS base] {`scheme.base`} An alias to `write-byte`.

`write-bytevector` *bv :optional oport start end* [Function]

[R7RS base] {`scheme.base`} Equivalent to `write-uvector`. See Section 9.36.4 [Uvector block I/O], page 488.

`flush-output-port` *:optional oport* [Function]

[R7RS base] {`scheme.base`} An alias to `flush`.

`features` [Function]

[R7RS base] {`scheme.base`} Returns a list of symbols of supported feature identifiers, recognized by `cond-expand` (see Section 4.12 [Feature conditional], page 68).

10.2.3 `scheme.case-lambda` - R7RS case-lambda

`scheme.case-lambda` [Module]

Exports bindings of R7RS `(scheme case-lambda)` library. From R7RS programs, those bindings are available by `(import (scheme case-lambda))`.

The only binding exported from this module is `case-lambda`, and it is the same as Gauche's built-in `case-lambda`; see Section 4.3 [Making procedures], page 42, for the details.

10.2.4 `scheme.char` - R7RS char library

`scheme.char` [Module]

Exports bindings of R7RS `(scheme char)` library. From R7RS programs, those bindings are available by `(import (scheme char))`.

The following procedures are the same as Gauche's builtin procedures; see Section 6.10 [Characters], page 143.

```
char-alphabetic? char-ci<=? char-ci<?  char-ci=? char-ci>=? char-ci>?
char-downcase char-foldcase char-lower-case? char-numeric?
char-upcase char-upper-case?  char-whitespace?
```

The following procedures are the same as the ones provided in `gauche.unicode` module (see Section 9.35.3 [Full string case conversion], page 475). They use full case folding by Unicode standard (e.g. taking into account of German eszett).

```
string-ci<=? string-ci<?  string-ci=? string-ci>=?  string-ci>?
string-downcase string-foldcase string-upcase
```

`digit-value` *c* [Function]

[R7RS char] {`scheme.char`} If *c* is a character with `Nd` general category—that is, if it represents a decimal digit—this procedure returns the value the character represents. Otherwise it returns `#f`.

```
(digit-value #\3) ⇒ 3

(digit-value #\z) ⇒ #f
```

Note that Unicode defines about two dozen sets of digit characters.

```
(digit-value #\x11068) ⇒ 2
```

Gauche's built-in procedure `digit->integer` has more general interface (see Section 6.10 [Characters], page 143).

```
(digit-value c) ≡ (digit->integer c 10 #t)
```

10.2.5 `scheme.complex` - R7RS complex numbers

`scheme.complex` [Module]

Exports bindings of R7RS `(scheme complex)` library. From R7RS programs, those bindings are available by `(import (scheme complex))`.

This module provides the following bindings, all of which are Gauche built-in (see Section 6.3.5 [Numerical conversions], page 118).

```
angle imag-part magnitude make-polar make-rectangular real-part
```

10.2.6 `scheme.cxr` - R7RS cxr accessors

`scheme.cxr` [Module]

Exports bindings of R7RS `(scheme cxr)` library. From R7RS programs, those bindings are available by `(import (scheme cxr))`.

This module provides the following bindings, all of which are Gauche built-in (see Section 6.6.4 [List accessors and modifiers], page 127).

```
caaar caadr cadar caddr cdaar cdadr cddar cdddr caaaar caaadr caadar
caaddr cadaar cadadr caddar cadddr cdaaar cdaadr cdadar cdaddr cddaar
cddadr cdddar cddddr
```

10.2.7 `scheme.eval` - R7RS eval

`scheme.eval` [Module]

Exports bindings of R7RS `(scheme eval)` library. From R7RS programs, those bindings are available by `(import (scheme eval))`.

`eval` *expr environment* [Function]

[R7RS eval] {`scheme.eval`} This is the same as Gauche's built-in `eval` (see Section 6.21 [Eval and repl], page 216).

`environment` *import-list ...* [Function]

[R7RS eval] {`scheme.eval`} This is R7RS way to create an environment specifier suitable to pass to `eval`. In Gauche, an environment specifier is just a module object.

The argument is the same as what `r7rs#import` takes. This procedure creates an empty environment (as a fresh anonymous module; see `make-module` in Section 4.13.6 [Module introspection], page 76, for the details), then imports the bindings as specified by *import-lists*.

The following example creates an environment that includes `scheme.base` bindings plus `select-module` syntax from Gauche.

```
(environment
 '(scheme base)
 '(only (gauche base) select-module))
 ⇒ #<module #f>  ; an anonymous module
```

10.2.8 `scheme.file` - R7RS file library

`scheme.file` [Module]

Exports bindings of R7RS `(scheme file)` library. From R7RS programs, those bindings are available by `(import (scheme file))`.

The following bindings provided in this module are Gauche built-in (see Section 6.22.4 [File ports], page 221, and Section 6.25.4.4 [File stats], page 255).

```
call-with-input-file call-with-output-file
file-exists?
open-input-file open-output-file
with-input-from-file with-output-to-file
```

The following binding is the same as one in `file.util` (see Section 12.24.4 [File operations], page 699).

```
delete-file
```

`open-binary-input-file` *filename* [Function]
`open-binary-output-file` *filename* [Function]

[R7RS file] {`scheme.file`} In Gauche, ports are both textual and binary at the same time, so these R7RS procedures are just aliases of `open-input-file` and `open-output-file`, respectively. See Section 6.22.4 [File ports], page 221.

10.2.9 `scheme.inexact` - R7RS inexact numbers

`scheme.inexact` [Module]

Exports bindings of R7RS `(scheme inexact)` library. From R7RS programs, those bindings are available by `(import (scheme inexact))`.

This module provides the following bindings, all of which are Gauche built-in (see Section 6.3.4 [Arithmetics], page 112, and Section 6.3.2 [Numerical predicates], page 109).

```
acos asin atan cos exp finite? infinite? log nan? sin sqrt tan
```

10.2.10 `scheme.lazy` - R7RS lazy evaluation

`scheme.lazy` [Module]

Exports bindings of R7RS `(scheme lazy)` library. From R7RS programs, those bindings are available by `(import (scheme lazy))`.

The following bindings this module provides are Gauche built-ins (see Section 6.19.1 [Delay force and lazy], page 199).

```
delay force promise?
```

`delay-force` *promise* [Special Form]

[R7RS lazy] {`scheme.lazy`} This is the same as Gauche's built-in `lazy`. see Section 6.19.1 [Delay force and lazy], page 199, for the discussion about when this form should be used.

`make-promise` *obj* [Function]

[R7RS lazy] {`scheme.lazy`} If *obj* is a promise, it is returned as is. Otherwise, A promise, which yields *obj* when forced, is returned. Because this is a procedure, expression passed as *obj* is eagerly evaluated, so this doesn't have effect on lazy evaluation, but can be used to ensure you have a promise.

This procedure is important on implementations where `force` only takes a promise, and portable code should use this procedure to yield a value that can be passed to `force`.

If you write Gauche-specific code, however, `force` can take non-promise values, so you don't need this.

10.2.11 `scheme.load` - R7RS load

`scheme.load` [Module]

Exports bindings of R7RS `(scheme load)` library. From R7RS programs, those bindings are available by `(import (scheme load))`.

`load` *file :optional env* [Function]

[R7RS load] {`scheme.load`} R7RS `load` takes environment as an optional argument, while Gauche `load` takes it as a keyword argument (among other keyword arguments). See Section 6.23.1 [Loading Scheme file], page 239.

In Gauche, *env* is just a module. In portable code, you can create a module with desired bindings with R7RS `environment` procedure; see Section 10.2.7 [R7RS eval], page 508.

10.2.12 `scheme.process-context` - R7RS process context

`scheme.process-context` [Module]

Exports bindings of R7RS `(scheme process-context)` library. From R7RS programs, those bindings are available by `(import (scheme process-context))`.

The following bindings are the same as Gauche built-ins (see Section 6.25.2 [Command-line arguments], page 248, and Section 6.25.1 [Program termination], page 246):

```
command-line exit
```

The following bindings are the same as SRFI-98 (see Section 11.19 [Accessing environment variables], page 622):

```
get-environment-variable get-environment-variables
```

`emergency-exit` *:optional* (*obj 0*) [Function]

[R7RS process-context] {`scheme.process-context`} Terminate the program without running any clean-up procedures (*after* thunks of `dynamic-wind`). Internally, it calls the `_exit(2)` system call directly. The optional argument is used for the process exit code.

This is almost the same as Gauche's `sys-exit`, except that `sys-exit` requires the exit code object (see Section 6.25.1 [Program termination], page 246).

10.2.13 `scheme.read` - R7RS read

`scheme.read` [Module]

Exports bindings of R7RS `(scheme read)` library. From R7RS programs, those bindings are available by `(import (scheme read))`.

The only binding exported from this module is `read`, which is the same as Gauche's built-in. See Section 6.22.7.1 [Reading data], page 227.

10.2.14 `scheme.repl` - R7RS repl

`scheme.repl` [Module]

Exports bindings of R7RS `(scheme repl)` library. From R7RS programs, those bindings are available by `(import (scheme repl))`.

The only binding exported from this module is `interaction-environment`, which is the same as Gauche's built-in. See Section 6.21 [Eval and repl], page 216.

10.2.15 scheme.time - R7RS time

scheme.time [Module]

Exports bindings of R7RS (scheme time) library. From R7RS programs, those bindings are available by (import (scheme time)).

current-second [Function]

[R7RS time] {scheme.time} Returns a real number represents the number of seconds since the midnight of Jan. 1, 1970 TAI (which is 23:59:52, Dec 31, 1969 UTC, that is, -8 seconds before Unix Epoch.) Number of leap seconds were inserted since then, and as of 2014, UTC is 35 seconds behind TAI. That means the number returned is 27 seconds larger than the unix time, which is returned from sys-time or sys-gettimeofday.

The reason that R7RS adopts TAI is that it is monotonic and suitable to take difference of two timepoints. The unix time returned by sys-time and sys-gettimeofday are defined in terms of UTC date and time, so if the interval spans across leap seconds, it won't reflect the actual number of seconds in the interval. (The precise definition is given in section 4.15 of IEEE Std 1003.1, 2013 Edition, a.k.a Single Unix Specification 4.)

However, since we don't know yet when the next leap second happen, the current implementation just uses a fixed amount of offset from the unix time.

Just be aware the difference, or you'll be surprised if you pass the return value of current-second to the UTC time formatter such as sys-strftime, or compare it with the file timestamps which uses the unix time. You can convert between TAI and UTC using srfi-19 (see Section 11.7.4 [SRFI-19 Date], page 602).

current-jiffy [Function]

[R7RS time] {scheme.time} Returns an exact integer measuring a real (wallclock) time elapsed since some point in the past, which does not change while a process is running. The time unit is (/ jiffies-per-second)-th second.

The absolute value of current jiffies doesn't matter, but the difference can be used to measure the time interval.

jiffies-per-second [Function]

[R7RS time] {scheme.time} Returns a constant to tell how many time units used in current-jiffy consists of a second. Currently this is 10^9 on 64bit architectures (that is, nanosecond resolution) and 10^4 on 32bit architectures (100 microseconds resolution).

The resolution for 32bit architectures is unfortunately rather coarse, but if we make it finer the current jiffy value easily becomes bignums, taking time to allocate and operate, beating the purpose of benchmarking. With the current choice, we have 53,867 seconds before we spill into bignum. On 64bit architectures we have enough bits not to worry about bignums, with nanosecond resolution.

If you want to do more finer benchmarks on 32bit machines, you need to roll your own with sys-clock-gettime-monotonic or sys-gettimeofday.

10.2.16 scheme.write - R7RS write

scheme.write [Module]

Exports bindings of R7RS (scheme write) library. From R7RS programs, those bindings are available by (import (scheme write)).

This module provides the following bindings, all of which are Gauche built-in (see Section 6.22.8.3 [Object output], page 233).

```
display write write-shared write-simple
```

10.2.17 scheme.r5rs - R5RS compatibility

scheme.r5rs [Module]

This module is to provide R5RS environment in R7RS programs. The following bindings are exported. Note that `lambda` is `scheme#lambda`, without the support of extended formals (`:optional` etc.) See Section 4.3 [Making procedures], page 42, for the details of extended formals.

```
* + - / < <= = > >= abs acos and angle append apply asin assoc assq
assv atan begin boolean? caaaar caaadr caaar caadar caaddr caadr
caar cadaar cadadr cadar caddar cadddr caddr cadr
call-with-current-continuation call-with-input-file
call-with-output-file call-with-values car case cdaaar cdaadr cdaar
cdadar cdaddr cdadr cdar cddaar cddadr cddar cdddar cddddr cdddr cddr
cdr ceiling char->integer char-alphabetic? char-ci<=? char-ci<?
char-ci=? char-ci>=? char-ci>? char-downcase char-lower-case?
char-numeric? char-ready? char-upcase char-upper-case? char-whitespace?
char<=? char<? char=? char>=? char>? char? close-input-port
close-output-port complex? cond cons cos current-input-port
current-output-port define define-syntax delay denominator display
do dynamic-wind eof-object? eq? equal? eqv? eval even? exact->inexact
exact? exp expt floor for-each force gcd if imag-part inexact->exact
inexact? input-port? integer->char integer? interaction-environment
lambda lcm length let let* let-syntax letrec letrec-syntax list
list->string list->vector list-ref list-tail list? load log magnitude
make-polar make-rectangular make-string make-vector map max member
memq memv min modulo negative? newline not null-environment null?
number->string number? numerator odd? open-input-file open-output-file
or output-port? pair? peek-char positive? procedure? quasiquote quote
quotient rational? rationalize read read-char real-part real? remainder
reverse round scheme-report-environment set! set-car! set-cdr! sin
sqrt string string->list string->number string->symbol string-append
string-ci<=? string-ci<? string-ci=? string-ci>=? string-ci>?
string-copy string-fill! string-length string-ref string-set!
string<=? string<? string=? string>=? string>? string? substring
symbol->string symbol? tan truncate values vector vector->list
vector-fill! vector-length vector-ref vector-set! vector?
with-input-from-file with-output-to-file write write-char zero?
```

10.3 R7RS large

R7RS large is still under development, and we're gradually adding support of the libraries that has been passed.

Currently R7RS-large has two editions (Red and Tangerine). Among the libraries in those editions, the following are *not* supported yet:

scheme.ilist	scheme.rlist	scheme.text
scheme.bytevector	scheme.show	

The following are supported libraries:

10.3.1 scheme.list - R7RS lists

scheme.list [Module]

This module is a rich collection of list manipulation procedures, and same as `srfi-1`.

Note that Gauche supports quite a few `scheme.list` procedures as built-in. The following procedures can be used without loading `scheme.list` module. For the manual entries of these procedures, Section 6.6 [Pairs and lists], page 125.

```
null-list? cons* last member
take drop take-right drop-right take! drop-right!
delete delete! delete-duplicates delete-duplicates!
assoc alist-copy alist-delete alist-delete!
any every filter filter! fold fold-right find find-tail
split-at split-at! iota
```

List constructors

`xcons` *cd ca* [Function]

[R7RS list] {`scheme.list`} Equivalent to `(cons ca cd)`. Useful to pass to higher-order procedures.

`list-tabulate` *n init-proc* [Function]

[R7RS list] {`scheme.list`} Constructs an *n*-element list, in which each element is generated by `(`*`init-proc`*` i)`.

```
(list-tabulate 4 values) ⇒ (0 1 2 3)
```

`circular-list` *elt1 elt2 . . .* [Function]

[R7RS list] {`scheme.list`} Constructs a circular list of the elements.

```
(circular-list 'z 'q) ⇒ (z q z q z q ...)
```

List predicates

`not-pair?` *x* [Function]

[R7RS list] {`scheme.list`} Same as `(lambda (x) (not (pair? x)))`.

SRFI-1 says: Provided as a procedure as it can be useful as the termination condition for list-processing procedures that wish to handle all finite lists, both proper and dotted.

`list=` *elt= list . . .* [Function]

[R7RS list] {`scheme.list`} Determines list equality by comparing every n-th element of given lists by the procedure *elt=*.

It is an error to apply `list=` to anything except proper lists.

The equality procedure must be consistent with `eq?`, i.e.

```
(eq? x y) ⇒ (elt= x y).
```

List selectors

`first` *pair* [Function]
`second` *pair* [Function]
`third` *pair* [Function]
`fourth` *pair* [Function]
`fifth` *pair* [Function]
`sixth` *pair* [Function]
`seventh` *pair* [Function]
`eighth` *pair* [Function]
`ninth` *pair* [Function]
`tenth` *pair* [Function]

[R7RS list] {`scheme.list`} Returns n-th element of the (maybe improper) list.

`car+cdr` *pair* [Function]

[R7RS list] {`scheme.list`} Returns two values, `(car pair)` and `(cdr pair)`.

List miscellaneous routines

`zip` *clist1 clist2 . . .* [Function]

[R7RS list] {`scheme.list`} Equivalent to `(map list clist1 clist2 ...)`. If `zip` is passed *n* lists, it returns a list as long as the shortest of these lists, each element of which is an *n*-element list comprised of the corresponding elements from the parameter lists.

```
(zip '(one two three)
     '(1 2 3)
     '(odd even odd even odd even odd even))
     ⇒ ((one 1 odd) (two 2 even) (three 3 odd))

(zip '(1 2 3)) ⇒ ((1) (2) (3))
```

At least one of the argument lists must be finite:

```
(zip '(3 1 4 1) (circular-list #f #t))
     ⇒ ((3 #f) (1 #t) (4 #f) (1 #t))
```

`unzip1` *list* [Function]
`unzip2` *list* [Function]
`unzip3` *list* [Function]
`unzip4` *list* [Function]
`unzip5` *list* [Function]

[R7RS list] {`scheme.list`} `unzip1` takes a list of lists, where every list must contain at least one element, and returns a list containing the initial element of each such list. `unzip2` takes a list of lists, where every list must contain at least two elements, and returns two values: a list of the first elements, and a list of the second elements. `unzip3` does the same for the first three elements of the lists, and so on.

```
(unzip2 '((1 one) (2 two) (3 three))) ⇒
   (1 2 3) and
   (one two three)
```

List fold, unfold & map

`pair-fold` *kons knil clist1 clist2 . . .* [Function]
`pair-fold-right` *kons knil clist1 clist2 . . .* [Function]

[R7RS list] {`scheme.list`} Like `fold` and `fold-right`, but the procedure *kons* gets each `cdr` of the given *clists*, instead of `car`.

```
(pair-fold cons '() '(a b c d e))
  ⇒ ((e) (d e) (c d e) (b c d e) (a b c d e))

(pair-fold-right cons '() '(a b c d e))
  ⇒ ((a b c d e) (b c d e) (c d e) (d e) (e))
```

`unfold` *p f g seed :optional tail-gen* [Function]

[R7RS list] {`scheme.list`} Fundamental recursive list constructor. Defined by the following recursion.

```
(unfold p f g seed tail-gen) ≡
   (if (p seed)
       (tail-gen seed)
       (cons (f seed)
```

```
                    (unfold p f g (g seed))))
```

That is, *p* determines where to stop, *g* is used to generate successive seed value from the current seed value, and *f* is used to map each seed value to a list element.

```
(unfold (pa$ = 53) integer->char (pa$ + 1) 48)
  ⇒ (#\0 #\1 #\2 #\3 #\4)
```

`unfold-right` *p f g seed :optional tail* [Function]

[R7RS list] {`scheme.list`} Fundamental iterative list constructor. Defined by the following recursion.

```
(unfold-right p f g seed tail) ≡
  (let lp ((seed seed) (lis tail))
    (if (p seed)
        lis
        (lp (g seed) (cons (f seed) lis))))

(unfold-right (pa$ = 53) integer->char (pa$ + 1) 48)
 ⇒ (#\4 #\3 #\2 #\1 #\0)
```

`map!` *f clist1 clist2 . . .* [Function]

[R7RS list] {`scheme.list`} The procedure *f* is applied to each element of *clist1* and corresponding elements of *clist2*s, and the result is collected to a list. Cells in *clist1* is reused to construct the result list.

`map-in-order` *f clist1 clist2 . . .* [Function]

[R7RS list] {`scheme.list`} A variant of `map`, but it guarantees to apply *f* on each elements of arguments in a left-to-right order. Since Gauche's `map` implementation follows the same order, this function is just a synonym of `map`.

`pair-for-each` *f clist1 clist2 . . .* [Function]

[R7RS list] {`scheme.list`} Like `for-each`, but the procedure *f* is applied on *clist*s themselves first, then each `cdr`s of them, and so on.

```
(pair-for-each write '(a b c))
 ⇒ prints (a b c)(b c)(c)
```

List partitioning

`partition` *pred list* [Function]

`partition!` *pred list* [Function]

[R7RS list] {`scheme.list`} `filter` and `remove` simultaneously, i.e. returns two lists, the first is the result of filtering elements of *list* by *pred*, and the second is the result of removing elements of *list* by *pred*.

```
(partition odd? '(3 1 4 5 9 2 6))
  ⇒ (3 1 5 9) (4 2 6)
```

`partition!` is the linear-update variant. It may destructively modifies *list* to produce the result.

List searching

`take-while` *pred clist* [Function]

`take-while!` *pred list* [Function]

[R7RS list] {`scheme.list`} Returns the longest initial prefix of *clist* whose elements all satisfy *pred*.

`drop-while` *pred clist* [Function]
[R7RS list] {`scheme.list`} Drops the longest initial prefix of *clist* whose elements all satisfy *pred*, and returns the rest.

`span` *pred clist* [Function]
`span!` *pred list* [Function]
`break` *pred clist* [Function]
`break!` *pred list* [Function]
[R7RS list] {`scheme.list`} `span` is equivalent to `(values (take-while pred clist) (drop-while pred clist))`. `break` inverts the sense of *pred*.

`list-index` *pred clist1 clist2 . . .* [Function]
[R7RS list] {`scheme.list`} Returns the index of the leftmost element that satisfies *pred*. If no element satisfies *pred*, `#f` is returned.

Association lists

`alist-cons` *key datum alist* [Function]
[R7RS list] {`scheme.list`} Returns `(cons (cons key datum) alist)`. This is an alias of the Gauche builtin procedure `acons`.

Lists as sets

These procedures use a list as a set, that is, the elements in a list matter, but their order doesn't.

All procedures in this category takes a comparison procedure *elt=*, as the first argument, which is used to determine two elements in the given sets are the same.

Since lists require linear time to search, those procedures aren't suitable to deal with large sets. See Section 10.3.5 [R7RS sets], page 525, if you know your sets will contain more than a dozen items or so.

See also Section 12.62 [Combination library], page 794, which concerns combinations of elements in the set.

`lset<=` *elt= list1 . . .* [Function]
[R7RS list] {`scheme.list`} Returns `#t` iff all elements in *list1* are also included in *list2*, and so on. If no lists are given, or a single list is given, `#t` is returned.

`lset=` *elt= list1 list2 . . .* [Function]
[R7RS list] {`scheme.list`} Returns `#t` if all elements in *list1* are in *list2*, and all elements in *list2* are in *list1*, and so on.

```
(lset= eq? '(b e a) '(a e b) '(e e b a)) ⇒ #t
```

`lset-adjoin` *elt= list elt . . .* [Function]
[R7RS list] {`scheme.list`} Adds *elt* . . . to the set *list*, if each one is not already a member of *list*. (The order doesn't matter).

```
(lset-adjoin eq? '(a b c) 'a 'e) ⇒ '(e a b c)
```

`lset-union` *elt= list1 . . .* [Function]
[R7RS list] {`scheme.list`} Returns the union of the sets *list1* . . . .

`lset-intersection` *elt= list1 list2 . . .* [Function]
[R7RS list] {`scheme.list`} Returns a set of elements that are in every *list*s.

`lset-difference` *elt= list1 list2 . . .* [Function]
[R7RS list] {`scheme.list`} Returns a set of elements that are in *list1* but not in *list2*. In n-ary case, binary difference operation is simply folded.

`lset-xor` *elt= list1 . . .* [Function]

[R7RS list] {`scheme.list`} Returns the exclusive-or of given sets; that is, the returned set consists of the elements that are in either *list1* or *list2*, but not in both. In n-ary case, binary xor operation is simply folded.

`lset-diff+intersection` *elt= list1 list2 . . .* [Function]

[R7RS list] {`scheme.list`} Returns two sets, a difference and an intersection of given sets.

`lset-union!` *elt= list . . .* [Function]
`lset-intersection!` *elt= list1 list2 . . .* [Function]
`lset-difference!` *elt= list1 list2 . . .* [Function]
`lset-xor!` *elt= list1 . . .* [Function]
`lset-diff+intersection!` *elt= list1 list2 . . .* [Function]

[R7RS list] {`scheme.list`} Linear update variant of the corresponding procedures. The cells in the first list argument may be reused to construct the result.

10.3.2 `scheme.vector` - R7RS vectors

`scheme.vector` [Module]

This module adds rich set of vector operations to the built-in / R7RS vector procedures.

The following procedures are built-in. See Section 6.14 [Vectors], page 174, for the description. We only explain the procedures that are not built-in.

```
make-vector          vector               vector?
vector-ref           vector-set!          vector-length
vector-fill!         vector-copy          vector-copy!
vector-append        vector->list         list->vector
reverse-list->vector vector->string       string->vector
vector-map           vector-map!          vector-for-each
```

This module is `srfi-133`, which supersedes `srfi-43` (see Section 11.13 [Vector library (Legacy)], page 615). Note that the interface of following procedures in `srfi-43` are changed for the consistency:

```
vector-map           vector-map!          vector-for-each
vector-fold          vector-fold-right    vector-count
```

Some of the functionalities of `srfi-43` version is supported by built-in procedures (e.g. Built-in `vector-map-with-index` is the same as `srfi-43`'s `vector-map`). So there's little point for new code to use `srfi-43`.

Vector constructors

`vector-unfold` *f length seed . . .* [Function]

[R7RS vector] {`scheme.vector`} Creates a vector of length *length*, filling elements left to right by calling *f* repeatedly.

The procedure *f* must take as many arguments as one plus number of seed values, and must return the same number of values. The first argument is the index. The first return value is used for the element of the result vector, and the rest of return values are passed to the next call of *f*.

```
(vector-unfold (^[i] (* i i)) 5)
 ⇒ #(0 1 4 9 16)

(vector-unfold (^[i x] (values (cons i x) (* x 2))) 8 1)
 ⇒ #((0 . 1) (1 . 2) (2 . 4) (3 . 8)
    (4 . 16) (5 . 32) (6 . 64) (7 . 128))
```

`vector-unfold-right` *f length seed . . .* [Function]

[R7RS vector] {`scheme.vector`} Creates a vector of length *length*, filling elements right to left by calling *f* repeatedly.

The procedure *f* must take as many arguments as one plus number of seed values, and must return the same number of values. The first argument is the index. The first return value is used for the element of the result vector, and the rest of return values are passed to the next call of *f*.

```
(vector-unfold-right (^[i] (* i i)) 5)
 ⇒ #(0 1 4 9 16)

(vector-unfold-right (^[i x] (values (cons i x) (* x 2))) 8 1)
 ⇒ #((0 . 128) (1 . 64) (2 . 32) (3 . 16)
    (4 . 8) (5 . 4) (6 . 2) (7 . 1))
```

`vector-reverse-copy` *vec :optional start end* [Function]

[R7RS vector] {`scheme.vector`} Copies the vector *vec* with reversing its elements. Optional *start* and *end* arguments can limit the range of the input.

```
(vector-reverse-copy '#(a b c d e) 1 4)
 ⇒ #(d c b)
```

`vector-concatenate` *list-of-vectors* [Function]

[R7RS vector] {`scheme.vector`} Same as `(apply vector-append list-of-vectors)`.

`vector-append-subvectors` *spec . . .* [Function]

[R7RS vector] {`scheme.vector`} The number of arguments must be multiple of 3. The argument list must be in the following format, where each *vecN* is a vector, and `startN` and `endN` are nonnegative integers:

```
vec1 start1 end1 vec2 start2 end2 ...
```

This procedure creates a new vector by concatenating subvectors specified by each triplet. That is, it works as if it's the following code, except it avoids copying each subvector:

```
(vector-append (vector-copy vec1 start1 end1)
               (vector-copy vec2 start2 end2)
               ...)
```

Here's an example:

```
(vector-append-subvectors '#(a b c d e) 0 3
                          '#(f g h i j) 2 5)
  ⇒ #(a b c h i j)
```

Vector predicates

`vector-empty?` *vec* [Function]

[R7RS vector] {`scheme.vector`} Returns `#t` if *vec*'s length is zero, and `#f` if *vec*'s length is more than zero. Signals an error if *vec* is not a vector.

`vector=` *elt= vec . . .* [Function]

[R7RS vector] {`scheme.vector`} Compares *vec*s element-wise, using given predicate *elt=*. Returns `#t` iff lengths of all the vectors are the same, and every corresponding elements are equal by *elt=*. *Elt=* is always called with two arguments and must return `#t` iff two are the same.

Vector iteration

`vector-fold` *kons knil vec1 vec2 . . .* [Function]

[R7RS vector] {`scheme.vector`} *Kons* is a procedure that takes *n*+1 arguments, where *n* is the number of given vectors. For each element of the given vectors, *kons* is called as `(kons seed e_1i e_2i ...)`, where and *e_ni* is the *i*-th element of the vector *n*. If the lengths of the vectors differ, iteration stops when the shortest vector is exhausted.

The initial value of *seed* is *knil*, and the return value from *kons* is used as the next seed value. The last return value of *kons* is returned from `vector-fold`.

The iteration is strictly left to right.

Note that the seed value precedes elements, which is opposite to `fold` (see Section 9.5.1 [Mapping over collection], page 345). It's an unfortunate historical glitch; `vector-fold-left` would be more consistent name.

```
(vector-fold (^[a b] (cons b a)) '() '#(a b c d))
  ⇒ (d c b a)
```

`vector-fold-right` *kons knil vec1 vec2 . . .* [Function]

[R7RS vector] {`scheme.vector`} Like `vector-fold`, but elements in the *vec1 vec2 . . .* are visited from right to left.

Unlike `fold-right` (see Section 9.29.3 [Mapping over sequences], page 442), the procedure *kons* takes the accumulated value in the first argument.

```
(vector-fold-right (^[a b] (cons b a)) '() '#(a b c d))
  ⇒ (a b c d)
```

`vector-count` *pred vec1 vec2 . . .* [Function]

[R7RS vector] {`scheme.vector`} Applies *pred* on each elements in argument vectors (if N vectors are given, *pred* takes N arguments, the first being *i*-th element of *vec1*, the second being *i*-th element of *vec2*, etc.) Then returns the number of times *pred* returned true value. The order *pred* applied to each element is unspecified.

```
(vector-count odd? '#(0 1 2 3 4)
  ⇒ 2

(vector-count < '#(7 3 9 1 5) '#(6 8 2 3 8 8))
  ⇒ 3
```

`vector-cumulate` *f seed vec* [Function]

[R7RS vector] {`scheme.vector`} Returns a fresh vector with the same size of *vec*, with the elements calculated as follows:

The first element of result vector is a result of procedure *f* called with *seed* and the first element of *vec*.

The *i*-th element of result vector is a result of procedure *f* called with *i-1*-th element of result vector and *i*-th element of *vec*.

```
(vector-cumulate string-append "z" '#("a" "b" "c"))
  ⇒ #("za" "zab" "zabc")
```

Vector searching

`vector-index` *pred vec1 vec2 . . .* [Function]
`vector-index-right` *pred vec1 vec2 . . .* [Function]

[R7RS vector] {`scheme.vector`} Returns the index of the first or the last elements in *vec1 vec2 . . .* that satisfy *pred*, respectively. Returns `#f` if no elements satisfy *pred*. In `vector-index`, comparison ends at the end of the shortest vector. For `vector-index-right`, all the vectors must have the same length.

`vector-skip` *pred vec1 vec2 . . .* [Function]
`vector-skip-right` *pred vec1 vec2 . . .* [Function]

[R7RS vector] {`scheme.vector`} Like `vector-index` and `vector-index-right`, except that the result of *pred* is negated. That is, returns the index of the first or the last elements that don't satisfy *pred*.

`vector-binary-search` *vec value cmp :optional start end* [Function]

[R7RS+] {`scheme.vector`} Look for *value* in a sorted vector *vec*, and returns its index if it is found, or `#f` if it is not found.

Comparison of *value* and an element in *vec* is done by a procedure *cmp*, which takes two arguments, and should return a negative integer if the first argument is less than the second, 0 if they are the same, and a positive integer if the first is greater than the second.

Elements in *vec* must be ordered from smaller to greater w.r.t. *cmp*. Using that fact, this procedure performs binary search instead of linear search.

The optional arguments *start* and *end* are an extension to SRFI-133, and can be used to limit the range of the search in *start*-th element (inclusive) to *end*-th element (exclusive).

`vector-any` *pred vec1 vec2 . . .* [Function]

[R7RS vector] {`scheme.vector`} Applies *pred* on each corresponding elements of *vec1 vec2 . . .* left to right, and as soon as *pred* returns non-`#f` value, the procedure stops iteration and returns the value.

If no elements that satisfy *pred* are found, it returns `#f`.

Vectors can have different lengths. Iteration stops at the end of the shortest.

`vector-every` *pred vec1 vec2 . . .* [Function]

[R7RS vector] {`scheme.vector`} Applies *pred* on each corresponding elements of *vec1 vec2 . . .* left to right. If all the elements (when the lengths of vectors differ, the first N elements where N is the length of the shortest) satisfy *pred*, returns the last result of *pred*. If any of the elements don't satisfy *pred*, it returns `#f` immediately without looking further.

```
(vector-every < '#(1 2 3 4 5) '#(2 3 4 4 5)
  ⇒ #f

(vector-every (^[x y] (and (real? x) (real? y) (- x y)))
              '#(1 2 3)
              '#(2 4 6))
  ⇒ -3
```

`vector-partition` *pred vec* [Function]

[R7RS vector] {`scheme.vector`} Allocates a fresh vector of the same size as *vec*, then fill it with elements in *vec* that satisfy *pred*, followed by elements that don't satisfy *pred*.

Returns two values, the newly created vector and an exact integer of the index of the first element that doesn't satisfy *pred* in the returned vector.

```
(vector-partition odd? '#(1 2 3 4 5 6 7 8))
  ⇒ #(1 3 5 7 2 4 6 8) and 4
```

Vector mutators

`vector-swap!` *vec i j* [Function]

[R7RS vector] {`scheme.vector`} Swaps vector *vec*'s *i*-th and *j*-th elements. Returns unspecified value.

```
(rlet1 v (vector 'a 'b 'c 'd 'e)
  (vector-swap! v 0 2))
  ⇒ #(c b a d e)
```

`vector-reverse!` *vec :optional start end* [Function]

[R7RS vector] {`scheme.vector`} Reverse the elements of *vec*. Returns an undefined value. Optional *start* and *end* arguments can limit the range of operation.

```
(rlet1 v (vector 'a 'b 'c 'd 'e)
  (vector-reverse! v 0 4))
  ⇒ #(d c b a e)
```

`vector-reverse-copy!` *target tstart source :optional sstart send* [Function]

[R7RS vector] {`scheme.vector`} Like `vector-copy!`, but reverses the order of elements from *start*.

```
(rlet1 v (vector 'a 'b 'c 'd 'e)
  (vector-reverse-copy! v 2 '#(1 2)))
  ⇒ #(a b 2 1 e)
```

It is ok to pass the same vector to *target* and *source*; it always works even if the regions of source and destination are overlapping.

```
(rlet1 v (vector 'a 'b 'c 'd 'e)
  (vector-reverse-copy! v 1 v 1))
  ⇒ #(a e d c b)
```

`vector-unfold!` *f rvec start end seeds ...* [Function]
`vector-unfold-right!` *f rvec start end seeds ...* [Function]

[R7RS vector] {`scheme.vector`} Fill *rvec* starting from index *start* (inclusive) and ending at index *end* (exclusive), with the elements calculated by *f*.

The procedure *f* takes the number of seed values *seeds* ... plus one arguments. The first argument is the current index, followed by seed values. The same number of values as the arguments must be returned from *f*; the first return value is used to fill the current element of *rvec*, and the rest of the values are used as the next seed values.

The result vector is filled from left to right by `vector-unfold!`, and right to left by `vector-unfold-right!`. The return value is unspecified.

```
(let1 rvec (vector 'a 'b 'c 'd 'e 'f)
  (vector-unfold! (^[i] (+ i 1)) rvec 1 4)
  rvec)
 ⇒ #(a 2 3 4 e f)

(let1 rvec (vector 'a 'b 'c 'd 'e 'f)
  (vector-unfold-right! (^[i] (+ i 1)) rvec 1 4)
  rvec)
 ⇒ #(a 2 3 4 e f)

(let1 rvec (vector 'a 'b 'c 'd 'e 'f)
  (vector-unfold! (^[i x] (values x (* x 2))) rvec 1 5 10)
  rvec)
 ⇒ #(a 10 20 40 80 f)

(let1 rvec (vector 'a 'b 'c 'd 'e 'f)
  (vector-unfold! (^[i x] (values x (* x 2))) rvec 1 5 10)
  rvec)
 ⇒ #(a 80 40 20 10 f)
```

Vector conversion

`reverse-vector->list` *vec :optional start end* [Function]

[R7RS vector] {`scheme.vector`} Same as `(reverse (vector->list vec start end))`, but more efficient.

10.3.3 `scheme.vector.@` - R7RS uniform vectors

`scheme.vector.@` [Module]

`@` is actually one of `u8`, `s8`, `u16`, `s16`, `u32`, `s32`, `u64`, `s64`, `f32`, `f64`, `c64` or `c128`. (Gauche's `gauche.uvector` module also provides `f16` and `c32` vectors.)

These modules provides vectors that can hold specific range of numeric values. In Gauche we use *packed* representation, meaning numbers are tightly stored in consecutive memory region.

Additionally, `scheme.vector.base` module exports basic procedures, `make-@vector`, `@vector`, `@vector?`, `@vector-length`, `@vector-ref`, `@vector-set!`, `@vector->list`, `list->@vector`, `@?`, for `@` being over all element types.

The `gauche.uvector` module is a superset of these modules, and all procedures are described there. See Section 9.36 [Uniform vectors], page 476, for the details.

10.3.4 `scheme.sort` - R7RS sort

`scheme.sort` [Module]

Provides utilities to sort, and to work on sorted lists/vectors. This module is the same as `srfi-132`.

Gauche has built-in sort and merge procedures (see Section 6.24 [Sorting and merging], page 244). This module has a bit different API. Notably, the ordering predicate comes first than the sequence to be sorted, and the procedures dealing with vectors uniformly support start/end arguments

This module also provide useful procedures working on sorted or partially sorted sequences.

`list-sort` *elt< lis* [Function]
`list-sort!` *elt< lis* [Function]
`list-stable-sort` *elt< lis* [Function]
`list-stable-sort!` *elt< lis* [Function]

[R7RS sort] {`scheme.sort`} Sort elements in a list *lis* according to the ordering predicate *elt<*, which takes two elements from *lis* and returns true iff the first argument is strictly less than the second argument.

Returns a sorted list. The procedures with bang are linear update version. They are allowed, but not required, to reuse *lis*. The "stable" variation guarantees stable sort.

These are basically the same as Gauche's built-in `sort`, `sort!`, `stable-sort` and `stable-sort!`, except the Gauche's version works on any sequences and takes arguments differently. (See Section 6.24 [Sorting and merging], page 244.)

`list-sorted?` *elt< lis* [Function]

[R7RS sort] {`scheme.sort`} Returns true if the list *list* is sorted according to the ordering predicate *elt<*.

See also `sorted?` in Section 6.24 [Sorting and merging], page 244.

`list-merge` *elt< lis1 lis2* [Function]
`list-merge!` *elt< lis1 lis2* [Function]

[R7RS sort] {`scheme.sort`} Given two sorted lists *lis1* and *lis2*, returns a new sorted list according to the ordering predicate *elt<*.

Note that `list-merge!` works in-place, that is, all the pairs in *lis1* and *lis2* are reused.

See also `merge` and *merge!* in Section 6.24 [Sorting and merging], page 244.

`vector-sort` *elt< vec :optional start end* [Function]
`vector-stable-sort` *elt< vec :optional start end* [Function]

[R7RS sort] {`scheme.sort`} Sort elements in a vector *vec* according to the ordering predicate *elt<*, which takes two elements from *vec* and returns true iff the first argument is strictly less than the second argument. Returns a fresh sorted vector. The "stable" variation guarantees stable sort.

When the optional *start* and/or *end* arguments are given, only the portion from *start* (inclusive) and *end* (exclusive) of *vec* are looked at. The result vector's length is `end - start`). When *end* is omitted, the length of *vec* is assumed.

See also `sort` and `stable-sort` in Section 6.24 [Sorting and merging], page 244.

`vector-sort!` *elt< vec :optional start end* [Function]
`vector-stable-sort!` *elt< vec :optional start end* [Function]

[R7RS sort] {`scheme.sort`} Sort elements "in-place" in a vector *vec* according to the ordering predicate *elt<*, which takes two elements from *vec* and returns true iff the first argument is strictly less than the second argument. Upon successful return, *vec*'s elements are sorted. Returns unspecified value; the caller must rely on the side effect.

When the optional *start* and/or *end* arguments are given, only the portion from *start* (inclusive) and *end* (exclusive) of *vec* are sorted; other elements will remain intact. When *end* is omitted, the length of *vec* is assumed.

See also `sort!` and `stable-sort!` in Section 6.24 [Sorting and merging], page 244.

`vector-sorted?` *elt< vec :optional start end* [Function]

[R7RS sort] {`scheme.sort`} Returns true iff *vec* between *start* (inclusive) and *end* (exclusive) is sorted according to the ordering predicate *elt<*. If *start* and/or *end* is/are omitted, 0 and the length of *vec* are assumed, respectively.

See also `sorted?` in Section 6.24 [Sorting and merging], page 244.

`vector-merge` *elt< vec1 vec2 :optional start1 end1 start2 end2* [Function]
`vector-merge!` *elt< rvec vec1 vec2 :optional rstart start1 end1 start2 end2* [Function]

[R7RS sort] {`scheme.sort`} Merge two sorted vectors *vec1* and *vec2* into one vector, according to the ordering predicate *elt<*.

The optional argument *start1* and *end1* restricts *vec1*'s portion to be looked at, and *start2* and *end2* restricts *vec2*'s portion to be looked at.

The functional version *vector-merge* allocates a fresh vector to hold the result, and returns it.

The side-effecting version *vector-merge!* uses *rvec.* to hold the result. The procedure doesn't return a meaningful value. The optional *rstart* argument specifies the index of *rvec* from which the result is filled; the default is 0.

`list-delete-neighbor-dups` *elt= lis* [Function]
`list-delete-neighbor-dups!` *elt= lis* [Function]
`vector-delete-neighbor-dups` *elt= vec :optional start end* [Function]
`vector-delete-neighbor-dups!` *elt= vec :optional start end* [Function]

[R7RS sort] {`scheme.sort`} From the given list *lis* or vector *vec*, these procedures delete adjacent duplicate elements. Equivalence is checked by *elt=* procedure.

```
(list-delete-neighbor-dups eq? '(m i s s i s s i p p i))
```

```
    ⇒ (m i s i s i p i)
```

The non-destructive versions `list-delete-neighbor-dups` and `vector-delete-neighbor-dups` returns a freshly allocated list and vector, respectively.

The destructive `list-delete-neighbor-dups!` works in-place, reusing pairs of *lis*. No allocation will be done.

The destructive `vector-delete-neighbor-dups!` has a bit different interface. It updates *vec* in-place, but since we can't change the length of the vector, it gathers the result from the beginning of the *vec*, then returns the next index *newend* of *vec*—that is, after calling this procedure, `[start, newend)` holds the result. The elements between `[newend, end)` will remain intact.

The optional *start* and *end* arguments limits the region of *vec* to be looked at.

```
(vector-delete-neighbor-dups eq? '#(a a a b b c c d d e e f f) 3 10)
  ⇒ #(b c d e)

(let1 v '#(a a a b b c c d d e e f f)
  (cons (vector-delete-neighbor-dups! eq? v 3 10) v))
  ⇒ (7 . #(a a a b c d e d d e e f f))
```

Note: The `gauche.sequence` module provides neighbor duplicate deletion on generic sequences. Those procedures are implemented by the generic versions as shown below. See Section 9.29.4 [Other operations over sequences], page 443, for the details.

`list-delete-neighbor-dups`
: `delete-neighbor-dups`

`list-delete-neighbor-dups!`
: `delete-neighbor-dups-squeeze!`

`vector-delete-neighbor-dups`
: `delete-neighbor-dups`

`vector-delete-neighbor-dups!`
: `delete-neighbor-dups!`

`vector-select!` *elt< vec k :optional start end* [Function]

[R7RS sort] {`scheme.sort`} Select *k*-th smallest element in *vec* according to the ordering predicate *elt<*. *K* is zero based, i.e. 0 means the smallest. The optional *start* and *end* arguments limits the range of *vec* to be looked at, and defaulted to 0 and the length of *vec*, respectively. *K* must satisfy `start <= k < end`.

This procedure runs in O(n) time, and requires no extra stroage. This procedure may partially modify *vec*.

`vector-separate!` *elt< vec k :optional start end* [Function]

[R7RS sort] {`scheme.sort`} Find *k*-th smallerst element in *vec* (pivot) between between *start* and *end*, according to the ordering predicate *elt<*, then rearrange elements between *start* and *end* so that elements smaller than the pivot comes between `start` and `start + k`, and the rest of the elements come afterwards. When omitted, *start* is 0 and *end* is the length of the *vec*.

This can be used as a building block for in-place divide-and-conquer algorithms. Runs in O(n) time.

`vector-find-median` *elt< vec knil :optional mean* [Function]
`vector-find-median!` *elt< vec knil :optional mean* [Function]

[R7RS sort] {`scheme.sort`} Find median value of elements in *vec*, when ordered by the ordering predicate *elt<*. Non-destructive version `vector-find-median` runs in O(n) time.

The destructive version `vector-find-median!` is specified to leave *vec* sorted, so it runs in O(n log n).

1. If *vec* is empty, *knil* is returned. This is the only case *knil* is used.
2. If *vec* has odd number of elements, the element falls in the exactly the midpoint when ordered, is returned.
3. If *vec* has even number of elements, the two elements closest to the midpoint is chosen and passed to the procedure *mean*, and its result is returned. The default of *mean* is an arithmetic mean of numbers.

```
(vector-find-median < #() 0)
  ⇒ 0

(vector-find-median < #(78 61 19 38 51) 0)
  ⇒ 51

(vector-find-median < #(78 61 19 38 51 52) 0)
  ⇒ 103/2
```

10.3.5 `scheme.set` - R7RS sets

`scheme.set` [Module]

Sets and bags are unordered collection of Scheme values. A set doesn't count duplicates; if you add an item which is already in a set, you still have one item of the kind. A bag counts duplicates; if you add an item which is already in a bag, you have two items of the kind.

To check whether the items are "the same", sets and bags takes a comparator at construction time. The comparator doesn't need to have an ordering predicate (we don't need to order the elements) but has to have a hash function. See Section 6.2.4 [Basic comparators], page 103, for the details of comparators.

This module is originally specified as `srfi-113`, and then incorporated to R7RS large.

As a Gauche's extension, sets and bags implement collection protocol (see Section 9.5 [Collection framework], page 344, for the details), and generic collection operations can be applied.

```
(coerce-to <list> (set eq-comparator 'a 'b 'a 'b))
  ⇒ (a b)       ; order may differ

(coerce-to <list> (bag eq-comparator 'a 'b 'a 'b))
  ⇒ (a a b b)   ; order may differ
```

Constructors

`set` *comparator elt . . .* [Function]
`bag` *comparator elt . . .* [Function]

[R7RS set] {`scheme.set`} Creates a new set and bag from given elements *elt* . . . . Given *comparator* will be used to compare equality of elements.

```
(set->list (set eq-comparator 'a 'b 'a 'b))
  ⇒ (a b)

(bag->list (bag eq-comparator 'a 'b 'a 'b))
  ⇒ (a a b b)
```

`set-unfold` *stop? mapper successor seed comparator* [Function]
`bag-unfold` *stop? mapper successor seed comparator* [Function]

[R7RS set] {`scheme.set`} Procedurally creates a set or a bag. The first three arguments, *stop?*, *mapper* and *successor*, are all procedures that takes one argument, the current seed value. It may be easier to know their types:

```
seed      :: Seed
stop?     :: Seed -> Boolean
mapper    :: Seed -> ElementType
successor :: Seed -> Seed
```

The `stop?` procedure takes the current seed value and returns a boolean value - if it is true, iteration stops.

The `mapper` procedure takes the current seed value and returns an item, which is to be included in the resulting set or bag.

The `successor` procedure takes the current seed value and returns the next seed value.

And the `seed` argument gives the initial seed value.

```
(set->list (set-unfold (^s (= s 75))
                       integer->char
                       (^s (+ s 1))
                       65
                       eqv-comparator))
 ⇒ (#\D #\H #\A #\E #\I #\J #\B #\F #\C #\G)
```

Predicates

`set-contains?` *set obj* [Function]
`bag-contains?` *bag obj* [Function]

[R7RS set] {`scheme.set`} Check if *obj* is in the set or the bag.

`set-empty?` *set* [Function]
`bag-empty?` *bag* [Function]

[R7RS set] {`scheme.set`} Returns `#t` iff the given set or bag is empty.

`set-disjoint?` *set1 set2* [Function]
`bag-disjoint?` *bag1 bag2* [Function]

[R7RS set] {`scheme.set`} Returns `#t` iff the given arguments (sets or bags) don't have common items. Both arguments must have the same comparator—otherwise an error is signaled.

Accessors

`set-member` *set obj default* [Function]
`bag-member` *bag obj default* [Function]

[R7RS set] {`scheme.set`} Returns an element in the given set or bag which is equal to *obj* in terms of the set's or the bag's comparator. If no such element is found, *default* will be returned.

Note that the returned object doesn't need to be "the same" as *obj* in a usual sense. See the following example:

```
(let s (set string-ci-comparator "abc" def")
  (set-member s "ABC" #f))
  ⇒ "abc"
```

`set-element-comparator` *set* [Function]
`bag-element-comparator` *bag* [Function]

[R7RS set] {`scheme.set`} Returns the comparator used to compare the elements for the set or the bag.

Updaters

`set-adjoin` *set elt ...* [Function]
`bag-adjoin` *bag elt ...* [Function]

[R7RS set] {`scheme.set`} Returns a newly created set or bag that contains all the elements in the original set/bag, plus given elements. The new set/bag's comparator is the same as the original set/bag's one.

`set-replace` *set elt* [Function]
`bag-replace` *bag elt* [Function]

[R7RS set] {`scheme.set`} Returns a newly created set/bag with the same comparator with the original set/bag, and the same elements, except that the elements equal to *elt* (in terms of set/bag's comparator) is replaced by *elt*. If the original set/bag doesn't contain an element equal to *elt*, the original one is returned.

```
(let ((s (set string-ci-comparator "ABC" "def")))
  (set->list (set-replace s "abc")))
  ⇒ ("abc" "def")
```

`set-delete` *set elt ...* [Function]
`bag-delete` *bag elt ...* [Function]

[R7RS set] {`scheme.set`} Returns a newly created set or bag that has the same comparator and the same elements in the original set/bag, except that the item which is equal to *elt*.

`set-delete-all` *set elt-list* [Function]
`bag-delete-all` *bag elt-list* [Function]

[R7RS set] {`scheme.set`} Returns a newly created set or bag with the same comparator of the original set/bag, with the elements of the original set/bag except the ones listed in *elt-list*.

`set-adjoin!` *set elt ...* [Function]
`bag-adjoin!` *bag elt ...* [Function]
`set-replace!` *set elt* [Function]
`bag-replace!` *bag elt* [Function]
`set-delete!` *set elt ...* [Function]
`bag-delete!` *bag elt ...* [Function]
`set-delete-all!` *set elt-list* [Function]
`bag-delete-all!` *bag elt-list* [Function]

[R7RS set] {`scheme.set`} These are the linear update versions of their counterparts. It works just like the ones without `!`, except that the original set/bag *may* be reused to produce the result, instead of new one being allocated.

Note that it's not guaranteed that the original set/bag is modified, so you should use the return value of them, instead of relying on the side effects.

`set-search!` *set elt failure success* [Function]
`bag-search!` *bag elt failure success* [Function]

[R7RS set] {`scheme.set`} Lookup-and-modify procedures. The *failure* and *success* arguments are procedures.

First, they search *elt* in the given set/bag. If an item that matches *elt* is found, the *success* procedure is called with three arguments, as follows:

```
(success item update remove)
```

The *update* argument is a procedure that takes two arguments, as `(update new-item retval)`. It replaces the matching *item* in the set/bag with *new-item*, and returns *retval*. The *remove* argument is a procedure that takes one argument, as `(remove retval)`. It removes the matching *item* in the set/bag, and returns *retval*.

If an item that matches *elt* is not found, the *failure* procedure is called with two arguments, as follows:

```
(failure insert ignore)
```

The *insert* argument is a procedure that takes one argument, as `(insert retval)`. It inserts *elt* into the set/bag, and returns *retval*. The *ignore* argument is a procedure that takes one argument, as `(ignore retval)`. It just returns *retval*.

The return values of `set-search!` and `bag-search!` is the modified set/bag (which may or may not be `eq?` to the passed one), and the value returned by *success* or *failure* procedures.

Note that *retval* isn't used in this process; it is just to provide one of the return values of `set-search!/bag-search!`, for the procedures passed to *success* or *failure* are expected to be tail-called.

If there are more than one item that matches *elt* in a bag, `bag-search!` only invokes *success* for the first item it finds. You can recurse into `bag-search!` in the *failure* procedure to visit all matching items. It is guaranteed that *success* and *failure* procedures are tail-called.

The whole set

`set-size` *set* [Function]
`bag-size` *bag* [Function]

[R7RS set] {`scheme.set`} Returns the number of items in the set/bag.

`set-find` *pred set failure* [Function]
`bag-find` *pred bag failure* [Function]

[R7RS set] {`scheme.set`} Apply *pred* on each item in the set/bag, and returns the first item on which *pred* returns true. Since sets and bags are unordered, if there are more than one items that satisfy *pred*, you won't know which one will be returned.

If there're no items that satisfy *pred*, a thunk `failure` is called and its result is returned.

`set-count` *pred set* [Function]
`bag-count` *pred bag* [Function]

[R7RS set] {`scheme.set`} Returns the number of items that satisfy *pred* in the set/bag.

`set-any?` *pred set* [Function]
`bag-any?` *pred bag* [Function]

[R7RS set] {`scheme.set`} Returns true iff any item in the set/bag satisfy *pred*.

`set-every?` *pred set* [Function]
`bag-every?` *pred bag* [Function]

[R7RS set] {`scheme.set`} Returns true iff every item in the set/bag satisfy *pred*.

Mapping and folding

`set-map` *comparator proc set* [Function]
`bag-map` *comparator proc bag* [Function]

[R7RS set] {`scheme.set`} Create and return a new set/bag with the comparator *comparator*, whose items are calculated by applying *proc* to each element in the original set/bag.

`set-for-each` *proc set* [Function]
`bag-for-each` *proc bag* [Function]

[R7RS set] {`scheme.set`} Apply *proc* to each element in the set/bag. The result of *proc* is ignored. Return value is undefined.

`set-fold` *proc seed set* [Function]
`bag-fold` *proc seed bag* [Function]

[R7RS set] {`scheme.set`} For each item in the set/bag, call *proc* with two arguments, an item and a seed value. What *proc* returns becomes the next seed value. The *seed* argument gives the initial seed value, and the last return value of *proc* will be the result of `set-fold/bag-fold`.

```
(bag-fold + 0 (bag eqv-comparator 1 1 2 2 3 3 4 4))
  ⇒ 20
```

`set-filter` *pred set* [Function]
`bag-filter` *pred bag* [Function]

[R7RS set] {`scheme.set`} Returns a newly created set/bag with the same comparator of the original set/bag, and its content consists of items from the original set/bag that satisfy *pred*.

```
(set->list (set-filter odd? (set eqv-comparator 1 2 3 4 5)))
  ⇒ (1 3 5)
```

`set-remove` *pred set* [Function]
`bag-remove` *pred bag* [Function]

[R7RS set] {`scheme.set`} Returns a newly created set/bag with the same comparator of the original set/bag, and its content consists of items from the original set/bag that does not satisfy *pred*.

```
(set->list (set-remove odd? (set eqv-comparator 1 2 3 4 5)))
  ⇒ (2 4)
```

`set-partition` *pred set* [Function]
`bag-partition` *pred bag* [Function]

[R7RS set] {`scheme.set`} Returns two sets or bags, both have the same comparator of the original set or bag. The first one consists of the items from the original set/bag that satisfy *pred*, and the second one consists of the items that don't.

```
(receive (in out) (set-remove odd? (set eqv-comparator 1 2 3 4 5))
  (values (set->list in)
          (set->list out)))
  ⇒ (1 3 5) and (2 4)
```

`set-filter!` *pred set* [Function]
`bag-filter!` *pred bag* [Function]
`set-remove!` *pred set* [Function]
`bag-remove!` *pred bag* [Function]
`set-partition!` *pred set* [Function]
`bag-partition!` *pred bag* [Function]

[R7RS set] {`scheme.set`} Linear update versions of their counterparts (the procedures without `!`). They work like their respective counterpart, but they are allowed (but not required) to reuse the original set/bag to produce the result(s).

Note that it is not guaranteed that the original set/bag is modified, so you have to use the return value(s) instead of relying on the side effects.

Copying and conversion

`set-copy` *set* [Function]
`bag-copy` *bag* [Function]

[R7RS set] {`scheme.set`} Returns a copy of the set/bag.

`set->list` *set* [Function]
`bag->list` *bag* [Function]

[R7RS set] {`scheme.set`} Returns a list of all items in the set/bag. Since sets and bags are unordered, there's no guarantee on the order of items.

`list->set` *comparator elt-list* [Function]
`list->bag` *comparator elt-list* [Function]

[R7RS set] {`scheme.set`} Creates a set or a bag with the given comparator, and the list of element. Functionally equivalent to the followings:

```
(apply set comparator elt-list)
(apply bag comparator elt-list)
```

`list->set!` *set elt-list* [Function]
`list->bag!` *bag elt-list* [Function]

[R7RS set] {`scheme.set`} Add items in *elt-list* to the existing set/bag, and returns the updated set/bag. The original set/bag is also modified. Functionally equivalent to the followings:

```
(apply set-adjoin! set elt-list)
(apply bag-adjoin! bag elt-list)
```

`bag->set` *bag* [Function]
`set->bag` *set* [Function]

[R7RS set] {`scheme.set`} Conversions between a bag and a set. Returns a newly created bag or set, respectively.

If *bag* has duplicated items, `bag->set` coerces them to one item.

`set->bag!` *bag set* [Function]

[R7RS set] {`scheme.set`} Adds all items in *set* to *bag*, and returns *bag*. Both *bag* and *set* must have the same comparator.

`bag->alist` *bag* [Function]

[R7RS set] {`scheme.set`} Returns a list of `(item . count)`, where *item* is an item in *bag*, and *count* is the number of that item in the bag.

`alist->bag` *comparator alist* [Function]

[R7RS set] {`scheme.set`} Creates a new bag with *comparator*, and fills it according to *alist*, which must be a list of `(item . count)`.

If there's duplicate items in *alist*, only fist one counts.

Subsets

`set=?` *set1 set2 . . .* [Function]
`bag=?` *bag1 bag2 . . .* [Function]

[R7RS set] {`scheme.set`} Returns true iff all sets/bags have exactly same items.

The comparators of the argument sets/bags are not checked, but assumed to be the same, in terms of the equality of items.

set<? *set1 set2 . . .* [Function]
bag<? *bag1 bag2 . . .* [Function]
set>? *set1 set2 . . .* [Function]
bag>? *bag1 bag2 . . .* [Function]
set<=? *set1 set2 . . .* [Function]
bag<=? *bag1 bag2 . . .* [Function]
set>=? *set1 set2 . . .* [Function]
bag>=? *bag1 bag2 . . .* [Function]

[R7RS set] {scheme.set} Returns true iff each preceding set/bag is a proper subset of, a proper superset of, a subset of, or a superset of the following set/bags, respectively.

Again, the comparators of the argument sets/bags are not checked, but assumed to be the same, in terms of the equality of items.

Set theory operations

set-union *set1 set2 . . .* [Function]
bag-union *bag1 bag2 . . .* [Function]

[R7RS set] {scheme.set} Returns a newly allocated set or bag which is a union of all the sets/bags.

set-intersection *set1 set2 . . .* [Function]
bag-intersection *bag1 bag2 . . .* [Function]

[R7RS set] {scheme.set} Returns a newly allocated set or bag which is an intersection of all the sets/bags.

set-difference *set1 set2 . . .* [Function]
bag-difference *bag1 bag2 . . .* [Function]

[R7RS set] {scheme.set} Returns a newly created set or bag that contains items in *set1*/*bag1* except those are also in *set2*/*bag2* . . . .

```
(sort (set->list (set-difference (set eqv-comparator 1 2 3 4 5 6 7)
                                 (set eqv-comparator 3 5 7 9 11 13)
                                 (set eqv-comparator 4 8 16 32))))
  ⇒ (1 2 6)
```

set-xor *set1 set2* [Function]
bag-xor *bag1 bag2* [Function]

[R7RS set] {scheme.set} Returns a newly created set or bag that consists of items that are either in *set1*/*bag1* or *set2*/*bag2*, but not in both.

```
(sort (set->list (set-xor (set eqv-comparator 2 3 5 7 11 13 17)
                          (set eqv-comparator 3 5 7 9 11 13 15))))
  ⇒ (2 9 15 17)
```

set-union! *set1 set2 . . .* [Function]
bag-union! *bag1 bag2 . . .* [Function]
set-intersection! *set1 set2 . . .* [Function]
bag-intersection! *bag1 bag2 . . .* [Function]
set-difference! *set1 set2 . . .* [Function]
bag-difference! *bag1 bag2 . . .* [Function]
set-xor! *set1 set2* [Function]
bag-xor! *bag1 bag2* [Function]

[R7RS set] {scheme.set} Linear update versions of their corresponding procedures. Those procedures works like their !-less counterparts, except that they are allowed to, but not required to, reuse *set1*/*bag1* to produce the result.

The caller should always use the returned set/bag instead of relying on the side effects.

Bag-specific procedures

`bag-sum` *bag1 bag2 ...* [Function]
`bag-sum!` *bag1 bag2 ...* [Function]

[R7RS set] {`scheme.set`} Returns a bag that gathers all the items in given bags, counting duplicates. The functional version `bag-sum` always creates new bag to return. The linear update version `bag-sum!` is allowed to, but not required to, modify *bag1* to produce the result.

```
(sort (bag->list (bag-sum (bag eqv-comparator 1 1 2 4 5 5 6)
                          (bag eqv-comparator 3 3 5 9))))
  ⇒ (1 1 2 3 3 4 5 5 5 6 9)
```

Note the difference from `bag-union`:

```
(sort (bag->list (bag-union (bag eqv-comparator 1 1 2 4 5 5 6)
                            (bag eqv-comparator 3 3 5 9))))
  ⇒ (1 1 2 3 3 4 5 5 6 9)
```

`bag-product` *n bag* [Function]
`bag-product!` *n bag* [Function]

[R7RS set] {`scheme.set`} Returns a bag that contains every item as *n*-times many as the original bag. A fresh bag is created and returned by `bag-product`, while `bag-product!` may reuse *bag* to produce the result.

```
(sort (bag->list (bag-product 2 (bag eq-comparator 'a 'b 'r 'a))))
  ⇒ (a a a a b b r r)
```

`bag-unique-size` *bag* [Function]

[R7RS set] {`scheme.set`} Returns the number of unique elements in *bag*.

```
(bag-unique-size (bag eqv-comparator 1 1 2 2 3 3 4))
 ⇒ 4
```

`bag-element-count` *bag elt* [Function]

[R7RS set] {`scheme.set`} Returns the number of specified element *elt* in *bag*.

```
(bag-element-count (bag eqv-comparator 1 1 2 2 2 3 3) 2)
 ⇒ 3
```

`bag-for-each-unique` *proc bag* [Function]

[R7RS set] {`scheme.set`} For each unique item in *bag*, calls `proc` with two arguments: The item, and the count of the item in the bag.

`bag-fold-unique` *proc seed bag* [Function]

[R7RS set] {`scheme.set`} For each unique item in *bag*, calls *proc* with three arguments: The item, the count of the item, and the previous seed value. The *seed* argument provides the initial seed value; the result of *proc* is used for the next seed value, and the last result of `proc` is returned from `bag-fold-unique`.

```
(sort (bag-fold-unique acons '()
        (bag equal-comparator "a" "a" "b" "b" "b" "c" "d"))
      string<? car)
 ⇒ (("a" . 2) ("b" . 3) ("c" . 1) ("d" . 1))
```

`bag-increment!` *bag elt count* [Function]
`bag-decrement!` *bag elt count* [Function]

[R7RS set] {`scheme.set`} Linear update *bag* to increase or decrease the count of *elt* in it by *count*, which must be an exact integer. Note that the element count won't get below zero; if a bag has two `a`'s, and you call `(bag-decrement! bag 'a 100)`, you get a bag with zero `a`'s.

Comparators

`set-comparator` [Comparator]
`bag-comparator` [Comparator]

[R7RS comparator] {`scheme.set`} Comparators to be used to compare sets or bags. They don't provide comparison procedure, for you cannot define a total order among sets or bags. They do provide hash functions.

10.3.6 `scheme.charset` - R7RS character sets

`scheme.charset` [Module]

Implements character set library, originally defined as SRFI-14. Note that the following character-set procedures and pre-defined charsets are Gauche's build-in. See Section 6.11 [Character set], page 147.

```
char-set              char-set?              char-set-contains?
char-set-copy         char-set-complement    char-set-complement!

char-set:lower-case   char-set:upper-case    char-set:title-case
char-set:letter       char-set:digit         char-set:letter+digit
char-set:graphic      char-set:printing      char-set:whitespace
char-set:iso-control  char-set:punctuation   char-set:symbol
char-set:hex-digit    char-set:blank         char-set:ascii
char-set:empty        char-set:full
```

In Gauche, the `<char-set>` class inherits `<collection>` and implements the collection protocol, so that the generic operations defined in `gauche.collection` can also be used (see Section 9.5 [Collection framework], page 344).

10.3.6.1 Character-set constructors

`list->char-set` *char-list :optional base-cs* [Function]
`list->char-set!` *char-list base-cs* [Function]

[R7RS comparator] {`scheme.charset`} Constructs a character set from a list of characters *char-list*. If *base-cs* is given, it must be a character set, and the characters in it are added to the result character set. `List->char-set!` is allowed, but not required, to reuse *base-cs* to store the result.

`string->char-set` *s :optional base-cs* [Function]
`string->char-set!` *s base-cs* [Function]

[R7RS charset] {`scheme.charset`} Like `list->char-set` and `list->char-set!`, but take a list of characters from a string *s*.

`char-set-filter` *pred char-set :optional base-cs* [Function]
`char-set-filter!` *pred char-set base-cs* [Function]

[R7RS charset] {`scheme.charset`} Returns a character set containing every character *c* in *char-set* such that `(pred c)` returns true. If a character set *base-cs* is given, its content is added to the result. The linear update version `char-set-filter!` is allowed, but not required, to modify *base-cs* to store the result.

`ucs-range->char-set` *lower upper :optional error? base-cs* [Function]
`ucs-range->char-set!` *lower upper error? base-cs* [Function]

[R7RS charset] {`scheme.charset`} Creates a character set containing every character whose ISO/IEC 10646 UCS-4 code lies in the half-open range [*lower*,*upper*).

`integer-range->char-set` *lower upper :optional error? base-cs* [Function]
`integer-range->char-set!` *lower upper error? base-cs* [Function]
{`scheme.charset`}

`->char-set` *x* [Function]
[R7RS charset] {`scheme.charset`} A convenience function to coerce various kinds of objects to a char-set. The argument *x* can be a collection of characters, a char-set, or a character. If the argument is a char-set, it is returned as-is. If the argument is a character, a char-set with that single character is returned.

Note: R7RS `(scheme charset)`'s `->char-set` only accepts a string, a char-set or a character as an argument. Gauche extends it so that it can accept any collection of characters.

10.3.6.2 Character-set comparison

`char-set=` *char-set1 ...* [Function]
[R7RS charset] {`scheme.charset`} Returns `#t` iff all the character sets have exactly the same members.

```
(char-set=)  ⇒ #t
(char-set= (char-set)) ⇒ #t
(char-set= (string->char-set "cba")
           (list->char-set #\a #\b #\c))
  ⇒ #t
```

`char-set<=` *char-set1 ...* [Function]
[R7RS charset] {`scheme.charset`} Returns `#t` iff every char-set argument is a subset of the following char-sets. If no arguments are given, `#t` is returned.

`char-set-hash` *char-set :optional bound* [Function]
[R7RS charset] {`scheme.charset`} Returns a non-negative exact integer as a hash value of *char-set*. If optional *bound* argument is given, it must be a positive integer that limits the range of the hash value, which will fall between 0 to `(- bound 1)`, inclusive.

10.3.6.3 Character-set iteration

`char-set-cursor` *char-set* [Function]
[R7RS charset] {`scheme.charset`} Returns an object that can point to a first character within *char-set* (here, 'first' merely means the beginning of iteration; the order of iteration is implementation-dependent and you can't assume a specific order). The caller must treat the return value as an opaque object; it can only be used to pass as the *cursor* argument of `char-set-ref`, `char-set-cursor-next` and `end-of-char-set?`.

`char-set-ref` *char-set cursor* [Function]
[R7RS charset] {`scheme.charset`} Returns a character in *char-set* pointed by *cursor*.

The *cursor* argument must be an object returned from `char-set-cursor` or `char-set-cursor-next` with *char-set*. The behavior is undefined if *cursor* is not a cursor created from *char-set*.

`char-set-cursor-next` *char-set cursor* [Function]
[R7RS charset] {`scheme.charset`} Returns a new cursor for the next character pointed by *cursor* within *char-set*.

The *cursor* argument must be an object returned from `char-set-cursor` or `char-set-cursor-next` with *char-set*. The behavior is undefined if *cursor* is not a cursor created from *char-set*.

`end-of-char-set?` *cursor* [Function]

[R7RS charset] {`scheme.charset`} Returns `#t` iff *cursor* points to the end of the charset.

`char-set-fold` *kons knil char-set* [Function]

[R7RS charset] {`scheme.charset`} Iterate over all characters in *char-set*, calling the procedure *kons* with a character and the seed value. The return value of *kons* becomes the next seed value, while *knil* gives the first seed value. Returns the last seed value. The order of traversal isn't specified.

`char-set-unfold` *pred fun gen seed :optional base-char-set* [Function]
`char-set-unfold!` *pred fun gen seed base-char-set* [Function]

[R7RS charset] {`scheme.charset`} Build a character set by calling *fun* on the seed value repeatedly. For each iteration, first *pred* is applied to the current seed value. If it returns true, a character set that gathers characters generated so far is returned. Otherwise, *fun* is called on the seed value, which must return a character. Then *gen* is called on the seed value to obtain the next seed value.

If *base-char-set* is given, a union of it and the generated characters is returned. The linear-update version `char-set-unfold!` may modify *base-char-set* to create the result.

`char-set-for-each` *proc char-set* [Function]

[R7RS charset] {`scheme.charset`} Applies *proc* on each character in *char-set*. The return value of *proc* is discarded. Returns undefined value.

`char-set-map` *proc char-set* [Function]

[R7RS charset] {`scheme.charset`} Applies *proc* on each character in *char-set*, which must return a character. Returns a new charset consists of the characters returned from *proc*.

10.3.6.4 Character-set query

`char-set-every` *pred char-set* [Function]
`char-set-any` *pred char-set* [Function]
`char-set-count` *pred char-set* [Function]

[R7RS charset] {`scheme.charset`} These procedures apply *pred* to each character in *char-set*.

`char-set-every` returns `#f` as soon as *pred* returns `#f`. Otherwise, it returns the result of the last application of *pred*.

`char-set-any` returns as soon as *pred* returns a true value, and the return value is the one *pred* returns. If *pred* returns `#f` for all characters, `#f` is returned.

`char-set-count` returns the number of times *pred* returns a true value.

Note that *char-set* can be huge (e.g. a complement of small char-set), which can make these procedures take very long.

`char-set->list` *char-set* [Function]
`char-set->string` *char-set* [Function]

[R7RS charset] {`scheme.charset`} Returns a list of each character, or a string consisting of each character, in *char-set*, respectively. Be careful to apply this on a large char set.

10.3.6.5 Character-set algebra

`char-set-adjoin` *char-set char1 . . .* [Function]
`char-set-adjoin!` *char-set char1 . . .* [Function]

[R7RS charset] {`scheme.charset`} Returns a character set that adds *char1 . . .* to *char-set*.

The linear update version `char-set-adjoin!` may modify *char-set*.

`char-set-delete` *char-set char1 ...* [Function]
`char-set-delete!` *char-set char1 ...* [Function]

[R7RS charset] {`scheme.charset`} Returns a character set that removes *char1 ...* from *char-set*. It is noop if *char-set* doesn't have *char1 ...*.

The linear update version `char-set-delete!` may modify *char-set*.

`char-set-union` *char-set ...* [Function]
`char-set-union!` *char-set1 char-set2 ...* [Function]

[R7RS charset] {`scheme.charset`} Returns a character set of all characters in any one of *char-set ...*. Without arguments, returns an empty charset.

The linear update version `char-set-union!` may modify *char-set1*.

`char-set-intersection` *char-set ...* [Function]
`char-set-intersection!` *char-set1 char-set2 ...* [Function]

[R7RS charset] {`scheme.charset`} Returns a character set of every character that is in all of *char-set ...*. Without arguments, returns `char-set:full`.

The linear update version `char-set-intersection!` may modify *char-set1*.

`char-set-difference` *char-set1 char-set2 ...* [Function]
`char-set-difference!` *char-set1 char-set2 ...* [Function]

[R7RS charset] {`scheme.charset`} Returns a character set of every character that is in *char-set1* but not in any of *char-set2 ...*.

The linear update version `char-set-difference!` may modify *char-set1*.

`char-set-xor` *char-set ...* [Function]
`char-set-xor!` *char-set1 char-set2 ...* [Function]

[R7RS charset] {`scheme.charset`} With zero arguments, returns an empty charset. With one argument, it is returned as is. With two arguments, returns a character set of every character that is in either one of two sets, but not in both. With more than two arguments, it returns `(char-set-xor (char-set-xor set1 set2) set3 ...)`.

The linear update version `char-set-xor!` may modify *char-set1*.

`char-set-diff+intersection` *char-set1 char-set2 ...* [Function]
`char-set-diff+intersection!` *char-set1 char-set2 char-set3 ...* [Function]

[R7RS charset] {`scheme.charset`} Returns two values, the result of `(char-set-difference char-set1 char-set2 ...` and the result of `(char-set-intersection char-set1 char-set2 ...`.

10.3.7 `scheme.hash-table` - R7RS hash tables

`scheme.hash-table` [Module]

This module provides hash table library, originally defined as `srfi-125`.

Hash table provided with this module is the same as Gauche's built-in hash table. However, srfi-125 introduces procedures that conflict with Gauche's original procedures, so Gauche provides those procedures built-in but under aliases. See Section 6.15 [Hashtables], page 177, for the built-in hash table procedures.

With this module, procedures are provided as defined in R7RS. Use this module when you're writing portable code.

Srfi-125 also defines compatibility procedures with srfi-69, although saying they're deprecated. Those deprecated procedures are supported in this module, too.

The following procedures are the same as Gauche's built-in.

```
hash-table-unfold   hash-table?       hash-table-contains?
```

```
hash-table-exists?  hash-table-empty? hash-table=?
hash-table-mutable? hash-table-ref    hash-table-ref/default
hash-table-set!     hash-table-update!/default
hash-table-clear!   hash-table-size   hash-table-keys
hash-table-values   hash-table-copy   hash-table-empty-copy
hash-table->alist   hash-table-union! hash-table-intersection!
hash-table-difference!                hash-table-xor!
```

See Section 6.15 [Hashtables], page 177, for the description of those procedures.

The following procedures are also provided as Gauche's built-in, but with `-r7` suffix.

```
hash-table         hash-table-delete! hash-table-intern!
hash-table-update! hash-table-pop!    hash-table-find
hash-table-count   hash-table-map     hash-table-for-each
hash-table-map!    hash-table-map->list
hash-table-prune!
```

`make-hash-table` *comparator arg . . .* [Function]
`make-hash-table` *equal-proc hash-proc arg . . .* [Function]

[R7RS hash-table] {`scheme.hash-table`} This enhances built-in `make-hash-table` with the second form, that takes two procedures instead of one comparator, as srfi-69.

In the srfi-69 form, equal-proc is a procedure taking two keys to see if they are the same, and hash-proc is a procedure taking a key to calculate its hash value (nonnegative fixnum). The compatibility form is deprecated and should be avoided in the new code.

The optional *arg dots* are ignored in Gauche.

`hash-table` *cmpr key value . . .* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-r7` (see Section 6.15 [Hashtables], page 177).

Construct a new hash table with key-comparator *cmpr*. It is populated by *key value* . . ., which is a list with keys and values appear alternatively. It is an error if the length of key-value list is not even.

Note that srfi-125 defines this procedure to return an immutable hash table if the implementation supports one. Gauche doesn't provide immutable hash tables (we do have immutable map instead, see Section 12.11 [Immutable map], page 654), but when you're writing a portable program, be careful not to modify the table returned by this procedure.

`alist->hash-table` *alist cmpr arg . . .* [Function]
`alist->hash-table` *equal-proc hash-proc cmpr arg . . .* [Function]

[R7RS hash-table] {`scheme.hash-table`} This enhances built-in `alist->hash-table` with the second form, that takes two procedures instead of one comparator, as srfi-69.

In the srfi-69 form, equal-proc is a procedure taking two keys to see if they are the same, and hash-proc is a procedure taking a key to calculate its hash value (nonnegative fixnum). The compatibility form is deprecated and should be avoided in the new code.

The optional *arg dots* are ignored in Gauche.

`hash-table-delete!` *ht key . . .* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-delete!-r7`.

Deletes entries associated with the given *key*s from the table *ht*. It is ok if *ht* doesn't have *key*. Returns the number of entries that are actually deleted.

It differs from built-in `hash-table-delete!` in two points: The built-in one can take exactly one *key*, and returns a boolean indicating if the entry is actually deleted.

`hash-table-intern!` *ht key failure* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-intern!-r7`.

Search *key* in *ht*. If it is found, returns the associated value. If it is not found, call *failure* without artuments, and insert a new entry associating *key* and the value *failure* returns, and returns that new value.

`hash-table-update!` *ht key updater :optional failure success* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-update!-r7`. It takes different optional arguments from built-in `hash-table-update!`.

Updater is a procedure that takes one argument, *failure* is a thunk, and *success* is a procedure that takes one argument.

Works the same as follows, except maybe more efficiently.

```
(hash-table-set! ht key (updater (hash-table-ref ht key failure success)))
```

`hash-table-pop!` *ht* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-pop!-r7`. It is a completely different procedure as built-in `hash-table-pop!`.

Removes an arbitrary entry in the hash table *ht*, and returns the removed entry's key and value as two values.

If *ht* is empty, an error is signalled.

`hash-table-find` *proc ht failure* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-find-r7`. It takes different arguments from built-in `hash-table-find`.

Calls *proc* with a key and a value of each entry in *ht*, until *proc* returns non-false value. If *proc* returns non-false value, `hash-table-find` immediately returns it. If *proc* returns `#f` for all entries, calls a thunk *failure* and returns its result.

`hash-table-count` *ht* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-count-r7`.

Calls *proc* with a key and a value of each entry in *ht*, and returns the number of times when *proc* returned true.

`hash-table-map` *proc cmpr ht* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-map-r7`. This is different from built-in `hash-table-map`.

Creates a fresh hashtable with a key comparator *cmpr*, then populate it by inserting the key and the result of applying *proc* on the value of each entry in *ht*.

`hash-table-map!` *proc ht* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-map!-r7`.

Calls *proc* on the value of each entry in *ht*, and update the entry with the result of *proc*.

`hash-table-map->list` *proc ht* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-map->list-r7`, and same as built-in `hash-table-map` (not the `scheme.hash-table`'s `hash-table-map`) except the order of the arguments.

Apply *proc* on a key and a value of each entry in *ht*, in arbitrary order, and returns a list of results.

`hash-table-for-each` *proc ht* [Function]
`hash-table-for-each` *ht proc* [Function]

[R7RS hash-table] {`scheme.hash-table`} Apply *proc* on a key and a value of each entry in *ht*. The result of *proc* is discarded. Returns an unspecified value.

This procedure allows arguments in both order for the compatibility— the first way is the `scheme.hash-table` recommended one, which is the same as built-in `hash-table-for-each-r7`, and the latter way is compatible with srfi-69, which is the same as built-in `hash-table-for-each`.

It is unfortunate that this compatibility thing is extremely confusing; especially in Gauche, you can make anything applicable, so the distinction between procedures and other objects is blurred.

We recommend that you stick to one way or another within a module; if your module uses built-in interface, use `(hash-table-for-each ht proc)`. If your module imports `scheme.hash-table`, use `(hash-table-for-each proc ht)`.

`hash-table-fold` *proc seed ht* [Function]
`hash-table-fold` *ht proc seed* [Function]

[R7RS hash-table] {`scheme.hash-table`} The *proc* argument takes three arguments, a key, a value, and the current seed value. The procedure applies *proc* for each entry in *ht*, using *seed* as the first seed value, and using the previous result of *proc* as the subsequent seed value. Returns the result of the last call of *seed*.

This procedure allows arguments in both order for the compatibility— the first way is the `scheme.hash-table` recommended one, which is the same as built-in `hash-table-fold-r7`, and the latter way is compatible with srfi-69, which is the same as built-in `hash-table-fold`.

It is unfortunate that this compatibility thing is extremely confusing. We recommend that you stick to one way or another within a module; if your module uses built-in interface, use the second interface. If your module imports `scheme.hash-table`, use the first interface.

`hash-table-prune!` *proc ht* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as built-in `hash-table-prune!-r7`.

Apply *proc* on a key and a value of each entry in *ht*, and deletes the entry if *proc* returns a true value. This procedure returns an unspecified value.

`hash-table-merge!` *ht1 ht2* [Function]

[R7RS hash-table] {`scheme.hash-table`} This is the same as `hash-table-union!`, and provided just for the compatibility with srfi-69. Deprecated.

`hash` *obj :optional ignore* [Function]
`string-hash` *obj :optional ignore* [Function]
`string-ci-hash` *obj :optional ignore* [Function]
`hash-by-identity` *obj :optional ignore* [Function]

[R7RS hash-table] {`scheme.hash-table`} Provided for the compatibility with srfi-69, and are deprecated.

The first three are the same as built-in `default-hash`, `string-hash`, and `string-ci-hash`, except that these accept an optional second argument, which is ignored. Note that `hash-by-identity` is also defined as the same as `default-hash` except the ignored optional second argument, per srfi-125, although the name suggests that it would work as if `eq-hash`.

Do not use these procedures in the new code; you can use comparators instead (`default-comparator`, `string-comparator`, `string-ci-comparator` and `eq-comparator`, see Section 6.2.4.3 [Predefined comparators], page 106). If you do need hash function, you should still avoid `hash` and `hash-by-identity`, and use `default-hash` and `eq-hash` instead.

`hash-table-equivalence-function` *ht* [Function]
`hash-table-hash-function` *ht* [Function]

[R7RS hash-table] {`scheme.hash-table`} Provided for the compatibility with srfi-69, and are deprecated.

Returns the equivalence function and hash function of a hash table *ht*.

For the introspection, we recommend to use built-in `hash-table-comparator`. (Unfortunately, it is not included in `scheme.hash-table`, though.)

10.3.8 `scheme.ideque` - R7RS immutable deques

`scheme.ideque` [Module]

This module provides a functional double-ended queue (deque, pronounced as "deck"), with amortized O(1) access of queue operations on either end.

It also serves as a convenient bidirectional list structures in a sense that operations from the end of the list is just as efficient as the ones from the front.

Note: If you don't need immutability and wants space-efficient deque, you can also use `data.ring-buffer` as a deque (see Section 12.14 [Ring buffer], page 665).

This module was originally defined as `srfi-134`, then became a part of R7RS large.

Gauche's `data.ideque` is a superset of this module. See Section 12.10 [Immutable deques], page 654.

`ideque` *element* . . . [Function]

[R7RS ideque] {`scheme.ideque`} Returns an ideque with the given elements.

`ideque-unfold` *p f g seed* [Function]

[R7RS ideque] {`scheme.ideque`}

`ideque-unfold-right` *p f g seed* [Function]

[R7RS ideque] {`scheme.ideque`}

`ideque-tabulate` *size init* [Function]

[R7RS ideque] {`scheme.ideque`}

`ideque?` *idq* [Function]

[R7RS ideque] {`scheme.ideque`}

`ideque-empty?` *idq* [Function]

[R7RS ideque] {`scheme.ideque`}

`ideque-add-front` *idq x* [Function]
`ideque-add-back` *idq x* [Function]

[R7RS ideque] {`scheme.ideque`}

`ideque-front` *idq* [Function]
`ideque-back` *idq* [Function]

[R7RS ideque] {`scheme.ideque`}

`ideque-remove-front` *idq* [Function]
`ideque-remove-back` *idq* [Function]

[R7RS ideque] {`scheme.ideque`}

`ideque-reverse` *idq* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque=` *idq idq2 . . .* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-ref` *idq n* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-take` *idq n* [Function]
`ideque-take-right` *idq n* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-drop` *idq n* [Function]
`ideque-drop-right` *idq n* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-split-at` *idq n* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-length` *idq* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-append` *idq . . .* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-zip` *idq idq2 . . .* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-map` *proc idq . . .* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-filter-map` *proc idq . . .* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-for-each` *proc idq . . .* [Function]
`ideque-for-each-right` *proc idq . . .* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-fold` *proc knil idq . . .* [Function]
`ideque-fold-right` *proc knil idq . . .* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-append-map` *proc idq . . .* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-filter` *pred idq* [Function]
`ideque-remove` *pred idq* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-partition` *pred idq* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-find` *pred idq :optional failure* [Function]
`ideque-find-right` *pred idq :optional failure* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-take-while` *pred idq* [Function]
`ideque-take-while-right` *pred idq* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-drop-while` *pred idq* [Function]
`ideque-drop-while-right` *pred idq* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-span` *pred idq* [Function]
`ideque-break` *pred idq* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque-any` *pred idq . . .* [Function]
`ideque-every` *pred idq . . .* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque->list` *idq* [Function]
`list->ideque` *list* [Function]
[R7RS ideque] {`scheme.ideque`}

`ideque->generator` *idq* [Function]
`generator->ideque` *gen* [Function]
[R7RS ideque] {`scheme.ideque`}

10.3.9 `scheme.generator` - R7RS generators

`scheme.generator` [Module]

This module provides generators and accumulators. They were first defined in srfi-121, then enhanced in srfi-158, and finally incorporated R7RS large as `(scheme generator)`.

A generator is a thunk to generate a sequence of values, potentially terminated by EOF. Procedures to deal with generators are provided by `gauche.generator`, which is a superset of srfi-121. See Section 9.11 [Generators], page 372, for the details.

An accumulator is an opposite of generators. They are procedures that work as consumers. An accumulator takes one argument. When non-eof value is given, the value is stored, and when EOF is given, the accumulated value is returned. How the values are accumulated depends on the accumulator.

Once EOF is given, the accumulator is "finalized". Subsequent EOF makes it return the same accumulated value. It is undefined if other values are passed after EOF is passed.

The accumulator can be used to parameterize procedures that yield aggregate objects. Consider the following procedure, which takes items from two generators and accumulate them alternatively. (Note that `glet*` is Gauche's procedure but not in srfi-158).

```
(define (intertwine acc gen1 gen2)
  (let loop ()
    (glet* ([a (gen1)]
            [b (gen2)])
      (acc a)
      (acc b)
      (loop)))
  (acc (eof-object)))
```

The procedure can return various type of collections, without knowing the actual type—the passed accumulator determines it.

```
(intertwine (list-accumulator) (giota 5) (giota 5 100))
```

```
    ⇒ (0 100 1 101 2 102 3 103 4 104)
  (intertwine (vector-accumulator) (giota 5) (giota 5 100))
    ⇒ #(0 100 1 101 2 102 3 103 4 104)
  (intertwine (bytevector-accumulator) (giota 5) (giota 5 100))
    ⇒ #u8(0 100 1 101 2 102 3 103 4 104)
```

Note: In Gauche, you can also use classes to parameterize returned container types (e.g. `map-to`), for many collection classes support *builder protocol*. See Section 9.5 [Collection framework], page 344, for the details. Accumulator has the flexibility that you can provide more than one ways to construct return value on the same type (e.g. forward and reverse list).

The following generator procedures are explained in `gauche.generator` section (see Section 9.11 [Generators], page 372):

- Section 9.11.1 [Generator constructors], page 372:

`generator`	`circular-generator`	`make-iota-generator`
`make-range-generator`	`make-coroutine-generator`	
`make-unfold-generator`	`make-for-each-generator`	
`list->generator`	`vector->generator`	`reverse-vector->generator`
`string->generator`	`bytevector->generator`	

- Section 9.11.2 [Generator operations], page 376:

`gcons*`	`gappend`	`gflatten`
`ggroup`	`gmerge`	`gmap`
`gcombine`	`gfilter`	`gremove`
`gstate-filter`	`ggroup`	
`gtake`	`gdrop`	`gtake-while`
`gdrop-while`	`gdelete`	`gdelete-neighbor-dups`
`gindex`	`gselect`	

- Section 9.11.3 [Generator consumers], page 381:

`generator->list`	`generator->reverse-list`	`generator-map->list`
`generator->vector`	`generator->vector!`	`generator->string`
`generator-count`	`generator-any`	`generator-every`
`generator-unfold`		

- Section 6.18.9 [Folding generated values], page 197:

`generator-fold`	`generator-for-each`	`generator-find`

The following are accumulator procedures:

`make-accumulator` *kons knil finalizer* [Function]

[R7RS generator] {`scheme.generator`} Creates and returns an accumulator with a state, whose initial value is *knil*. When non-EOF value *v* is passed to the accumulator, *kons* is called as `(kons v state)`, and its result becomes the new state value. When EOF value is passed, `(finalizer state)` is called and its result becomes the result of accumulator.

`list-accumulator` [Function]
`reverse-list-accumulator` [Function]

[R7RS generator] {`scheme.generator`} Creates and returns accumulators that return accumulated value as a list, in the accumulated order (`list-accumulator`) or the reverse order (`reverse-list-accumulator`).

`vector-accumulator` [Function]
`reverse-vector-accumulator` [Function]

`bytevector-accumulator` [Function]

[R7RS generator] {`scheme.generator`} Returns accumulators that return accumulated value as a fresh vector or bytevector (u8vector), in the accumulated order (`vector-accumulator`, `bytevector-accumulator`) or the reverse order (`reverse-vector-accumulator`). There's no `reverse-bytevector-accumulator`.

`vector-accumulator!` *vec at* [Function]
`bytevector-accumulator!` *bvec at* [Function]

[R7RS generator] {`scheme.generator`} The *vec* or *bvec* argument is a mutable vector or bytevector (u8vector), and is used as a buffer.

Returns an accumulator that stores the accumulated values in the buffer, starting from the index *at*. It is an error if the accumulator gets more values after the buffer reaches at the end.

Once EOF is passed to the accumulator, *vec* or *bvec* is returned, respectively.

`string-accumulator` [Function]

[R7RS generator] {`scheme.generator`} Returns an accumulator that accepts characters and accumulates them to a string.

`sum-accumulator` [Function]
`product-accumulator` [Function]
`count-accumulator` [Function]

[R7RS generator] {`scheme.generator`} Returns accumulators that yield a scalar value.

The accumulator created by `sum-accumulator` and `product-accumulator` accepts numbers, and keep adding or multiplying it with the accumulated value (the default value is 0 and 1, respectively).

The accumulator created by `count-accumulator` accepts any objects and just counting it.

10.3.10 `scheme.lseq` - R7RS lazy sequences

`scheme.lseq` [Module]

This module provides lightweight lazy sequence (lseq), conceptually represented by a pair of element and generator. When the rest of sequence is taken, the generator is evaluated and yields another pair of element and generator, and so on. The overhead is one allocation of a pair per element. It is much lighter than streams (see Section 12.71 [Stream library], page 809), which requires to create a thunk for every element.

Gauche already has built-in support for such lazy sequences; we go further to make it behave like ordinary pairs—that is, if you take cdr of a lazy pair, we automatically forces the generator so it is indistinguishable from an ordinary pair, modulo side effects. See Section 6.19.2 [Lazy sequences], page 200.

Srfi-127, the original srfi for this module, is a bit ambiguous whether its lazy sequence *must* be implemented with a pair whose cdr is a generator procedure, or it refers to the pair+generator as a conceptual model. Considering of the purpose of lazy sequence, the concrete implementation shouldn't matter; that is, the user of lazy sequence should not count on the fact that the lseq is an improper list terminated by a generator procedure. Instead, an lseq should be treated as an opaque object that can be passed to `scheme.lseq` procedures.

With that premise, we implement this module as just a thin wrapper of Gauche's native lazy sequence. It is upper-compatible, except that the code that assumes the internal structure could break. Notably, the constructor `generator->lseq` is the same as Gauche's built-in, which returns Gauche's lseq, undistinguishable to the ordinary list.

```
(procedure? (generator->lseq (generator 1)))
  ;; => #t, in srfi-127 reference implementation,
  ;;    #f, in our implementation.
```

`lseq?` *x* [Function]

[R7RS lseq] {`scheme.lseq`} Returns true iff *x* is an object that can be passed to lseq procedures. In Gauche, it returns `#t` if *x* is a pair or an empty list, since a lazy pair is indistinguishable from a pair.

`lseq=?` *elt=? lseq1 lseq2* [Function]

[R7RS lseq] {`scheme.lseq`} Compare two lseqs element-wise using `elt=?` and returns `#t` iff two lseqs are equal.

`lseq-car` *lseq* [Function]
`lseq-first` *lseq* [Function]

[R7RS lseq] {`scheme.lseq`} Returns the first item of *lseq*. If *lseq* is empty, an error is raised. In Gauche, these are just aliases of `car`.

`lseq-cdr` *lseq* [Function]
`lseq-rest` *lseq* [Function]

[R7RS lseq] {`scheme.lseq`} Returns the rest of *lseq*. If *lseq* is empty, an error is raised. In Gauche, these are just aliases of `cdr`.

`lseq-take` *lseq k* [Function]
`lseq-drop` *lseq k* [Function]

[R7RS lseq] {`scheme.lseq`} Returns an lseq that has first `k` items, or an lseq that skips first `k` items, respectively.

An error is signaled when the resulting lseq of `lseq-take` reached at the end of sequence before `k` items are taken. It is different from Gauche's `ltake`, which simply returns `()` in such case.

On the other hand, `lseq-drop` is the same as `drop` in Gauche; it just drops *k* items from the head of input sequence, regardless of whether it is an ordinary list or lseq.

`lseq-realize` *lseq* [Function]

[R7RS lseq] {`scheme.lseq`} Realizes all the elements in *lseq*, resulting an ordinary list.

`lseq->generator` *lseq* [Function]

[R7RS lseq] {`scheme.lseq`} Creates a generator from *lseq*. In Gauche, this is same as `list->generator`.

`lseq-length` *lseq* [Function]

[R7RS lseq] {`scheme.lseq`} Returns the length of *lseq*. All the elements in *lseq* are realized as the side effect. In Gauche, this is same as `length`.

`lseq-append` *lseq lseq2 . . .* [Function]

[R7RS lseq] {`scheme.lseq`} Append one or more *lseq*s lazily. This is the same as `lappend` in Gauche.

`lseq-zip` *lseq lseq2 . . .* [Function]

[R7RS lseq] {`scheme.lseq`} Returns a lazy sequence in which the first element is a list of first elements of `lseq`s, and so on.

`lseq-map` *proc lseq lseq2 . . .* [Function]

[R7RS lseq] {`scheme.lseq`} Lazy map. The same as Gauche's `lmap`. Returns a lazy sequence.

`lseq-for-each` *proc lseq lseq2 . . .* [Function]

[R7RS lseq] {`scheme.lseq`} This one consumes all the input lseqs, applying *proc* on each corresponding elements of the input sequences for the side effects. In Gauche, it is the same as `for-each`, for Gauche doesn't distinguish lseqs and ordinary lists.

`lseq-filter` *pred lseq* [Function]
`lseq-remove` *pred lseq* [Function]

[R7RS lseq] {`scheme.lseq`} Returns an lseq that contains elements from the input *lseq* that satisfy or don't satisfy *pred*, respectively. `Lseq-filter` is the same as Gauche's `lfilter`.

`lseq-take-while` *pred lseq* [Function]
`lseq-drop-while` *pred lseq* [Function]

[R7RS lseq] {`scheme.lseq`} These are the same as Gauche's `ltake-while` and `drop-while` (the latter doesn't have `l`-prefix, since it just drops items from the head of the input sequence, regardless of whether it is an ordinary list or an lseq.

`lseq-find` *pred lseq* [Function]
`lseq-find-tail` *pred lseq* [Function]
`lseq-any` *pred lseq* [Function]
`lseq-every` *pred lseq* [Function]
`lseq-index` *pred lseq* [Function]
`lseq-member` *pred lseq :optional eq* [Function]
`lseq-memq` *pred lseq* [Function]
`lseq-memv` *pred lseq* [Function]

[R7RS lseq] {`scheme.lseq`} In Gauche, these are the same as the corresponding list functions, `find`, `find-tail`, `any`, `every`, `list-index`, `member`, `memq` and `memv`, respectively, for all of those functions won't look at input more than necessary so *lseq*s work just as well as ordinary lists.

10.3.11 `scheme.stream` - R7RS stream

`scheme.stream` [Module]

This module provides utilities for lazily evaluated streams. It is more heavyweight than lazy sequences (see Section 6.19.2 [Lazy sequences], page 200), but it strictly implements "as lazy as possible" semantics—elements are never evaluated until it is actually accessed.

The following procedures are provided in Gauche's `util.stream` module; see Section 12.71 [Stream library], page 809, for their description:

```
stream-null          stream-cons         stream?            stream-null?
stream-pair?         stream-car          stream-cdr         stream-lambda

define-stream        list->stream        port->stream
stream->list         stream-append       stream-concat      stream-constant
stream-drop-while    stream-filter       stream-fold
stream-for-each      stream-from         stream-iterate     stream-length
stream-let           stream-map          stream-match       stream-of
stream-range         stream-ref          stream-reverse     stream-scan
stream-take-while    stream-unfold       stream-unfolds     stream-zip
```

The following macro and procedures have different interface from Gauche's `util.stream` module:

```
stream               stream-take       stream-drop
```

`stream` *expr . . .* [Macro]

[R7RS stream] {`scheme.stream`} Returns a new stream whose elements are the result of *expr* . . . . Arguments won't be evaluated until required.

This differs from srfi-40 and `util.stream`'s `stream`, which is a procedure so arguments are evaluated (see Section 12.71.2 [Stream constructors], page 810, for the details).

`stream-take` *n stream* [Function]
`stream-drop` *n stream* [Function]

[R7RS stream] {`scheme.stream`} Returns a stream that contains first *n* elements from *stream*, or elements without first *n* elements from it, respectively. If *stream* has less than *n* elements, `stream-take` returns a copy of the entire *stream*, while `stream-drop` returns a null stream.

Note that the argument order doesn't follow the Scheme tradition, which takes the main object (*stream* in this case) first, then the count. Procedures with the same name is provided in `util.stream` with the different argument order.

10.3.12 `scheme.box` - R7RS boxes

`scheme.box` [Module]

This module defines the box datatype, which is a simple container that can hold one Scheme object. It can be used as a minimal data storage, or a sort of mutable indirect "pointer".

Traditionally a pair (with ignoring its cdr) or a single-element vector has been used for this purpose; in modern Scheme you can also define a record type with one mutable field. Nevertheless, a box is very common abstraction to describe various algorithms, and having common interface to it is useful.

The srfi leaves some details to implementations. Here are our choices:

- We don't support autoboxing; that is, it is an error to pass non-box value to the procedure expecting boxed value and vice versa.
- Comparing two boxes with `equal?` compares their contents when two are not `eqv?`. In the spec, when two boxes are `eqv?` then they must also be `equal?` to each other, but it's up to the implementation when two are not `eqv?`.

When you're writing portable code, be careful not to depend on the `equal?` behavior.

`box` *val* [Function]

[R7RS box] {`scheme.box`} Returns a fresh box object that contains the value `val`.

`box?` *obj* [Function]

[R7RS box] {`scheme.box`} Returns `#t` iff *obj* is a box object.

`unbox` *box* [Function]

[R7RS box] {`scheme.box`} Returns *box*'s content.

`set-box!` *box val* [Function]

[R7RS box] {`scheme.box`} Mutate *box*'s content with *val*. Returns unspecified value.

10.3.13 `scheme.list-queue` - R7RS list queues

`scheme.list-queue` [Module]

A library of simple queue based on lists. Gauche has a queue support in `data.queue` module, which also includes MT-safe queue (see Section 12.12 [Queue], page 656). This library is implemented on top of `data.queue`'s `<queue>` object and mainly provided for portable code.

The list-queue is just an instance of `<queue>`, so you can pass a queue created by `make-queue` to `scheme.list-queue` API and a list-queue created by `make-list-queue` to Gauche's queue API.

Note: Some API of this library requires to return internal pairs the queue uses, for the efficiency. The pair's car/cdr will be mutated by subsequent queue operation, and also any mutation done on the pair would cause inconsistency in the original queue.

`make-list-queue` *lis :optional last* [Function]

[R7RS list-queue] {`scheme.list-queue`} Creates and returns a list-queue whose initial content is *lis*. In Gauche, a list queue is just an instance of `<queue>` (see Section 12.12 [Queue], page 656).

The cells in *lis* are owned by the queue; the caller shouldn't mutate it afterwards, nor assume its structure remains the same.

The optional *last* argument must be the last pair of *lis*. If it is passed, `make-list-queue` will skip scanning *lis* and just hold a reference to *last* as the tail of the queue.

`list-queue` *elt ...* [Function]

[R7RS list-queue] {`scheme.list-queue`} Creates and returns a list-queue whose initial content is *elt* .... In Gauche, a list queue is just an instance of `<queue>` (see Section 12.12 [Queue], page 656).

`list-queue-copy` *queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns a copy of a list-queue *queue*.

`list-queue-unfold` *p f g seed :optional queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Prepend *queue* with the items generated by `(unfold p f g seed)` and returns the updated queue. See Section 10.3.1 [R7RS lists], page 512, for `unfold`. If *queue* is omitted, a fresh queue is created.

```
(list-queue-unfold (pa$ = 5) ; p
                   (pa$ * 2) ; f
                   (pa$ + 1) ; g
                   0         ; seed
                   (list-queue 'x 'y 'z))
  ⇒ a queue containing (0 2 4 6 8 x y z)
```

`list-queue-unfold-right` *p f g seed :optional queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Append *queue* with the items generated by `(unfold-right p f g seed)` and returns the updated queue. See Section 10.3.1 [R7RS lists], page 512, for `unfold-right`. If *queue* is omitted, a fresh queue is created.

```
(list-queue-unfold-right (pa$ = 5) ; p
                         (pa$ * 2) ; f
                         (pa$ + 1) ; g
                         0         ; seed
                         (list-queue 'x 'y 'z))
  ⇒ a queue containing (x y z 8 6 4 2 0)
```

`list-queue?` *obj* [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns true iff *queue* is a list-queue. In Gauche, it is the same as `queue?` in the `data.queue` module.

`list-queue-empty?` *queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns true iff *queue* is empty. Same as `queue-empty?` of `data.queue`.

`list-queue-front` *queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns the front element of the *queue*. An error is thrown if *queue* is empty. Same as `queue-front` of `data.queue`.

`list-queue-back` *queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns the rear element of the *queue*. An error is thrown if *queue* is empty. Same as `queue-rear` of `data.queue`.

`list-queue-list` *queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns the internal list of *queue*. Note that the list would be modified by subsequent operations of *queue*, and any modification on the list would make *queue* inconsistent. The primary purpose of this procedure is to implement other queue-related operations with small overhead.

If you merely need a cheap access the content of the queue, consider `list-queue-remove-all!`. That returns the list of elements of the queue without copying, and simultaneously reset the queue to empty, so it's safe.

`list-queue-fist-last` *queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns two values, the first and last pair of *queue*. If the queue is empty, two empty lists are returned.

This also returns the internal pair of the queue, so any subsequent operations of *queue* would change the contents of the pairs, and any modification on the pairs would make *queue* inconsistent. The purpose of this procedure is to implement other queue-related operations with small overhead. This procedure should not be used in general.

`list-queue-add-front!` *queue elt* [Function]

[R7RS list-queue] {`scheme.list-queue`} Add *elt* to the front of *queue*. Same as `(queue-push! queue elt)` of `data.queue`.

`list-queue-add-back!` *queue elt* [Function]

[R7RS list-queue] {`scheme.list-queue`} Add *elt* to the back of *queue*. Same as `(enqueue! queue elt)` of `data.queue`.

`list-queue-remove-front!` *queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Remove an element from the front of *queue* and returns the removed element. Throws an error if *queue* is empty. Same as `dequeue!` of `data.queue`.

`list-queue-remove-back!` *queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Remove an element from the back of *queue* and returns the removed element. Throws an error if *queue* is empty. This isn't guaranteed to be efficient; it is O(n) operation where n is the number of elements. In general, if you need this operation frequently, you should consider double-ended queue. (See Section 12.10 [Immutable deques], page 654, and also see Section 12.14 [Ring buffer], page 665.)

`list-queue-remove-all!` *queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Remove all the elements from *queue* and returns them as a list. The list isn't copied—this is O(1) operation. This should be preferred over `list-queue-list`, for it's safer. In Gauche, this is the same as `dequeue-all!` in `data.queue`.

`list-queue-set-list!` *queue lis :optional last* [Function]

[R7RS list-queue] {`scheme.list-queue`} Modify *queue* to have the elements in *lis* as its element. The original content of *queue* is discarded. If the optional *last* argument is provided, it must be the last pair of *lis*, and the procedure uses that instead of scanning *lis*, to achieve O(1) operation.

After calling this, *lis* is owned by *queue* and it may be mutated. The caller shouldn't change, or rely on *lis* afterwards.

`list-queue-append` *queue* ... [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns a fresh list-queue whose contents are concatenation of *queue*s. The contents of arguments are intact. This is O(n) operation where n is the total number of elements.

`list-queue-append!` *queue* ... [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns a list-queue whose contents are concatenation of *queue*s. During the operation, the contents of *queue*s may be mutated, and they shouldn't be used any longer. (In Gauche, to avoid accident, we actually empty all the *queue*s.) It is also noted that the result doesn't need to be `eq?` to any of the arguments. This is O(m) operation where m is the total number of queues (as opposed to the number of elements).

`list-queue-concatenate` *queues* [Function]

[R7RS list-queue] {`scheme.list-queue`} `(apply list-queue-append queues)`.

`list-queue-map` *proc queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Returns a fresh list-queue whose elements are obtained by applying *proc* on every elements in *queue*.

`list-queue-map!` *proc queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Replaces every element in *queue* by the result of application of *proc* on the element.

`list-queue-for-each` *proc queue* [Function]

[R7RS list-queue] {`scheme.list-queue`} Applies *proc* on every element of *queue*. The results are discarded.

10.3.14 `scheme.ephemeron` - R7RS ephemeron

`scheme.ephemeron` [Module]

This module defined *ephemerons*, a weak reference structure to hold key-value association. This is originally defined as srfi-142.

Gauche supports weak pointers in the form of weak vectors (see Section 6.17 [Weak pointers], page 186), but it is known that a simple weak pointer (a single pointer that doesn't prevent the pointed object from being collected) isn't enough to implement weak key-value association such as mappings.

An ephemeron is a record that points to a key and an associated datum, with the following characteristics:

1. Reference to the key is weak; it doesn't prevent the key from being collected if there's no strong reference to the key, *except from the datum associated by the ephemeron.*
2. Reference to the datum is also a kind of weak; it doesn't prevent the datum from being collected if there's no strong reference to the datum, *and there's no strong reference to the associated key.* Note that the datum is retained as long as the key is retained, even there's no strong reference to the datum itself.

Implementing the proper ephemeron requires deep integration with the GC. At this moment, Gauche's ephemeron is implemented separately from GC, and has the following limitations:

- If the datum has a strong reference to the associated key, the key won't be collected even if there's no other strong reference to it.
- After the key is collected and there's no strong reference to the datum, `ephemeron-broken?` needs to be called in order to trigger the collection of the datum.

Since the timing of collection isn't specified in the spec, Gauche's implementation still conforms srfi-142, but in practice you need to be aware of these limitations. Eventually we want to support full ephemeron integrated with GC.

Once the key and/or the datum is collected (we call such ephemeron "broken"), referencing them returns a bogus value. The proper way to use an ephemeron *e* is the following pattern:

```
(let ([k (ephemeron-key e)]
```

```
          [d (ephemeron-datum e)])
      (if (ephemeron-broken? e)
        (... k and d are invalid ...)
        (... k and d are valid ...)))
```

You should take values, then check if the ephemeron isn't broken yet. If you call `ephemeron-broken?` first, there's a chance that the ephemeron is broken between the check and the time you reference it.

`make-ephemeron` *key datum* [Function]

[R7RS ephemeron] {`scheme.ephemeron`} Create a new ephemeron associating the *key* to the *datum*.

`ephemeron?` *obj* [Function]

[R7RS ephemeron] {`scheme.ephemeron`} Returns `#t` iff *obj* is an ephemeron.

`ephemeron-key` *ephemeron* [Function]
`ephemeron-datum` *ephemeron* [Function]

[R7RS ephemeron] {`scheme.ephemeron`} Returns the key and the datum of *ephemeron*, respectively. If the ephemeron is already broken, there's no guarantee on what is returned. Thus you should always call `ephemeron-broken?` *after* calling these procedure to ensure the values are meaningful. See the `scheme.ephemeron` entry for the details.

`ephemeron-broken?` *ephemeron* [Function]

[R7RS ephemeron] {`scheme.ephemeron`} Returns `#t` iff *ephemeron* has been broken, that is, its key and/or datum may be collected and cannot be reliably retrieved. See the `scheme.ephemeron` entry for the details.

`reference-barrier` *key* [Function]

[R7RS ephemeron] {`scheme.ephemeron`} This procedure does nothing by itself, but guarantees *key* is strongly reference until returning from this procedure.

10.3.15 `scheme.comparator` - R7RS comparators

`scheme.comparator` [Module]

This module defines comparators and related procedures. Originally called `srfi-128`.

Gauche supports comparators fully compatible to `scheme.comparator` built-in. See Section 6.2.4 [Basic comparators], page 103, for the following procedures defined in this module.

```
comparator? comparator-ordered? comparator-hashable?
make-comparator make-pair-comparator
make-list-comparator make-vector-comparator
make-eq-comparator make-eqv-comparator make-equal-comparator

boolean-hash char-hash char-ci-hash string-hash
string-ci-hash symbol-hash number-hash
hash-bound hash-salt

make-default-comparator default-hash
comparator-register-default!

comparator-type-test-predicate comparator-equality-predicate
comparator-ordering-predicate comparator-hash-function
comparator-test-type comparator-check-type comparator-hash
```

```
=? <? >? <=? >=? comparator-if<=>
```

10.3.16 scheme.regex - R7RS regular expressions

scheme.regex [Module]

This module provides operations on Scheme Regular Expressions (SRE). Originally defined as srfi-115.

Note that support for this module is not complete. Grapheme syntax is still missing.

Scheme regular expression syntax

Syntax summary

SRE is just an S-expression with the structure summarized below.

With the exception of `or`, any syntax that takes multiple `<sre>` processes them in a sequence. In other words `(foo <sre> ...)` is equivalent to `(foo (seq <sre> ...))`.

Note: SRE uses the symbol `|` for alteration, but the vertical bar character is used for symbol escape in Gauche (and R7RS), so you have to write such symbol as `|\||`. We recommend to use `or` instead.

```
<sre> ::=
 | <string>                    ; A literal string match.
 | <cset-sre>                  ; A character set match.
 | (* <sre> ...)               ; 0 or more matches.
 | (zero-or-more <sre> ...)
 | (+ <sre> ...)               ; 1 or more matches.
 | (one-or-more <sre> ...)
 | (? <sre> ...)               ; 0 or 1 matches.
 | (optional <sre> ...)
 | (= <n> <sre> ...)           ; <n> matches.
 | (exactly <n> <sre> ...)
 | (>= <n> <sre> ...)          ; <n> or more matches.
 | (at-least <n> <sre> ...)
 | (** <n> <m> <sre> ...)      ; <n> to <m> matches.
 | (repeated <n> <m> <sre> ...)

 | (|\||  <sre> ...)           ; Alternation.
 | (or <sre> ...)

 | (:   <sre> ...)             ; Sequence.
 | (seq <sre> ...)
 | ($ <sre> ...)               ; Numbered submatch.
 | (submatch <sre> ...)
 | (-> <name> <sre> ...)               ;  Named submatch.  <name> is
 | (submatch-named <name> <sre> ...)   ;  a symbol.

 | (w/case   <sre> ...)        ; Introduce a case-sensitive context.
 | (w/nocase <sre> ...)        ; Introduce a case-insensitive context.

 | (w/unicode   <sre> ...)     ; Introduce a unicode context.
 | (w/ascii <sre> ...)         ; Introduce an ascii context.
```

```
  | (w/nocapture <sre> ...)     ; Ignore all enclosed submatches.

  | bos                         ; Beginning of string.
  | eos                         ; End of string.

  | bol                         ; Beginning of line.
  | eol                         ; End of line.

  | bow                         ; Beginning of word.
  | eow                         ; End of word.
  | nwb                         ; A non-word boundary.
  | (word <sre> ...)            ; An SRE wrapped in word boundaries.
  | (word+ <cset-sre> ...)      ; A single word restricted to a cset.
  | word                        ; A single word.

  | (?? <sre> ...)              ; A non-greedy pattern, 0 or 1 match.
  | (non-greedy-optional <sre> ...)
  | (*? <sre> ...)              ; Non-greedy 0 or more matches.
  | (non-greedy-zero-or-more <sre> ...)
  | (**? <m> <n> <sre> ...)     ; Non-greedy <m> to <n> matches.
  | (non-greedy-repeated <sre> ...)
  | (atomic <sre> ...)          ; Atomic clustering.

  | (look-ahead <sre> ...)      ; Zero-width look-ahead assertion.
  | (look-behind <sre> ...)     ; Zero-width look-behind assertion.
  | (neg-look-ahead <sre> ...)  ; Zero-width negative look-ahead assertion.
  | (neg-look-behind <sre> ...) ; Zero-width negative look-behind assertion.

  | (backref <n-or-name>)       ; Match a previous submatch.
```

The grammar for `cset-sre` is as follows.

```
<cset-sre> ::=
  | <char>                      ; literal char
  | "<char>"                    ; string of one char
  | <char-set>                  ; embedded SRFI 14 char set
  | (<string>)                  ; literal char set
  | (char-set <string>)
  | (/ <range-spec> ...)        ; ranges
  | (char-range <range-spec> ...)
  | (or <cset-sre> ...)         ; union
  | (|\|| <cset-sre> ...)
  | (and <cset-sre> ...)        ; intersection
  | (& <cset-sre> ...)
  | (- <cset-sre> ...)          ; difference
  | (- <difference> ...)
  | (~ <cset-sre> ...)          ; complement of union
  | (complement <cset-sre> ...)
  | (w/case <cset-sre>)         ; case and unicode toggling
  | (w/nocase <cset-sre>)
  | (w/ascii <cset-sre>)
  | (w/unicode <cset-sre>)
```

```
    | any | nonl | ascii | lower-case | lower
    | upper-case | upper | title-case | title
    | alphabetic | alpha | alphanumeric | alphanum | alnum
    | numeric | num | punctuation | punct | symbol
    | graphic | graph | whitespace | white | space
    | printing | print | control | cntrl | hex-digit | xdigit

   <range-spec> ::= <string> | <char>
```

Basic patterns

`<string>`

A literal string.

```
(regexp-search "needle" "hayneedlehay") => #<regexp-match>
(regexp-search "needle" "haynEEdlehay") => #f
```

`(seq <sre> ...)`
`(: <sre> ...)`

A sequence of patterns that should be matched in the same order. This is the same as RE syntax `(?:re...)`

```
(regexp-search '(: "one" space "two" space "three") "one two three") => #<reg
```

`(or <sre> ...)`
`(|\|| <sre> ...)`

Matches one of the given patterns. This is the same as RE syntax `pattern1|pattern2|...`

(regexp-search '(or "eeney" "meeney" "miney") "meeney") => #<regexp-match>
(regexp-search '(or "eeney" "meeney" "miney") "moe") => #f

`(w/nocase <sre> ...)`

Changes to match the given patterns case-insensitively. Sub-patterns can still be made sensitive with `w/case`. This is the same as RE syntax `(?i:re...)`

```
(regexp-search "needle" "haynEEdlehay") => #f
(regexp-search '(w/nocase "needle") "haynEEdlehay") => #<regexp-match>

(regexp-search '(~ ("Aab")) "B") => #<regexp-match>
(regexp-search '(~ ("Aab")) "b") => #f
(regexp-search '(w/nocase (~ ("Aab"))) "B") => #f
(regexp-search '(w/nocase (~ ("Aab"))) "b") => #f
(regexp-search '(~ (w/nocase ("Aab"))) "B") => #f
(regexp-search '(~ (w/nocase ("Aab"))) "b") => #f
```

`(w/case <sre> ...)`

Changes to match the given patterns case-sensitively. Sub-patterns can still be made case-insensitive. This is the same as RE syntax `(?-i:re...)`. This is the default.

```
(regexp-search '(w/nocase "SMALL" (w/case "BIG")) "smallBIGsmall") => #<rege
(regexp-search '(w/nocase (~ (w/case ("Aab")))) "b") => #f
```

`(w/ascii <sre> ...)`

Limits the character sets and other predefined patterns to ASCII. This affects patterns or character sets like `any`, `alpha`, `(word)`...

```
(regexp-search '(w/ascii bos (* alpha) eos) "English") => #<regexp-match>
(regexp-search '(w/ascii bos (* alpha) eos) "Ελληνική") => #f
```

`(w/unicode <sre> ...)`
: Changes the character sets and other predefined patterns back to Unicode if `w/ascii` has been used in the outer scope. This is the default.

```
(regexp-search '(w/unicode bos (* alpha) eos) "English") => #<regexp-match>
(regexp-search '(w/unicode bos (* alpha) eos) "Ελληνική") => #<regexp-match>
```

`(w/nocapture <sre> ...)`
: Disables capturing for all `submatch` and `submatch-named` inside.

```
(let ((number '($ (+ digit))))
  (cdr
   (regexp-match->list
    (regexp-search `(: ,number "-" ,number "-" ,number)
                   "555-867-5309")))  ; => '("555" "867" "5309")
  (cdr
   (regexp-match->list
    (regexp-search `(: ,number "-" (w/nocapture ,number) "-" ,number)
                   "555-867-5309"))))   => '("555" "5309")
```

Repeating patterns

`(optional <sre> ...)`
`(? <sre> ...)`
: Matches the pattern(s) one or zero times.

```
(regexp-search '(: "match" (? "es") "!") "matches!") => #<regexp-match>
(regexp-search '(: "match" (? "es") "!") "match!") => #<regexp-match>
(regexp-search '(: "match" (? "es") "!") "matche!") => #f
```

`(zero-or-more <sre> ...)`
`(* <sre> ...)`
: Matches the pattern(s) zero or more times.

```
(regexp-search '(: "<" (* (~ #\>)) ">") "<html>") => #<regexp-match>
(regexp-search '(: "<" (* (~ #\>)) ">") "<>") => #<regexp-match>
(regexp-search '(: "<" (* (~ #\>)) ">") "<html") => #f
```

`(one-or-more <sre> ...)`
`(+ <sre> ...)`
: Matches the pattern(s) at least once.

```
(regexp-search '(: "<" (+ (~ #\>)) ">") "<html>") => #<regexp-match>
(regexp-search '(: "<" (+ (~ #\>)) ">") "<a>") => #<regexp-match>
(regexp-search '(: "<" (+ (~ #\>)) ">") "<>") => #f
```

`(at-least n <sre> ...)`
`(>= n <sre> ...)`
: Matches the pattern(s) at least `n` times.

```
(regexp-search '(: "<" (>= 3 (~ #\>)) ">") "<table>") => #<regexp-match>
(regexp-search '(: "<" (>= 3 (~ #\>)) ">") "<pre>") => #<regexp-match>
(regexp-search '(: "<" (>= 3 (~ #\>)) ">") "<tr>") => #f
```

`(exactly n <sre> ...)`
`(= n <sre> ...)`
: Matches the pattern(s) exactly `n` times.

```
(regexp-search '(: "<" (= 4 (~ #\>)) ">") "<html>") => #<regexp-match>
(regexp-search '(: "<" (= 4 (~ #\>)) ">") "<table>") => #f
```

(repeated from to <sre> ...)
(** from to <sre> ...)

Matches the pattern(s) at least from times and up to to times.

```
(regexp-search '(: (= 3 (** 1 3 numeric) ".") (** 1 3 numeric)) "192.168.1.1
(regexp-search '(: (= 3 (** 1 3 numeric) ".") (** 1 3 numeric)) "192.0168.1.
```

Submatch Patterns

(submatch <sre> ...)
($ <sre> ...)

Captures the matched string. Each capture is numbered increasing from one (capture zero is the entire matched string). For nested captures, the numbering scheme is depth-first walk.

(submatch-named <name> <sre> ...)
(-> <name> <sre> ...)

Captures the matched string and assigns a name to it in addition to a number. This is the equivalent of (?<*name*>*re*...)

(backref <n-or-name>)

Matches a previously matched submatch. This is the same as RE syntax *n* or \k<*name*>.

Character Sets

<char>

A character set contains a single character.

```
(regexp-matches '(* #\-) "---") => #<regexp-match>
(regexp-matches '(* #\-) "-_-") => #f
```

"<char>"

A character set contains a single character. This is technically ambiguous with SRE matching a literal string. However the end result of both syntaxes is the same.

<char-set>

A SRFI-14 character set.

Note that while currently there is no portable written representation of SRFI 14 character sets, you can use Gauche reader syntax #[char-set-spec], see Section 6.11 [Character set], page 147.

```
(regexp-partition `(+ ,char-set:vowels) "vowels")
  => ("v" "o" "w" "e" "ls")
```

(char-set <string>)
(<string>)

A character set contains the characters in the given string. This is the same as `(char-set ,(string->char-set <string>)).

```
(regexp-matches '(* ("aeiou")) "oui") => #<regexp-match>
(regexp-matches '(* ("aeiou")) "ouais") => #f
(regexp-matches '(* ("e\x0301")) "e\x0301") => #<regexp-match>
(regexp-matches '("e\x0301") "e\x0301") => #f
(regexp-matches '("e\x0301") "e") => #<regexp-match>
(regexp-matches '("e\x0301") "\x0301") => #<regexp-match>
(regexp-matches '("e\x0301") "\x00E9") => #f
```

(char-range <range-spec> ...)
(/ <range-spec> ...)

A character set contains the characters within <range-set>. This is the same as RE syntax [].

```
(regexp-matches '(* (/ "AZ09")) "R2D2") => #<regexp-match>
(regexp-matches '(* (/ "AZ09")) "C-3PO") => #f
```

(or <cset-sre> ...)
(|\|| <cset-sre> ...)

A shorthand for `(char-set ,(char-set-union <cset-sre>...)).

(complement <cset-sre> ...)
(~ <cset-sre> ...)

A shorthand for `(char-set ,(char-set-complement <cset-sre>...)).

(difference <cset-sre> ...)
(- <cset-sre> ...)

A shorthand for `(char-set ,(char-set-difference <cset-sre>...)).

```
(regexp-matches '(* (- (/ "az") ("aeiou"))) "xyzzy") => #<regexp-match>
(regexp-matches '(* (- (/ "az") ("aeiou"))) "vowels") => #f
```

(and <cset-sre> ...)
(& <cset-sre> ...)

A shorthand for `(char-set ,(char-set-intersection <cset-sre>...)).

```
(regexp-matches '(* (& (/ "az") (~ ("aeiou")))) "xyzzy") => #<regexp-match>
(regexp-matches '(* (& (/ "az") (~ ("aeiou")))) "vowels") => #f
```

(w/case <cset-sre>)
(w/nocase <cset-sre>)
(w/ascii <cset-sre>)
(w/unicode <cset-sre>)

This is similar to the SRE equivalent, listed to indicate that they can also be applied on character sets.

Named Character Sets

Note that if w/ascii is in effect, these character sets will return the ASCII subset. Otherwise they return full Unicode ones.

any

Matches any character. This is the . in regular expression.

nonl

Matches any character other than #\return or #\newline.

ascii

A shorthand for `(char-set ,char-set:ascii).

lower-case
lower

A shorthand for `(char-set ,char-set:lower-case).

upper-case
upper

A shorthand for `(char-set ,char-set:upper-case).

`title-case`
`title`
: A shorthand for `` `(char-set ,char-set:title-case) ``.

`alphabetic`
`alpha`
: A shorthand for `` `(char-set ,char-set:letter) ``.

`numeric`
`num`
: A shorthand for `` `(char-set ,char-set:digit) ``.

`alphanumeric`
`alphanum`
`alnum`
: A shorthand for `` `(char-set ,char-set:letter+digit) ``.

`punctuation`
`punct`
: A shorthand for `` `(char-set ,char-set:punctuation) ``.

`symbol`
: A shorthand for `` `(char-set ,char-set:symbol) ``.

`graphic`
`graph`
: A shorthand for `` `(char-set ,char-set:graphic) ``.

```
(or alphanumeric punctuation symbol)
```

`whitespace`
`white`
`space`
: A shorthand for `` `(char-set ,char-set:whitespace) ``.

`printing`
`print`
: A shorthand for `` `(char-set ,char-set:printing) ``.

`control`
`cntrl`
: A character set contains ASCII characters with from 0 to 31.

`hex-digit`
`xdigit`
: A shorthand for `` `(char-set ,char-set:hex-digit) ``.

Boundary Assertions

`bos`
`eos`
: Matches the beginning of the string. If start/end parameters are specified, matches the start or end of the substring as specified.

`bol`
`eol`
: Matches the beginning or end of a line (or the string). For single line matching, this is the same as `bos` and `eos`. A line is interpreted the same way with `read-line`.

bow
eow

Matches the beginning or the end of a word.

```
(regexp-search '(: bow "foo") "foo") => #<regexp-match>
(regexp-search '(: bow "foo") "<foo>>") => #<regexp-match>
(regexp-search '(: bow "foo") "snafoo") => #f
(regexp-search '(: "foo" eow) "foo") => #<regexp-match>
(regexp-search '(: "foo" eow) "foo!") => #<regexp-match>
(regexp-search '(: "foo" eow) "foobar") => #f
```

nwb

A shorthand for (neg-look-ahead (or bow eow)).

(word <sre> ...)

Matches the word boundary around the given SRE:

```
(: bow <sre> ... eow)
```

(word+ <cset-sre> ...)

Matches a single word composed of characters of the given characters sets:

```
(word (+ (and (or alphanumeric "_") (or cset-sre ...))))
```

word

A shorthand for (word+ any).

Non-Greedy Patterns

(non-greedy-optional <sre> ...)
(?? <sre> ...)

The non-greedy equivalent of (optional <sre>...). This is the same as RE syntax *re*??

(non-greedy-zero-or-more< <sre> ...)
(*? <sre> ...)

The non-greedy equivalent of (zero-or-more <sre>...). This is the same as RE syntax *re**?

(non-greedy-repeated <m> <n> <sre> ...)
(**? <m> <n> <sre> ...)

The non-greedy equivalent of (repeated <sre>...). This is the same as RE syntax *re*{*n*,*m*}?

(atomic <sre> ...)

Atomic clustering. Once <sre> ... matches, the match is fixed; even if the following pattern fails, the engine won't backtrack to try the alternative match in <sre> .... This is Gauche extension and is the same as RE syntax (?>*pattern*)

Look Around Patterns

(look-ahead <sre> ...)

Zero-width look-ahead assertion. Asserts the sequence matches from the current position, without advancing the position. This is the same as RE syntax (?=*pattern*)

```
(regexp-matches '(: "regular" (look-ahead " expression") " expression") "regu
(regexp-matches '(: "regular" (look-ahead " ") "expression") "regular expres
```

(look-behind <sre> ...)

Zero-width look-behind assertion. Asserts the sequence matches behind the current position, without advancing the position. It is an error if the sequence does not have a fixed length. This is the same as RE syntax (?<=*pattern*)

`(neg-look-ahead <sre> ...)`
: Zero-width negative look-ahead assertion. This is the same as RE syntax `(?!pattern)`

`(neg-look-behind <sre> ...)`
: Zero-width negative look-behind assertion. This is the same as RE syntax `(?<!pattern)`

Using regular expressions

`regexp` *re* [Function]

[R7RS regex] {`scheme.regex`} Compiles the given Scheme Regular Expression into a `<regexp>` object. If *re* is already a regexp object, the object is returned as-is.

`rx` *sre ...* [Macro]

[R7RS regex] {`scheme.regex`} A macro shorthand for `(regexp `(: sre ...))`.

`regexp->sre` *re* [Function]

[R7RS regex] {`scheme.regex`} Returns the SRE corresponding to the given given regexp object. Note that if the regexp object is not created from an SRE, it may contain features that cannot be expressed in SRE and cause an error.

`char-set->sre` *char-set* [Function]

[R7RS regex] {`scheme.regex`} Returns the SRE of the given character set. Currently this is not optimized. If you convert `any` to SRE for example, you may get an SRE listing every single character.

`valid-sre?` *obj* [Function]

[R7RS regex] {`scheme.regex`} Returns true iff *obj* can be safely passed to regexp.

`regexp?` *obj* [Function]

[R7RS regex] {`scheme.regex`} Returns true iff *obj* is a regexp.

`regexp-matches` *re str* [*start* [*end*]] [Function]

[R7RS regex] {`scheme.regex`} Returns an `<regexp-match>` object if re successfully matches the entire string *str* or optionally from *start* (inclusive) to *end* (exclusive), or `#f` is the match fails.

For convenience, *end* accepts `#f` and interprets it as the end of the string.

The regexp-match object will contain information needed to extract any submatches.

`regexp-matches?` *re str* [*start* [*end*]] [Function]

[R7RS regex] {`scheme.regex`} Similar to `regexp-matches` but returns `#t` instead of a `<regexp-match>` object.

`regexp-search` *re str* [*start* [*end*]] [Function]

[R7RS regex] {`scheme.regex`} Similar to `regexp-matches` except that *re* only has to match a substring in *str* instead.

`regexp-fold` *re kons knil str* [*finish* [*start* [*end*]]] [Function]

[R7RS regex] {`scheme.regex`} Calls the procedure *kons* for every match found in *str* with following four arguments:

- The position of the end of the last matched string.
- The `<regexp-match>` object.
- The argument *str*.
- The result of the last *kons* call or *knil* if this is the first call.

If *finish* is given, it is called after all matches with the same parameters as calling *kons* except that `#f` is passed instead of `<regexp-match>` and the result is returned. Otherwise the result of the last *kons* call is returned.

```
(regexp-fold 'word
             (lambda (i m str acc)
               (let ((s (regexp-match-submatch m 0)))
                (cond ((assoc s acc)
                       => (lambda (x) (set-cdr! x (+ 1 (cdr x))) acc))
                      (else `((,s . 1) ,@acc)))))
             '()
             "to be or not to be")
=> '(("not" . 1) ("or" . 1) ("be" . 2) ("to" . 2))
```

`regexp-extract` *re str* [*start* [*end*]] [Function]

[R7RS regex] {`scheme.regex`} Returns a list of matched string or an empty list if no matches.

```
(regexp-extract '(+ numeric) "192.168.0.1")
=> ("192" "168" "0" "1")
```

`regexp-split` *re str* [*start* [*end*]] [Function]

[R7RS regex] {`scheme.regex`} Returns a list of not matched substrings. This can be seen as the opposite of `regexp-extract` where the matched strings are removed instead of returned.

```
(regexp-split '(+ space) " fee fi  fo\tfum\n")
=> ("fee" "fi" "fo" "fum")
(regexp-split '(",;") "a,,b,")
=> ("a" "" "b" "")
(regexp-split '(* numeric) "abc123def456ghi789")
=> ("abc" "def" "ghi" "")
```

`regexp-partition` *re str* [*start* [*end*]] [Function]

[R7RS regex] {`scheme.regex`} Returns a list of all matched and not matched substrings. In other words it's the combination of `regexp-extract` and `regexp-split` where the boundary of matched strings are used to split the original string.

```
(regexp-partition '(+ (or space punct)) "")
=> ("")
(regexp-partition '(+ (or space punct)) "Hello, world!\n")
=> ("Hello" ", " "world" "!\n")
(regexp-partition '(+ (or space punct)) "¿Dónde Estás?")
=> ("" "¿" "Dónde" " " "Estás" "?")
(regexp-partition '(* numeric) "abc123def456ghi789")
=> ("abc" "123" "def" "456" "ghi" "789")
```

`regexp-replace` *re str subst* [*start* [*end* [*count*]]] [Function]

[R7RS regex] {`scheme.regex`} Returns a new string where the first matched substring is replaced with *subst*. If *count* is specified, the *count*-th match will be replaced instead of the first one.

subst can be either a string (the replacement), an integer or a symbol to refer to the capture group that will be used as the replacement, or a list of those.

The special symbols `pre` and `post` use the substring to the left or right of the match as replacement, respectively.

As a Gauche extension, *subst* could also be a procedure, which is called with the given match object and the result will be used as the replacement.

The optional parameters *start* and *end* restrict both the matching and the substitution, to the given indices, such that the result is equivalent to omitting these parameters and replacing on `(substring str start end)`. As a convenience, a value of `#f` for end is equivalent to `(string-length str)`.

```
(regexp-replace '(+ space) "one two three" "_")
=> "one_two three"
(regexp-replace '(+ space) "one two three" "_" 0 #f 0)
=> "one_two three"
(regexp-replace '(+ space) "one two three" "_" 0 #f 1)
=> "one two_three"
(regexp-replace '(+ space) "one two three" "_" 0 #f 2)
=> "one two three"
```

Note that Gauche also has a builtin procedure of the same name, but works slightly differently, see Section 6.13.2 [Using regular expressions], page 165.

`regexp-replace-all` *re str subst* [*start* [*end*]] [Function]

[R7RS regex] {`scheme.regex`} Returns a new string where all matches in *str* are replaced with *subst*. *subst* can also take a string, a number, a symbol or a procedure similar to `regexp-replace`.

```
(regexp-replace-all '(+ space) "one two three" "_")
   => "one_two_three"
```

Note that Gauche also has a builtin procedure of the same name, but works slightly differently, see Section 6.13.2 [Using regular expressions], page 165.

`regexp-match?` *obj* [Function]

[R7RS regex] {`scheme.regex`} Returns true iff *obj* is a `<regexp-match>` object.

```
(regexp-match? (regexp-matches "x" "x"))  => #t
(regexp-match? (regexp-matches "x" "y"))  => #f
```

`regexp-match-count` *regexp-match* [Function]

[R7RS regex] {`scheme.regex`} Returns the number of matches in *match* except the implicit zero full match. This is just an alias of `rxmatch-num-matches` minus one.

```
(regexp-match-count (regexp-matches "x" "x"))  => 0
(regexp-match-count (regexp-matches '($ "x") "x"))  => 1
```

`regexp-match-submatch` *regexp-match field* [Function]

[R7RS regex] {`scheme.regex`} This is an alias of `rxmatch-substring`

```
(regexp-match-submatch (regexp-search 'word "**foo**") 0)  => "foo"
(regexp-match-submatch
 (regexp-search '(: "*" ($ word) "*") "**foo**") 0)  => "*foo*"
(regexp-match-submatch
 (regexp-search '(: "*" ($ word) "*") "**foo**") 1)  => "foo"
```

`regexp-match-submatch-start` *regexp-match field* [Function]

[R7RS regex] {`scheme.regex`} This is an alias of `regexp-match-submatch-start`.

```
(regexp-match-submatch-start
 (regexp-search 'word "**foo**") 0)  => 2
(regexp-match-submatch-start
 (regexp-search '(: "*" ($ word) "*") "**foo**") 0)  => 1
(regexp-match-submatch-start
 (regexp-search '(: "*" ($ word) "*") "**foo**") 1)  => 2
```

`regexp-match-submatch-end` *regexp-match field* [Function]

[R7RS regex] {`scheme.regex`} This is an alias of `regexp-match-submatch-end`.

```
(regexp-match-submatch-end
 (regexp-search 'word "**foo**") 0)  => 5
(regexp-match-submatch-end
 (regexp-search '(: "*" ($ word) "*") "**foo**") 0)  => 6
(regexp-match-submatch-end
 (regexp-search '(: "*" ($ word) "*") "**foo**") 1)  => 5
```

`regexp-match->list` *regexp-match* [Function]

[R7RS regex] {`scheme.regex`} This is an alias of `rxmatch-substrings`

```
(regexp-match->list
 (regexp-search '(: ($ word) (+ (or space punct)) ($ word)) "cats & dogs"))
 => '("cats & dogs" "cats" "dogs")
```

10.3.17 `scheme.mapping` - R7RS mappings

`scheme.mapping` [Module]

`scheme.mapping.hash` [Module]

This module defines immutable mappings from keys to values. Originally called `srfi-146` and `srfi-146.hash`.

The `scheme.mapping` module provides *mapping* objects, where keys have total order. The `scheme.mapping.hash` module provides `hashmap` objects, where keys can be hashed.

Currently, Gauche uses built-in `<tree-map>` for the mapping object, and built-in `<hash-table>` for the hashmap object. The actual implementation may be changed in future versions, so the user must not rely on the underlying implementations.

The caller must treat mappings and hashmaps as immutable object. The modules also provide "linear update" APIs, which is *allowed* to mutate the mappings passed to the arguments, under assumption that the argument won't be used afterwards. The linear update APIs are marked with `!` at the end of the name.

10.3.17.1 Mappings

`<mapping>` [Class]

{`scheme.mapping`} On Gauche, this is just an alias of `<tree-map>`.

Constructors

`mapping` *comparator key value ...* [Function]

[R7RS mapping] {`scheme.mapping`} Creates a new mapping with the given *comparator*, whose initial content is provided by *key value ....*

The *comparator* argument must be a comparator (see Section 6.2.4 [Basic comparators], page 103).

The *key value ...* arguments must be even length, alternating keys and values.

```
(define m (mapping default-comparator 'a 1 'b 2))

(mapping-ref m 'a) ⇒ 1
(mapping-ref m 'b) ⇒ 2
```

`mapping-unfold` *p f g seed comparator* [Function]

[R7RS mapping] {`scheme.mapping`} Creates a new mapping, whose content is populated by three procedures, *p*, *f* and *g*, and a seed value *seed*, as follows.

In each iteration, we have a current seed value, whose initial value is *seed*.

First, *p*, a stop predicate, is applied to the current seed value. If it returns true, we stop iteration and returns the new mapping.

Next, *f* is applied to the current seed value. It must return two values. The first one is for a key and the second one for the value. We add this pair to the mapping.

Then, *g* is applied to the current seed value. The result becomes the seed value of the next iteration. And we iterate.

The following example creates a mapping that maps ASCII characters to their character codes:

```
(mapping-unfold (cut >= <> 128)
                (^c (values (integer->char c) c))
                (cut + <> 1)
                0
                default-comparator)
```

`mapping/ordered` *comparator key value . . .* [Function]

[R7RS mapping] {`scheme.mapping`} Similar to `mapping`, but keys are given in the ascending order w.r.t. the comparator. An implementation may use more efficient algorithm than `mapping`. In Gauche, this is the same as `mapping` at this moment.

`mapping-unfold/ordered` *p f g seed comparator* [Function]

[R7RS mapping] {`scheme.mapping`} Similar to `mapping-unfold`, but keys are generated in the ascending order w.r.t. the comparator. An implementation may use more efficient algorithm than `mapping-unfold`. In Gauche, this is the same as `mapping-unfold` at this moment.

Predicates

`mapping?` *obj* [Function]

[R7RS mapping] {`scheme.mapping`} Returns `#t` iff *obj* is a mapping object.

`mapping-empty?` *m* [Function]

[R7RS mapping] {`scheme.mapping`} *M* must be a mapping. Returns `#t` if *m* is empty, `#f` otherwise. In Gauche, this is same as `tree-map-empty?` (see Section 6.16 [Treemaps], page 182).

`mapping-contains?` *m key* [Function]

[R7RS mapping] {`scheme.mapping`} *M* must be a mapping. Returns `#t` if *m* has an entry with *key*, `#f` otherwise. In Gauche, this is same as `tree-map-exists?` (see Section 6.16 [Treemaps], page 182).

`mapping-disjoint?` *m1 m2* [Function]

[R7RS mapping] {`scheme.mapping`} Returns `#t` iff two mappings *m1* and *m2* have no keys in common. In other words, there's no such key K that satisfy both `(mapping-contains? m1 K)` and `(mapping-contains? m2 K)`.

Accessors

`mapping-ref` *m key :optional failure success* [Function]

[R7RS mapping] {`scheme.mapping`} Get the value from a mapping *m* associated with *key*, and calls *success* on the value, and returns its result. If *m* doesn't have *key*, *failure* is invoked with no arguments and its result is returned. Both *success* and *failure* is called in tail context.

When *failure* is omitted and *key* is not found, an error is signaled. When *success* is omitted, `identity` is assumed.

`mapping-ref/default` *m key default* [Function]

[R7RS mapping] {`scheme.mapping`} Returns the value associated to *key* from a mapping *m*. If *m* doesn't have *key*, *default* is returned.

`mapping-key-comparator` *m* [Function]

[R7RS mapping] {`scheme.mapping`} Returns a comparator used to compare keys in a mapping *m*. See Section 6.2.4 [Basic comparators], page 103, for the details of comparators.

Updaters

Note that the basic premise of mappings srfi is to treat mappings as immutable. Each updating operation comes with a purely functional version (without bang) and a linear update version (with bang), but the linear update version may not require to destructively modiy the passed mapping; it's merely a hint that it may reuse the argument for the efficiency. You always need to use the returned mapping as the result of update. If you use linear update versions, you shouldn't use the passed mapping afterwards, for there's no guarantee how the state of the passed mapping is.

`mapping-adjoin` *m arg . . .* [Function]
`mapping-adjoin!` *m arg . . .* [Function]

[R7RS mapping] {`scheme.mapping`} The *arg* . . . are alternating between key and value. Returns a mapping that contains all the entries in *m* plus given keys and values, with the same comparator as *m*. Linear update version `mapping-adjoin!` may destructively modify *m* to create the return value, while `mapping-adjoin` creates a new mapping.

Arguments are processed in order. If there's already an entry in *m* with the same key as given to *arg*, the original entry remains.

```
(mapping-adjoin (mapping default-comparator 'a 1 'b 2) 'c 3 'a 4 'c 5)
 ⇒ mapping with a → 1, b → 2, c → 3
```

`mapping-set` *m arg . . .* [Function]
`mapping-set!` *m arg . . .* [Function]

[R7RS mapping] {`scheme.mapping`} The *arg* . . . are alternating between key and value. Returns a mapping that contains all the entries in *m* plus given keys and values, with the same comparator as *m*. Linear update version `mapping-set!` may destructively modify *m* to create the return value, while `mapping-set` creates a new mapping.

Arguments are processed in order. If there's already an entry in *m* with the same key as given to *arg*, the new key-value pair supersedes the old one.

```
(mapping-set (mapping default-comparator 'a 1 'b 2) 'c 3 'a 4 'c 5)
 ⇒ mapping with a → 4, b → 2, c → 5
```

`mapping-replace` *m key value* [Function]
`mapping-replace!` *m key value* [Function]

[R7RS mapping] {`scheme.mapping`} If the mapping *m* has an entry of *key*, return a mapping with the value of the entry replaced for *value*. If *m* doesn't have an entry with *key*, *m* is returned unchanged.

Linear update version `mapping-replace!` may destructively modify *m* to produce the return value, while `mapping-replace` creates a new mapping.

```
(mapping-replace (mapping default-comparator 'a 1 'b 2) 'a 3)
 ⇒ mapping with a → 3, b → 2

(mapping-replace (mapping default-comparator 'a 1 'b 2) 'c 3)
 ⇒ mapping with a → 1, b → 2
```

`mapping-delete` *m key . . .* [Function]
`mapping-delete!` *m key . . .* [Function]

[R7RS mapping] {`scheme.mapping`} Returns a mapping that is the same as *m* except its entries with any of the given *key* . . . being removed. Keys that are not in *m* are ignored.

Linear update version `mapping-delete!` may destructively modify *m* to produce the return value, while `mapping-delete` creates a new mapping.

`mapping-delete-all` *m key-list* [Function]
`mapping-delete-all!` *m key-list* [Function]

[R7RS mapping] {`scheme.mapping`} Returns a mapping that is the same as *m* except its entries with any of the given keys in *key-list* being removed. Keys that are not in *m* are ignored.

Linear update version `mapping-delete-all!` may destructively modify *m* to produce the return value, while `mapping-delete-all` creates a new mapping.

`mapping-intern` *m key make-value* [Function]
`mapping-intern!` *m key make-value* [Function]

[R7RS mapping] {`scheme.mapping`} Looks up *key* in the mapping *m*, and returns two values, *m* and the associated value. If *m* does not contain an entry with *key*, a thunk *make-value* is invoked, and creates a new mapping that contains all entries in *m* plus a new entry with *key* and the return value of *make-value*, then returns the new mapping and the return value of *make-value*.

Linear update version `mapping-intern!` may destructively modify *m* to produce the return value, while `mapping-intern` creates a new mapping.

```
(mapping-intern (mapping default-comparator 'a 1) 'b (^[] 2))
  ⇒
  mapping with a → 1, b → 2
  and
  2

(mapping-intern (mapping default-comparator 'a 1) 'a (^[] 2))
  ⇒
  mapping with a → 1
  and
  1
```

`mapping-update` *m key updater :optional failure success* [Function]
`mapping-update!` *m key updater :optional failure success* [Function]

[R7RS mapping] {`scheme.mapping`} Semantically equivalent to this:

```
(mapping-set m var
  (updater (mapping-ref m key failure success)))
```

The *failure* and *success* optional arguments are procedures with zero and one arguments, respectively. When omitted, *failure* defaults to a thunk that raises an error, and *success* defaults to `identity`.

First, *key* is looked up in *m*. If an entry is found, the associated value is passed to *success*; otherwise, *failure* is called with no arguments. Either way, let the returned value be *v0*.

Then, *v0* is passed to *updater*. Let its result be *v1*.

Finally, a mapping with the same entries as *m* except the value of *key* is altered to *v1* is returned.

Linear update version `mapping-update!` may destructively modify *m* to produce the return value, while `mapping-update` creates a new mapping.

```
(mapping-update (mapping default-comparator 'a 1)
                'a (pa$ + 1))
 ⇒ mapping with a → 2

(mapping-update (mapping default-comparator)
                'a (pa$ + 1) (^[] 0))
 ⇒ mapping with a → 1
```

`mapping-update/default` *m key updater default* [Function]
`mapping-update!/default` *m key updater default* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-pop` *m :optional failure* [Function]
`mapping-pop!` *m :optional failure* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-search` *m k failure success* [Function]
`mapping-search!` *m k failure success* [Function]
[R7RS mapping] {`scheme.mapping`}

The whole mapping

`mapping-size` *m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-find` *pred m failure* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-count` *pred m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-any?` *pred m* [Function]
`mapping-every?` *pred m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-keys` *m* [Function]
`mapping-values` *m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-entries` *m* [Function]
[R7RS mapping] {`scheme.mapping`}

Mapping and folding

`mapping-map` *proc comparator m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-map/monotone` *proc comparator m* [Function]
`mapping-map/monotone!` *proc comparator m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-for-each` *proc m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-fold` *kons knil m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-fold/reverse` *kons knil m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-map->list` *proc m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-filter` *pred m* [Function]
`mapping-filter!` *pred m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-remove` *pred m* [Function]
`mapping-remove!` *pred m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-partition` *pred m* [Function]
`mapping-partition!` *pred m* [Function]
[R7RS mapping] {`scheme.mapping`}

Copying and conversion

`mapping-copy` *m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping->alist` *m* [Function]
[R7RS mapping] {`scheme.mapping`}

`alist->mapping` *comparator alist* [Function]
[R7RS mapping] {`scheme.mapping`}

`alist->mapping!` *m alist* [Function]
[R7RS mapping] {`scheme.mapping`}

`alist->mapping/ordered` *comparator alist* [Function]
`alist->mapping/ordered!` *m alist* [Function]
[R7RS mapping] {`scheme.mapping`}

Submappings

`mapping=?` *comparator m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping<?` *comparator m1 m2 . . .* [Function]
`mapping<=?` *comparator m1 m2 . . .* [Function]
`mapping>?` *comparator m1 m2 . . .* [Function]
`mapping>=?` *comparator m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping`}

Set operations

`mapping-union` *m1 m2 . . .* [Function]
`mapping-union!` *m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-intersection` *m1 m2 . . .* [Function]
`mapping-intersection!` *m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-difference` *m1 m2 . . .* [Function]
`mapping-difference!` *m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-xor` *m1 m2 . . .* [Function]
`mapping-xor!` *m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping`}

Mappings with ordered keys

`mapping-min-key` *m* [Function]
`mapping-max-key` *m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-min-value` *m* [Function]
`mapping-max-value` *m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-min-entry` *m* [Function]
`mapping-max-entry` *m* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-key-predecessor` *m obj failure* [Function]
`mapping-key-successor` *m obj failure* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-range=` *m obj* [Function]
`mapping-range<` *m obj* [Function]
`mapping-range<=` *m obj* [Function]
`mapping-range>` *m obj* [Function]
`mapping-range>=` *m obj* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-range=!` *m obj* [Function]
`mapping-range<!` *m obj* [Function]
`mapping-range<=!` *m obj* [Function]
`mapping-range>!` *m obj* [Function]
`mapping-range>=!` *m obj* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-split` *m obj* [Function]
`mapping-split!` *m obj* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-catenate` *comparator m1 key value m2* [Function]
`mapping-catenate!` *m1 key value m2* [Function]
[R7RS mapping] {`scheme.mapping`}

Comparators

`make-mapping-comparator` *comparator* [Function]
[R7RS mapping] {`scheme.mapping`}

`mapping-comparator` [Variable]
[R7RS mapping] {`scheme.mapping`}

10.3.17.2 Hashmaps

Constructors

`hashmap` *comparator key value ...* [Function]
[R7RS mapping] {`scheme.mapping.hash`} Creates a new hashmap with the given *comparator*, whose initial content is provided by *key value* ....

The *comparator* argument must be a comparator (see Section 6.2.4 [Basic comparators], page 103).

The *key value* ... arguments must be even length, alternating keys and values.

```
(define m (hashmap default-comparator 'a 1 'b 2))

(hashmap-ref m 'a) ⇒ 1
(hashmap-ref m 'b) ⇒ 2
```

`hashmap-unfold` *p f g seed comparator* [Function]
[R7RS mapping] {`scheme.mapping.hash`} Creates a new hashmap, whose content is populated by three procedures, *p*, *f* and *g*, and a seed value *seed*, as follows.

In each iteration, we have a current seed value, whose initial value is *seed*.

First, *p*, a stop predicate, is applied to the current seed value. If it returns true, we stop iteration and returns the new hashmap.

Next, *f* is applied to the current seed value. It must return two values. The first one is for a key and the second one for the value. We add this pair to the hashmap.

Then, *g* is applied to the current seed value. The result becomes the seed value of the next iteration. And we iterate.

The following example creates a hashmap that maps ASCII characters to their character codes:

```
(hashmap-unfold (cut >= <> 128)
                (^c (values (integer->char c) c))
                (cut + <> 1)
                0
                default-comparator)
```

Predicates

`hashmap?` *obj* [Function]
[R7RS mapping] {`scheme.mapping.hash`} Returns `#t` iff *obj* is a hashmap object.

`hashmap-empty?` *m* [Function]
[R7RS mapping] {`scheme.mapping.hash`} *M* must be a hashmap. Returns `#t` if *m* is empty, `#f` otherwise. In Gauche, this is same as `tree-map-empty?` (see Section 6.16 [Treemaps], page 182).

hashmap-contains? *m key* [Function]

[R7RS mapping] {scheme.mapping.hash} *M* must be a hashmap. Returns #t if *m* has an entry with *key*, #f otherwise. In Gauche, this is same as tree-map-exists? (see Section 6.16 [Treemaps], page 182).

hashmap-disjoint? *m1 m2* [Function]

[R7RS mapping] {scheme.mapping.hash} Returns #t iff two hashmaps *m1* and *m2* have no keys in common. In other words, there's no such key K that satisfy both (hashmap-contains? m1 K) and (hashmap-contains? m2 K).

Accessors

hashmap-ref *m key :optional failure success* [Function]

[R7RS mapping] {scheme.mapping.hash} Get the value from a hashmap *m* associated with *key*, and calls *success* on the value, and returns its result. If *m* doesn't have *key*, *failure* is invoked with no arguments and its result is returned. Both *success* and *failure* is called in tail context.

When *failure* is omitted and *key* is not found, an error is signaled. When *success* is omitted, identity is assumed.

hashmap-ref/default *m key default* [Function]

[R7RS mapping] {scheme.mapping.hash} Returns the value associated to *key* from a hashmap *m*. If *m* doesn't have *key*, *default* is returned.

hashmap-key-comparator *m* [Function]

[R7RS mapping] {scheme.mapping.hash} Returns a comparator used to compare keys in a hashmap *m*. See Section 6.2.4 [Basic comparators], page 103, for the details of comparators.

Updaters

Note that the basic premise of hashmaps srfi is to treat hashmaps as immutable. Each updating operation comes with a purely functional version (without bang) and a linear update version (with bang), but the linear update version may not require to destructively modiy the passed hashmap; it's merely a hint that it may reuse the argument for the efficiency. You always need to use the returned hashmap as the result of update. If you use linear update versions, you shouldn't use the passed hashmap afterwards, for there's no guarantee how the state of the passed hashmap is.

hashmap-adjoin *m arg ...* [Function]
hashmap-adjoin! *m arg ...* [Function]

[R7RS mapping] {scheme.mapping.hash}

hashmap-set *m arg ...* [Function]
hashmap-set! *m arg ...* [Function]

[R7RS mapping] {scheme.mapping.hash}

hashmap-replace *m key value* [Function]
hashmap-replace! *m key value* [Function]

[R7RS mapping] {scheme.mapping.hash}

hashmap-delete *m key ...* [Function]
hashmap-delete! *m key ...* [Function]

[R7RS mapping] {scheme.mapping.hash}

hashmap-delete-all *m key-list* [Function]
hashmap-delete-all! *m key-list* [Function]

[R7RS mapping] {scheme.mapping.hash}

`hashmap-intern` *m key failure* [Function]
`hashmap-intern!` *m key failure* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-update` *m key updater :optional failure success* [Function]
`hashmap-update!` *m key updater :optional failure success* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-update/default` *m key updater default* [Function]
`hashmap-update!/default` *m key updater default* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-pop` *m :optional failure* [Function]
`hashmap-pop!` *m :optional failure* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-search` *m k failure success* [Function]
`hashmap-search!` *m k failure success* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

The whole hashmap

`hashmap-size` *m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-find` *pred m failure* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-count` *pred m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-any?` *pred m* [Function]
`hashmap-every?` *pred m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-keys` *m* [Function]
`hashmap-values` *m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-entries` *m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

Mapping and folding

`hashmap-map` *proc comparator m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-for-each` *proc m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-fold` *kons knil m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-map->list` *proc m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-filter` *pred m* [Function]
`hashmap-filter!` *pred m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-remove` *pred m* [Function]
`hashmap-remove!` *pred m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-partition` *pred m* [Function]
`hashmap-partition!` *pred m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

Copying and conversion

`hashmap-copy` *m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap->alist` *m* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`alist->hashmap` *comparator alist* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`alist->hashmap!` *m alist* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

Subhashmaps

`hashmap=?` *comparator m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap<?` *comparator m1 m2 . . .* [Function]
`hashmap<=?` *comparator m1 m2 . . .* [Function]
`hashmap>?` *comparator m1 m2 . . .* [Function]
`hashmap>=?` *comparator m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

Set operations

`hashmap-union` *m1 m2 . . .* [Function]
`hashmap-union!` *m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-intersection` *m1 m2 . . .* [Function]
`hashmap-intersection!` *m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-difference` *m1 m2 . . .* [Function]
`hashmap-difference!` *m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-xor` *m1 m2 . . .* [Function]
`hashmap-xor!` *m1 m2 . . .* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

Comparators

`make-hashmap-comparator` *comparator* [Function]
[R7RS mapping] {`scheme.mapping.hash`}

`hashmap-comparator` [Variable]
[R7RS mapping] {`scheme.mapping.hash`}

10.3.18 `scheme.division` - R7RS integer division

`scheme.division` [Module]
This module provides a comprehensive set of integer division operators.

Quotient and remainder in integer divisions can be defined in multiple ways, when you consider the choice of sign of the result with regard to the operands. Gauche has builtin procedures in several flavors: R5RS `quotient`, `remainder` and `modulo`, R6RS `div`, `mod`, `div0` and `mod0`, and R7RS `floor-quotient`, `floor-remainder`, `floor/`, `truncate-quotient`, `truncate-remainder`, `truncate/`.

This module complements R7RS procedures, by adding `ceiling`, `round`, `euclidean` and `balanced` variants.

The following procedures are in scheme.division but built-in in Gauche (see Section 6.3.4 [Arithmetics], page 112).

```
floor-quotient     floor-remainder    floor/
truncate-quotient  truncate-remainder truncate/
```

`ceiling-quotient` *n d* [Function]
`ceiling-remainder` *n d* [Function]
`ceiling/` *n d* [Function]
[R7RS division] {`scheme.division`}

```
ceiling-quotient = ceiling(n / d)
ceiling-remainder = n - d * ceiling-quotient
ceiling/ = values(ceiling-quotient, ceiling-remainder)
```

`round-quotient` *n d* [Function]
`round-remainder` *n d* [Function]
`round/` *n d* [Function]
[R7RS division] {`scheme.division`}

```
round-quotient = round(n/d)
round-remainder = n - d * round-quotient
round/ = values(round-quotient, round-remainder)
```

`euclidean-quotient` *n d* [Function]
`euclidean-remainder` *n d* [Function]
`euclidean/` *n d* [Function]
[R7RS division] {`scheme.division`}

```
euclidean-quotient = floor(n / d)   if d > 0
                     ceiling(n / d) if d < 0
euclidean-remainder = n - d * euclidean-quotient
euclidean/ = values(euclidean-quotient, euclidean-remainder)
```

The Eclidean variant satisfies a property `0 <= remainder < abs(d)`. These are the same as R6RS's `div`, `mod`, and `div-and-mod`, except that they accept non-integers (see Section 6.3.4 [Arithmetics], page 112)

`balanced-quotient` *n d* [Function]
`balanced-remainder` *n d* [Function]
`balanced/` *n d* [Function]

[R7RS division] {`scheme.division`}

```
balanced-quotient = roundup(n / d)
balanced-remainder = n - d * balanced-quotient
balanced/ = values(balanced-quotient, balanced-remainder)
  where roundup(x) is  ceiling(x) if x - floor(x) <= 0.5
                   and floor(x)   if x - floor(x) > 0.5
```

The balanced variant satisfies a property `-abs(d/2) <= remainder < abs(d/2)`. These are the same as R6RS's `div0`, `mod0`, and `div0-and-mod0`, except that they accept non-integers (see Section 6.3.4 [Arithmetics], page 112).

10.3.19 `scheme.bitwise` - R7RS bitwise operations

`scheme.bitwise` [Module]

This module provides comprehensive bitwise operations. Originally it was srfi-151. It is mostly a superset of srfi-60, with some change of names for the consistency and the compatibility (see Section 11.15 [Integers as bits], page 616). We keep srfi-60 for legacy code, while recommend this module to be used in the new code.

The following procedures are Gauche built-in. See Section 6.3.6 [Basic bitwise operations], page 121, for the description.

```
integer-length    copy-bit         bit-field
```

Basic operations

`bitwise-not` *n* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns the bitwise complement of *n*. Same as builtin `lognot` (see Section 6.3.6 [Basic bitwise operations], page 121).

`bitwise-and` *n* ... [Function]
`bitwise-ior` *n* ... [Function]
`bitwise-xor` *n* ... [Function]
`bitwise-eqv` *n* ... [Function]

[R7RS bitwise] {`scheme.bitwise`} When no arguments are given, these procedures returns `-1`, `0`, `0` and `-1`, respectively. With one arguments, they return the argument as is. With two arguments, they return bitwise and, ior, xor, and eqv (complement of xor). With three or more arguments, they apply binary operations associaively, that is,

```
(bitwise-xor a b c)
 ≡ (bitwise-xor a (bitwise-xor b c))
 ≡ (bitwise-xor (bitwise-xor a b) c)
```

Be careful that multi-argument `bitwise-eqv` does not produce bit 1 everywhere that all the argument's bit agree.

The first three procedures are the same as built-in `logand`, `logior` and `logxor`, respectively (see Section 6.3.6 [Basic bitwise operations], page 121).

`bitwise-nand` *n0 n1* [Function]
`bitwise-nor` *n0 n1* [Function]
`bitwise-andc1` *n0 n1* [Function]
`bitwise-andc2` *n0 n1* [Function]
`bitwise-orc1` *n0 n1* [Function]

`bitwise-orc2` *n0 n1* [Function]

[R7RS bitwise] {`scheme.bitwise`} These operations are not associative.

```
nand n0 n1    ≡  (NOT (AND n0 n1))
nor n0 n1     ≡  (NOT (OR n0 n1))
andc1 n0 n1   ≡  (AND (NOT n0) n1)
andc2 n0 n1   ≡  (AND n0 (NOT n1))
orc1 n0 n1    ≡  (OR (NOT n0) n1)
orc2 n0 n1    ≡  (OR n0 (NOT n1))
```

Integer operations

`arithmetic-shift` *n count* [Function]

[R7RS bitwise] {`scheme.bitwise`} Shift *n* for *count* bits to left; if *count* is negative, it shifts *n* to right for *-count* bits.

Same as builtin `ash` (see Section 6.3.6 [Basic bitwise operations], page 121).

`bit-count` *n* [Function]

[R7RS bitwise] {`scheme.bitwise`} If *n* is positive, returns the number of `1`'s in *n*. If *n* is negative, returns the number of `0`'s in *n*.

Same as builtin `logcount` (see Section 6.3.6 [Basic bitwise operations], page 121).

`bitwise-if` *mask n0 n1* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns integer, whose *n*-th bit is taken as follows: If the *n*-th bit of *mask* is 1, the *n*-th bit of *n0*; otherwise, the *n*-th bit of *n1*.

```
(bitwise-if #b10101100 #b00110101 #b11001010)
 ⇒ #b01100110
```

Single-bit operations

`bit-set?` *index n* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns `#t` or `#f` if *index*-th bit (counted from LSB) of *n* is `1` or `0`, respectively.

Same as built-in `logbit?` (see Section 6.3.6 [Basic bitwise operations], page 121).

`bit-swap` *index1 index2 n* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns an integer with *index1*-th bit and *index2*-th bit are swapped. Index is counted from LSB.

`any-bit-set?` *mask n* [Function]
`every-bit-set?` *mask n* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns `#t` iff any/all bits set in *mask* are also set in *n*.

`any-bit-set?` is the same as built-in `logtest`, except `logtest` accepts one or more arguments (see Section 6.3.6 [Basic bitwise operations], page 121).

`first-set-bit` *n* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns the number of factors of two of integer *n*; that is, returns a maximum `k` such that `(expt 2 k)` divides *n* without a remainder. It is the same as the index of the least significant `1` in *n*, hence the alias `first-set-bit`.

```
(first-set-bit 0) ⇒ -1    ; edge case
(first-set-bit 1) ⇒ 0
(first-set-bit 2) ⇒ 1
(first-set-bit 15) ⇒ 0
(first-set-bit 16) ⇒ 4
```

This is equivalent to Gauche's built-in `twos-exponent-factor` (see Section 6.3.6 [Basic bitwise operations], page 121).

Bit field operations

`bit-field-any?` *n start end* [Function]
`bit-field-every?` *n start end* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns `#t` iff any/all bits of *n* from *start* (inclusive) to *end* (exclusive) are set.

`bit-field-clear` *n start end* [Function]
`bit-field-set` *n start end* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns *n* with the bits from *start* (inclusive) to *end* (exclusive) are set to all `0`'s/`1`'s.

`bit-field-replace` *dst src start end* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns *dst* with the bitfield from *start* to *end* are replaced with the least-significant (*end*-*start*) bits of *src*.

```
(bit-field-replace #b101010 #b010 1 4) ⇒ #b100100
```

Same as built-in `copy-bit-field` (see Section 6.3.6 [Basic bitwise operations], page 121).

`bit-field-replace-same` *dst src start end* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns *dst* with the bitfield from *start* to *end* are replaced with the `src`'s bitfield from *start* to *end*.

```
(bit-field-replace-same #b111111 #b100100 1 4) ⇒ #b110101
```

`bit-field-rotate` *n count start end* [Function]

[R7RS bitwise] {`scheme.bitwise`} Rotate the region of *n* between *start*-th bit (inclusive) and *end*-th bit (exclusive) by *count* bits to the left. If *count* is negative, it rotates to the right by -*count* bits.

```
(bit-field-rotate #b110100100010000 -1 5 9)
 ⇒ 26768 ;#b110100010010000

(bit-field-rotate #b110100100010000 1 5 9)
 ⇒ 26672 ;#b110100000110000
```

`bit-field-reverse` *n start end* [Function]

[R7RS bitwise] {`scheme.bitwise`} Reverse the order of bits of *n* between *start*-th bit (inclusive) and *end*-th bit (exclusive).

```
(bit-field-reverse #b10100111 0 8)
 ⇒ 229 ; #b11100101
```

Bits conversion

`bits->list` *n :optional len* [Function]
`bits->vector` *n :optional len* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns a list/vector of booleans of length *len*, corresponding to each bit in non-negative integer *n*, LSB-first. When *len* is omitted, `(integer-length n)` is used.

```
(bits->vector #b101101110)
  ⇒ #(#f #t #t #t #f #t #t #f #t)
```

Note: Srfi-60 has a similar `integer->list`, but the order of bits is reversed.

`list->bits` *bool-list* [Function]
`vector->bits` *bool-vector* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns an exact integer formed from boolean values in given list/vector, LSB first. The result will never be negative.

```
(list->bits '(#f #t #t #t #f #t #t #f #t))
 ⇒ #b101101110
```

Note: Srfi-60 has a similar `list->integer`, but the order of bits is reversed.

`bits` *bool . . .* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns the integer coded by *bool*s, LSB first. The result will never be negative.

```
(bits #f #t #t #t #f #t #t #f #t)
 ⇒ #b101101110
```

Note: Srfi-60 has a similar `booleans->integer`, but the order of bits is reversed.

Fold, unfold and generate

`bitwise-fold` *kons knil n* [Function]

[R7RS bitwise] {`scheme.bitwise`} Traverse bits in integer *n* from LSB to the `(integer-length n)` bit, applying `kons` on the bit as boolean and the seed value, whose initial value is given by *knil*. Returns the last result of *kons*.

```
(bitwise-fold cons '() #b10110111)
 ⇒ (#t #f #t #t #f #t #t #t)
```

`bitwise-for-each` *proc n* [Function]

[R7RS bitwise] {`scheme.bitwise`} Applies *proc* to the bit as boolean in *n*, from LSB to the `(integer-length n)` bit. The result is discarded.

`bitwise-unfold` *p f g seed* [Function]

[R7RS bitwise] {`scheme.bitwise`} Generates a non-negative integer bit by bit, from LSB to MSB. The *seed* gives the initial state value. For each iteration, *p* is applied to the current state value, and if it returns a true value, the iteration ends and `bitwise-unfold` returns the accumulated bits as an integer. Otherwise, *f* is applied to the current state value, and its result, coerced to a boolean value, determines the bit value. Then *g* is applied to the current state value to produce the next state value of the next iteration.

The following expression produces a bitfield of width 100, where n-th bit indicates whether n is prime or not:

```
(use math.prime)
(bitwise-unfold (cut = 100 <>)
                small-prime?
                (cut + 1 <>)
                0)
```

`make-bitwise-generator` *n* [Function]

[R7RS bitwise] {`scheme.bitwise`} Returns a generator that generates boolean values corresponding to the bits in *n*, LSB-first. The returned generator is infinite.

This is similar to `bits->generator` in `gauche.generator`, except that the generator created by it stops at the integer length of *n* (see Section 9.11 [Generators], page 372).

10.3.20 scheme.fixnum - R7RS fixnums

scheme.fixnum [Module]

This module provides a set of fixnum-specific operations. Originally defined as srfi-143.

A fixnum is a small exact integer that can be handled very efficiently. In Gauche, fixnum is 62bit wide on 64bit platforms, and 30bit wide on 32bit platforms.

Note that these procedures are defined only to work on fixnums, but it is not enforced. If you pass non-fixnum arguments, or the result falls out of range of fixnums, what happens is up to the implementation. Consider these procedures as the way to tell your intentions to the compiler for potential optimizations.

In the current Gauche architecture, generic numeric operators are just as efficient, so most procedures provided in this module are aliases to corresponding operators. However, we might employ some optimizations in future versions.

The procedure `fixnum?` is built-in, and not explained here. See Section 6.3.2 [Numerical predicates], page 109.

fx-width [Variable]

[R7RS fixnum] {`scheme.fixnum`} A variable bound to an exact positive integer *w*, where *w* is the greatest number such that exact integers between `2^(w-1) - 1` and `-2^(w-1)` are all fixnums. This value is the same as the built-in procedure `fixnum-width` returns (see Section 6.3.4 [Arithmetics], page 112).

In Gauche, it is usually `30` for 32bit platforms, and `62` for 64bit platforms.

fx-greatest [Variable]
fx-least [Variable]

[R7RS fixnum] {`scheme.fixnum`} Variables bound to the greatest fixnum and the least fixnum. They are the same as the built-in procedures `greatest-fixnum` and `least-fixnum` return, respectively (see Section 6.3.4 [Arithmetics], page 112).

The following table shows the typical values on Gauche:

Platform	fx-greatest	fx-least
32bit	536,870,911	-536,870,912
64bit	2,305,843,009,213,693,951	-2,305,843,009,213,693,952

fx=? *i . . .* [Function]
fx<? *i . . .* [Function]
fx<=? *i . . .* [Function]
fx>? *i . . .* [Function]
fx>=? *i . . .* [Function]

[R7RS fixnum] {`scheme.fixnum`} These are equivalent to built-in `=`, `<`, `<=`, `>` and `>=`, except that you should use these only for fixnums.

fxzero? *i* [Function]
fxpositive? *i* [Function]
fxnegative? *i* [Function]
fxodd? *i* [Function]
fxeven? *i* [Function]

[R7RS fixnum] {`scheme.fixnum`} These are equivalent to built-in `zero?`, `positive?`, `negative?`, `odd?` and `even?`, except that you should use these only for fixnums.

fxmax *i j . . .* [Function]
fxmin *i j . . .* [Function]

[R7RS fixnum] {`scheme.fixnum`} These are equivalent to built-in `max` and `min`, except that you should use these only for fixnums.

fx+ *i j* [Function]
fx- *i j* [Function]
fx* *i j* [Function]

[R7RS fixnum] {scheme.fixnum} These are equivalent to built-in +, - and *, except that these take exactly two arguments, and you should use these only for fixnums and when the result fits within fixnum range.

fxneg *i* [Function]

[R7RS fixnum] {scheme.fixnum} This is equivalent to single-argument -, except that you should use this only for fixnums and when the result fits within fixnum range.

fxquotient *i j* [Function]
fxremainder *i j* [Function]
fxabs *i* [Function]
fxsquare *i* [Function]

[R7RS fixnum] {scheme.fixnum} These are equivalent to built-in quotient, remainder, abs and square, except that you should use these only for fixnums and when the result fits within fixnum range.

fxsqrt *i* [Function]

[R7RS fixnum] {scheme.fixnum} This is equivalent to exact-integer-sqrt (not sqrt), except that you should use it only for fixnums. See Section 6.3.4 [Arithmetics], page 112.

fx+/carry *i j k* [Function]
fx-/carry *i j k* [Function]
fx*/carry *i j k* [Function]

[R7RS fixnum] {scheme.fixnum} These calculates (+ i j k), (- i j k) and (+ (* i j) k), respectively, then split the result to the remainder value R in the fixnum range, and spilled value Q, and return those values. That is, (+ (* Q (expt 2 fx-width)) R) is the result of above calculations. Both Q and R fits in the fixnum range, and - 2^(w-1) <= R < 2^(w-1), where w is fx-width.

```
(fx*/carry 1845917459 19475917581 4735374)
 ⇒ -942551854601421179 and 8

(+ (* 8 (expt 2 fx-width)) -942551854601421179)
 ⇒ 35950936292817682053

(+ (* 1845917459 19475917581) 4735374)
 ⇒ 35950936292817682053
```

These are primitives to implement extended-precision integer arithmetic on top of fixnum operations. In Gauche, however, you can just use built-in bignums. We provide these for the compatibility.

fxnot *i* [Function]
fxand *i ...* [Function]
fxior *i ...* [Function]
fxxor *i ...* [Function]
fxarithmetic-shift *i count* [Function]
fxlength *i* [Function]
fxbit-count *i* [Function]
fxcopy-bit *index i boolean* [Function]
fxbit-set? *index i* [Function]
fxbit-field *i start end* [Function]

`fxfirst-set-bit` *i* [Function]

[R7RS fixnum] {`scheme.fixnum`} These are equivalent to `lognot`, `logand`, `logior`, `logxor`, `ash`, `integer-length`, `logcount`, `copy-bit`, `logbit?`, `bit-field`, and `twos-exponent-factor` respectively, except that you should use these only for fixnums. See Section 6.3.6 [Basic bitwise operations], page 121.

`fxif` *mask i j* [Function]
`fxbit-field-rotate` *i start end* [Function]
`fxbit-field-rotate` *i start end* [Function]

[R7RS fixnum] {`scheme.fixnum`} These are equivalent to srfi-60's `bitwise-if`, `rotate-bit-field` and `reverse-bit-field`, except that you should use these only for fixnums. See Section 11.15 [Integers as bits], page 616.

10.3.21 `scheme.flonum` - R7RS flonum

`scheme.flonum` [Module]

This module provides a set of flonum-specific operations. Originally defined as srfi-144.

In Gauche, a flonum is IEEE 754 double-precision floating point numbers.

Note that these procedures are defined only to work on flonums, but it is not enforced. If you pass non-flonum arguments, the result is undefined. Consider these procedures as the way to tell your intentions to the compiler for potential optimizations.

In the current Gauche architecture, generic numeric operators are just as efficient, so most procedures provided in this module are aliases to corresponding operators. However, we might employ some optimizations in future versions.

The procedure `flonum?` is built-in, and not explained here. See Section 6.3.2 [Numerical predicates], page 109.

Constants

We also have a few constants in `math.const` module (see Section 12.25 [Mathematic constants], page 704).

`fl-e` [Constant]

[R7RS flonum] {`scheme.flonum`} e.

`fl-1/e` [Constant]

[R7RS flonum] {`scheme.flonum`} `(/ e)`.

`fl-e-2` [Constant]

[R7RS flonum] {`scheme.flonum`} `(square e)`.

`fl-e-pi/4` [Constant]

[R7RS flonum] {`scheme.flonum`} `(expt e (/ pi 4))`.

`fl-log2-e` [Constant]

[R7RS flonum] {`scheme.flonum`} `(log2 e)`. (Same as `(/ (log 2))`).

`fl-log10-e` [Constant]

[R7RS flonum] {`scheme.flonum`} `(log10 e)`. (Same as `(/ (log 10))`).

`fl-log-2` [Constant]

[R7RS flonum] {`scheme.flonum`} `(log 2)`.

`fl-1/log-2` [Constant]

[R7RS flonum] {`scheme.flonum`} `(/ (log 2))`. (Same as `(log2 e)`).

`fl-log-3` [Constant]
[R7RS flonum] {scheme.flonum} (log 3).

`fl-log-pi` [Constant]
[R7RS flonum] {scheme.flonum} (log pi).

`fl-log-10` [Constant]
[R7RS flonum] {scheme.flonum} (log 10).

`fl-1/log-10` [Constant]
[R7RS flonum] {scheme.flonum} (/ (log 10)). (Same as (log10 e)).

`fl-pi` [Constant]
[R7RS flonum] {scheme.flonum} pi.

`fl-1/pi` [Constant]
[R7RS flonum] {scheme.flonum} (/ pi).

`fl-2pi` [Constant]
[R7RS flonum] {scheme.flonum} (* 2 pi).

`fl-pi/2` [Constant]
[R7RS flonum] {scheme.flonum} (/ pi 2).

`fl-pi/4` [Constant]
[R7RS flonum] {scheme.flonum} (/ pi 4).

`fl-pi-squared` [Constant]
[R7RS flonum] {scheme.flonum} (square pi).

`fl-degree` [Constant]
[R7RS flonum] {scheme.flonum} (/ pi 180).

`fl-2/pi` [Constant]
[R7RS flonum] {scheme.flonum} (/ 2 pi).

`fl-2/sqrt-pi` [Constant]
[R7RS flonum] {scheme.flonum} (/ 2 (sqrt pi)).

`fl-sqrt-2` [Constant]
(sqrt 2).

`fl-sqrt-3` [Constant]
[R7RS flonum] {scheme.flonum} (sqrt 3).

`fl-sqrt-5` [Constant]
[R7RS flonum] {scheme.flonum} (sqrt 5).

`fl-sqrt-10` [Constant]
[R7RS flonum] {scheme.flonum} (sqrt 10).

`fl-1/sqrt-2` [Constant]
[R7RS flonum] {scheme.flonum} (/ (sqrt 2)).

`fl-cbrt-2` [Constant]
[R7RS flonum] {scheme.flonum} (expt 2 1/3).

`fl-cbrt-3` [Constant]
[R7RS flonum] {scheme.flonum} (expt 3 1/3).

`fl-4thrt-2` [Constant]
[R7RS flonum] {`scheme.flonum`} `(expt 2 1/4)`.

`fl-phi` [Constant]
[R7RS flonum] {`scheme.flonum`} `(/ (+ 1 (sqrt 5)) 2)`.

`fl-log-phi` [Constant]
[R7RS flonum] {`scheme.flonum`} `(log fl-phi)`.

`fl-1/log-phi` [Constant]
[R7RS flonum] {`scheme.flonum`} `(/ (log fl-phi))`.

`fl-euler` [Constant]
[R7RS flonum] {`scheme.flonum`} Euler's constant.

`fl-e-euler` [Constant]
[R7RS flonum] {`scheme.flonum`} `(exp fl-euler)`

`fl-sin-1` [Constant]
[R7RS flonum] {`scheme.flonum`} `(sin 1)`

`fl-cos-1` [Constant]
[R7RS flonum] {`scheme.flonum`} `(cos 1)`

`fl-gamma-1/2` [Constant]
[R7RS flonum] {`scheme.flonum`} `(gamma 1/2)`

`fl-gamma-1/3` [Constant]
[R7RS flonum] {`scheme.flonum`} `(gamma 1/3)`

`fl-gamma-2/3` [Constant]
[R7RS flonum] {`scheme.flonum`} `(gamma 2/3)`

`fl-gamma-1/3` [Constant]
[R7RS flonum] {`scheme.flonum`} `(gamma 1/3)`

`fl-greatest` [Constant]
`fl-least` [Constant]
[R7RS flonum] {`scheme.flonum`} Bound to the largest/smallest positive finite flonum.

The latter is the same as `(flonum-min-denormalized)` in Gauche (see Section 6.3.3 [Numerical comparison], page 111).

`fl-epsilon` [Constant]
[R7RS flonum] {`scheme.flonum`} The same as `(flonum-epsilon)` in Gauche (see Section 6.3.3 [Numerical comparison], page 111).

`fl-fast-fl+*` [Constant]
[R7RS flonum] {`scheme.flonum`} If this is `#t`, `(fl+* x y z)` executes as fast as, or faster than, `(fl+ (fl* x y) z)`. If not, `#f`.

`fl-integer-exponent-zero` [Constant]
`fl-integer-exponent-nan` [Constant]
[R7RS flonum] {`scheme.flonum`} These are exact integer values returned in special occasion from `flinteger-exponent`. The values themselves don't mean much; they're to be compared with the result of `flinteger-exponent`.

If its argument is 0, `flinteger-exponent` returns `fl-integer-exponent-zero`. If its argument is `+nan.0`, `flinteger-exponent` returns `fl-integer-exponent-nan`.

Constructors

`flonum` *number* [Function]

[R7RS flonum] {`scheme.flonum`} Returns a flonum that's equal to or (if there's no equivalent flnoum) the closest to *number*.

If *number* is not a real number, `+nan.0` is returned. (Note: srfi-144 recommends it, but a conformant implementation may signal an error in such case. Portable code shouldn't rely on this behavior.)

`fladjacent` *x y* [Function]

[R7RS flonum] {`scheme.flonum`} Returns a flonum adjacent to *x* in the direction of *y*. If *x* = *y*, *x* is returned.

```
(fladjacent 1.0 2.0) ⇒ 1.0000000000000002
(fladjacent 1.0 0.0) ⇒ 0.9999999999999999
```

`flcopysign` *x y* [Function]

[R7RS flonum] {`scheme.flonum`} Returns a flonum whose absolute value is `(abs x)` and whose sign is the sign of *y*.

`make-flonum` *x n* [Function]

[R7RS flonum] {`scheme.flonum`} Returns a flonum `(* x (expt 2 n))`. It is the same as `ldexp` (see Section 6.3.5 [Numerical conversions], page 118).

Accessors

`flinteger-fraction` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns two flnoums, the integral part of *x* and the fractional part of *x*. Same as `modf` (see Section 6.3.5 [Numerical conversions], page 118).

If *x* is `+inf.0`, `+inf.0` and `0.0` is returned. If *x* is `-inf.0`, `-inf.0` and `-0.0` is returned. If *x* is `+nan.0`, `+nan.0` and `+nan.0` is returned. (These corner cases are not explicit in srfi-144, but follows the POSIX specification of `modf`.)

```
(flinteger-fraction fl-pi)
 ⇒ 3.0 and 0.14159265358979312
```

`flexponent` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns the exponent part of *x* as a flnoum. If *x* is a non-zero finite value, the result is an integer.

If *x* is zero, `-inf.0` is returned. If *x* is *+inf.0* or `-inf.0`, `+inf.0` is returned.

```
(flexponent 1.0)      ⇒ 0.0
(flexponent 1024.0) ⇒ 10.0
(flexponent 0.01)    ⇒ -7.0
(flexponent fl-least) ⇒ -1074.0
```

`flinteger-exponent` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns the same as `flexponent` but as an exact integer.

If *x* is zero, the value of `fl-integer-exponent-zero` is returned. If *x* is *+inf.0* or `-inf.0`, a large implementation-dependent exact integer (usually it's so large that `(ldexp 1 (flinteger-exponent +inf.0))` becomes `+inf.0`) is returned. If *x* is `+nan.0`, the value of `fl-integer-exponent-nan` is returned.

`flnormalized-fraction-exponent` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns two values, a normalized mantissa of *x* with the same sign as *x* as a flonum, and an exact integer exponent of *x*. If it returns a value *y* and *n*, `x = (* y (expt 2 n))` and `x = (ldexp y n)`. This is the same as `frexp` (see Section 6.3.5 [Numerical conversions], page 118).

If *x* is non-zero finite value, the first value falls between 0.5 (inclusive) and 1.0 (exclusive). The corner cases are not explicit in srfi-144, but Gauche follows `frexp`: If *x* is (minus) zero, it returns (minus) zero and 0; if *x* is infinity, it returns infinity (of the same sign) and 0; if *x* is `+nan.0`, it returns `+nan.0` and 0.

```
(flnormalized-fraction-exponent 12345.6789)
  ⇒ 0.7535204406738282 and 14
(make-flonum 0.7535204406738282 14)
  ⇒ 12345.6789
```

`flsign-bit` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns 0 if *x* is positive or *0.0*, and 1 if it is negative (including `-0.0`). `(flsign-bit +nan.0)` is implementation-dependent.

There's also `flsgn`, if you need sign instead of a bit value.

Predicates

(Note: `flonum?` is built-in; see Section 6.3.2 [Numerical predicates], page 109).

`fl=?` *x y z . . .* [Function]
`fl<?` *x y z . . .* [Function]
`fl>?` *x y z . . .* [Function]
`fl<=?` *x y z . . .* [Function]
`fl>=?` *x y z . . .* [Function]

[R7RS flonum] {`scheme.flonum`} Flonum specific version of numerical comparison predicates `=`, `<`, `>`, `<=` and `>=`, respectively.

Currently these are just an alias of the generic numerical comparison predicates; hence they don't reject when you pass non-flonum arguments, but doing so is not portable. These are to give compilers hints that you are passing flonums so that it can optimize.

Note that `-0.0` and `0.0` are the same in terms of these predicates, e.g. `(fl<? -0.0 0.0)` is `#f`.

If a `+nan.0` is passed to any of the arguments, these predicates returns `#f`.

`flunordered?` *x y* [Function]

[R7RS flonum] {`scheme.flonum`} Returns `#t` iff at least one of *x* or *y* is a `+nan.0`.

`flinteger?` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns `#t` if *x* is an integral flonum, `#f` otherwise.

`flzero?` *x* [Function]
`flpositive?` *x* [Function]
`flnevative?` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Flonum-specific version of `zero?`, `positive?` and `negative?`. Note that `(flnegative? -0.0)` is `#f`. You need `flsign-bit` or `flsgn` to distinguish negative zero from zero.

`flodd?` *x* [Function]
`fleven?` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Flonum-specific version of `odd?` and `even?`. An error is thrown if *x* is not an integer.

flfinite? *x* [Function]
flinfinite? *x* [Function]
flnan? *x* [Function]

[R7RS flonum] {scheme.flonum} Flonum-specific version of finite?, infinite? and nan? (see Section 6.3.2 [Numerical predicates], page 109).

flnormalized? *x* [Function]
fldenormalized? *x* [Function]

[R7RS flonum] {scheme.flonum} Returns #t iff *x* is a normalized/denormalized flonum, respectively.

Arithmetic

flmax *x ...* [Function]
flmin *x ...* [Function]

[R7RS flonum] {scheme.flonum} Flonum-specific version of min and max, except that these can take no arguments, in which case -inf.0 and +inf.0 are returned, respectively.

Note that we don't check whether the arguments are flonums or not, but passing non-flonums are not portable.

fl+ *x ...* [Function]
fl* *x ...* [Function]

[R7RS flonum] {scheme.flonum} Flonum-specific version of +. and *. (see Section 6.3.4 [Arithmetics], page 112).

Note that we don't check whether the arguments are flonums or not, but passing non-flonums is not portable.

fl+* *x y z* [Function]

[R7RS flonum] {scheme.flonum} Returns (+ (* x y) z), but (potentially) faster, and more accurately. It calls C99 fma function internally. "More accurately" means that calculation is done in a single step, with rounding once at the end.

Note: As of release of 0.9.8, MinGW implementation of fma seems to produce slightly off value occasionally, as if it is calculated with the two steps (rounding after multiplication).

fl- *x y ...* [Function]
fl- *x y ...* [Function]

[R7RS flonum] {scheme.flonum} Flonum-specific version of -. and /. (see Section 6.3.4 [Arithmetics], page 112).

Note that we don't check whether the arguments are flonums or not, but passing non-flonums is not portable.

flabs *x* [Function]

[R7RS flonum] {scheme.flonum} Flonum-specific version of abs (see Section 6.3.4 [Arithmetics], page 112).

Note that we don't check whether the argument is a flonum or not, but passing non-flonum is not portable.

flabsdiff *x y* [Function]

[R7RS flonum] {scheme.flonum} Returns (abs (- x y)).

Note that we don't check whether the arguments are flonums or not, but passing non-flonum is not portable.

flposdiff [Function]

[R7RS flonum] {scheme.flonum} Returns (max (- x y) 0).

Note that we don't check whether the arguments are flonums or not, but passing non-flonum is not portable.

flsgn *x* [Function]

[R7RS flonum] {scheme.flonum} Returns 1.0 if *x*'s sign bit is 0 (zero or positive), -1.0 if it's 1 (negative). Same as (flcopysign 1.0 x), or (if (zero? (flsign-bit x)) 1.0 -1.0). Note that (flsgn 0.0) is 1.0, while (flsgn -0.0) is -1.0. The result of passing +nan.0 is implementation-dependent, reflecting the sign bit of underlying representation; it returns either 1.0 or -1.0 but you can't count on which.

To extract the sign bit, instead of obtaining a signed flonum, you can use flsign-bit.

flnumerator *x* [Function]
fldenominator *x* [Function]

[R7RS flonum] {scheme.flonum} Flonum-specific version of numerator and denominator (see Section 6.3.4 [Arithmetics], page 112).

For infinity and zero, denominator is 1.0.

Note that we don't check whether the arguments are flonums or not, but passing non-flonum is not portable.

flfloor *x* [Function]
flceiling *x* [Function]
flround *x* [Function]
fltruncate *x* [Function]

[R7RS flonum] {scheme.flonum} Flonum-specific version of floor, ceiling, round and truncate (see Section 6.3.4 [Arithmetics], page 112).

Note that we don't check whether the arguments are flonums or not, but passing non-flonum is not portable.

Exponents and logarithms

flexp *x* [Function]

[R7RS flonum] {scheme.flonum} Flonum-specific version of exp (see Section 6.3.4 [Arithmetics], page 112). Returns (expt fl-e x).

flexp2 *x* [Function]

[R7RS flonum] {scheme.flonum} Returns (expt 2 x)

flexp-1 *x* [Function]

[R7RS flonum] {scheme.flonum} Returns (- 1 (expt fl-e x)), but is much more accurate when *x* is small. We all C's expm1 internally.

flsquare *x* [Function]

[R7RS flonum] {scheme.flonum} Returns (*. x x) (see Section 6.3.4 [Arithmetics], page 112)

flsqrt *x* [Function]

[R7RS flonum] {scheme.flonum} Flonum-specific version of sqrt.

flcbrt *x* [Function]

[R7RS flonum] {scheme.flonum} Returns cubic root of a flonum *x*.

flhypot *x y* [Function]

[R7RS flonum] {scheme.flonum} Calculates (sqrt (* x x) (* y y)), with avoiding overflow or underflow during the intermediate steps.

`flexpt` *x y* [Function]
[R7RS flonum] {`scheme.flonum`} Flonum-specific version of `expt` (see Section 6.3.4 [Arithmetics], page 112).

`fllog` *x* [Function]
[R7RS flonum] {`scheme.flonum`} Flonum-specific version of `log` (see Section 6.3.4 [Arithmetics], page 112).

`fllog1+` *x* [Function]
[R7RS flonum] {`scheme.flonum`} Returns `(log (+ x 1))`, but is more accurate than `log` when `x` is near zero.

`fllog2` *x* [Function]
`fllog10` *x* [Function]
[R7RS flonum] {`scheme.flonum`} Returns base-2 and base-10 logarithm of a flonum *x*, respectively.

`make-fllog-base` *x* [Function]
[R7RS flonum] {`scheme.flonum`} Returns a procedure that calculates base-*x* logarithm. An error is signalled if *x* isn't a rela number greater than 1.0.

Trigonometric functions

`flsin` *x* [Function]
`flcos` *x* [Function]
`fltan` *x* [Function]
`flasin` *x* [Function]
`flacos` *x* [Function]
`flatan` *x* [Function]
`flatan` *y x* [Function]
`flsinh` *x* [Function]
`flcosh` *x* [Function]
`fltanh` *x* [Function]
`flasinh` *x* [Function]
`flacosh` *x* [Function]
`flatanh` *x* [Function]
[R7RS flonum] {`scheme.flonum`} Flonum-specific version of `sin`, `cos`, `tan`, `asin`, `acos`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, and `atanh`, respectively (see Section 6.3.4 [Arithmetics], page 112).

Note that `flatan` can take one or two arguments; if two arguments are passed, it calculates `(atan (/ y x)`.

Integer division

`flquotient` *x y* [Function]
[R7RS flonum] {`scheme.flonum`} Returns `(fltruncate (fl/ x y))`. The result is always integer flonum, but *x* and *y* doesn't need to be integers.

```
(flquotient 14.0 4.0)  ⇒ 3.0
(flquotient -14.0 4.0) ⇒ -3.0
(flquotient 14.0 2.5)  ⇒ 5.0
(flquotient -14.2 2.8) ⇒ -5.0
```

`flremainder` *x y* [Function]
[R7RS flonum] {`scheme.flonum`} Returns `(- x (* y (flquotient x y)))`.

```
(flquotient 14.0 4.0)  ⇒ 2.0
```

```
(flquotient -14.0 4.0) ⇒ -2.0
(flquotient 14.0 2.5)  ⇒ 1.5
(flquotient -14.2 2.8) ⇒ -0.1999999999999993 ; inexact calculation
```

`flremquo` *x y* [Function]

[R7RS flonum] {`scheme.flonum`} Returns two values:

- Remainder of `x/y`. (Same as `(flremainder x y)`, but we calculate it in different routines so the result may differ in a few ulp.
- Integer quotient of x/y, modulo 2^*n* (*n* >= 3), as an exact integer. Its sign is the same as x/y.

This corresponds to C99's *remquo*.

This function is useful to reduce the input for the periodic functions with symmetries.

Special functions

`flgamma` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Computes the value of gamma function for *x*. When *x* is integer, it is the same as the factorial of *x*-1.

This is same as Gauche's built-in `gamma` (see Section 6.3.4 [Arithmetics], page 112).

`flloggamma` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns two values, `(log (abs (flgamma x)))` and the sign of `(flgamma x)` as 1.0 if it is positive and -1.0 if it is negative.

The first value is calculated by Gauche's built-in `lgamma` (see Section 6.3.4 [Arithmetics], page 112). It's more accurate than using `gamma` then calling `log`. The second value is `+nan.0` when *x* is `-inf.0` or `+nan.0`.

`flfirst-bessel` *n x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns the *n*-th order Bessel function of the first kind.

`flsecond-bessel` *n x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns the *n*-th order Bessel function of the second kind.

`flerf` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns the error function erf(*x*).

`flerfc` *x* [Function]

[R7RS flonum] {`scheme.flonum`} Returns the complementary error function, 1 - erf(*x*).

11 Library modules - SRFIs

This chapter lists modules that provides SRFI functionalities. Note that some of SRFI features are built in Gauche core and not listed here. See Section 2.1 [Standard conformance], page 5, for entire list of supported SRFIs.

(Even if a srfi is not listed here, you can still say `(use srfi-N)` or `(import (srfi N))`, as far as srfi N is supported by Gauche.)

11.1 `srfi-1` - List library

`srfi-1` [Module]

SRFI-1 has become a part of R7RS large. See Section 10.3.1 [R7RS lists], page 512.

11.2 `srfi-4` - Homogeneous vectors

`srfi-4` [Module]

SRFI-4 is now implemented in `gauche.uvector` module See Section 9.36 [Uniform vectors], page 476. This module simply inherits `gauche.uvector` for backward-compatibility.

11.3 `srfi-5` - A compatible let form with signatures and rest arguments

`srfi-5` [Module]

This module provides srfi-5's extended `let` syntax.

`let` ((*var val*) . . . [. (*rest val* . . .)]) *body* . . . [Macro]
`let` *name* ((*var val*) . . . [. (*rest val* . . .)]) *body* . . . [Macro]
`let` (*name* (*var val*) . . . [. (*rest val* . . .)]) *body* . . . [Macro]

[SRFI-5] {`srfi-5`} The `let` syntax is extended in two ways.

- The extended `let` syntax accepts the *name* identifier (for named let syntax) within the list of bindings (as in the third syntax above).
- The extended `let` syntax accepts the rest parameter binding which works like the rest parameter in the `lambda` syntax.

See SRFI-5 document for rationale of this extension.

11.4 `srfi-7` - Feature-based program configuration language

`srfi-7` [Module]

This module provides a program configuration metalanguage (`program` form) defined in srfi-7. Gauche autoloads srfi-7 module, so you don't need to say `(use srfi-7)` explicitly. Note that the `program` form isn't necessary to be a Scheme expression. Srfi-7 allows an implementation to preprocess the `program` form to produce a Scheme program, then executes it with different means. Gauche implements `program` form as a macro, so it can evaluates the form directly. Nonetheless, it doesn't make sense to mix `program` form and other forms in one file, or expecting a return value of `program` form. A typical usage of `program` form is to prepare a single file which just contains `program` form. (It can load other files using `files` clause (see below) within the `program` form.) To execute such a program file in Gauche, you can just load it.

`program` *program-clause program-clause2 . . .* [Configuration Language]

[SRFI-7] {`srfi-7`} This is a configuration language to structure a Scheme program, based on availability of the features.

A Scheme program is constructed from the `program` form. Gauche evaluates the constructed Scheme program on-the-fly.

Each *program-clause* needs to be one of the "Program Clauses" below.

`requires` *feature-id feature-id2 . . .* [Program Clause]

[SRFI-7] The *feature-id*'s are the same as `srfi-0`'s (see Section 4.12 [Feature conditional], page 68). It tells that the following code requires these *feature-id*'s.

If a feature-id which is not supported in Gauche is given, an error is signaled.

`files` *filename . . .* [Program Clause]

[SRFI-7] Inserts the content of the *filenames* into a program. In Gauche, this clause just causes *filenames* to be loaded into the current module.

`code` *scheme-expression . . .* [Program Clause]

[SRFI-7] The *scheme-expressions* are inserted into a program.

`feature-cond` *clause clause2 . . .* [Program Clause]

[SRFI-7] *Clause* is a following form:

```
(requirement program-clause program-clause2 ...)
```

Where *requirement* should be one of the following:

- *feature-id*
- `(and requirement ...)`
- `(or requirement ...)`
- `(not requirement)`

The *requirement* of the last *clause* may be `else`.

Gauche checks each *requirement* one by one, and if it finds a fulfilled *requirement*, inserts the *program-clauses* in that *clause* into the program.

11.5 `srfi-13` - String library

`srfi-13` [Module]

Defines a large set of string-related functions. In Gauche, those functions are splitted to number of files and the form `(use srfi-13)` merely sets up autoloading of those files. So it is not likely to slow down the script startup. See SRFI-13 ([SRFI-13], page 830) for the detailed specification and discussion of design issues. This manual serves as a reference of function API. Some SRFI-13 functions are Gauche built-in and not listed here. Note: SRFI-13 documents suggests the name of the module that implements these functions to be "string-lib" and "string-lib-internals". Gauche uses the name "srfi-13" for consistency.

11.5.1 General conventions

There are a few common factors in string library API, which I don't repeat in each function description

argument convention

The following argument names imply their types.

s, *s1*, *s2* Those arguments must be strings.

char/char-set/pred
: This argument can be a character, a character-set object, or a predicate that takes a single character and returns a boolean value. "Applying *char/char-set/pred* to a character" means, if *char/char-set/pred* is a character, it is compared to the given character; if *char/char-set/pred* is a character set, it is checked if the character set contains the given character; if *char/char-set/pred* is a procedure, it is applied to the given character. "A character satisfies *char/char-set/pred*" means such application to the character yields true value.

start, *end*
: Lots of SRFI-13 functions takes these two optional arguments, which limit the area of input string from *start*-th character (inclusive) to *end*-th character (exclusive), where the operation is performed. When specified, the condition 0 <= *start* <= *end* <= *length of the string* must be satisfied. Default value of *start* and *end* is 0 and the length of the string, respectively.

`shared` *variant*
: Some functions have variants with "/shared" attached to its name. SRFI-13 defines those functions to allow to share the part of input string, for better performance. Gauche doesn't have a concept of shared string, and these functions are mere synonyms of their non-shared variants. However, Gauche *internally* shares the storage of strings, so generally you don't need to worry about the overhead of copying substrings.

`right` *variant*
: Most functions works from left to right of the input string. Some functions have variants with "-right" to its name, that works from right to left.

11.5.2 String predicates

`string-null?` *s* [Function]

[SRFI-13] {`srfi-13`} Returns `#t` if *s* is an empty string, `""`.

`string-every` *char/char-set/pred s :optional start end* [Function]

[SRFI-13] {`srfi-13`} Sees if every character in *s* satisfies *char/char-set/pred*. If so, `string-every` returns the value that is returned at the last application of *char/char-set/pred*. If any of the application returns `#f`, `string-every` returns `#f` immediately.

`string-any` *char/char-set/pred s :optional start end* [Function]

[SRFI-13] {`srfi-13`} Sees if any character in *s* satisfies *char/char-set/pred*. If so, `string-any` returns the value that is returned by the application. If no character satisfies *char/char-set/pred*, `#f` is returned.

11.5.3 String Constructors

`string-tabulate` *proc len* [Function]

[SRFI-13] {`srfi-13`} *proc* must be a procedure that takes an integer argument and returns a character. `string-tabulate` creates a string, whose *i*-th character is calculated by `(proc i)`.

```
(string-tabulate
  (lambda (i) (integer->char (+ i #x30))) 10)
 ⇒ "0123456789"
```

`reverse-list->string` *char-list* [Function]

[SRFI-13] {`srfi-13`} ≡ `(list->string (reverse char-list))`.

11.5.4 String selection

`substring/shared` *s start :optional end* [Function]

[SRFI-13] {`srfi-13`} In Gauche, this is the same as `substring`, except that the *end* argument is optional.

```
(substring/shared "abcde" 2) ⇒ "cde"
```

`string-copy!` *target tstart s :optional start end* [Function]

[SRFI-13] {`srfi-13`} Copies a string *s* into a string *target* from the position *tstart*. The *target* string must be mutable. Optional *start* and *end* arguments limits the range of *s*. If the copied string run over the end of *target*, an error is signaled.

```
(define s (string-copy "abcde"))
(string-copy! s 2 "ZZ")
s ⇒ "abZZe"
```

It is ok to pass the same string to *target* and *s*; this always work even if the regions of source and destination are overlapping.

Note that Gauche encourages you to treat strings as immutable objects. Internally, a string is an indirect pointer to a immutable entity, and mutating a string means copying the original entity and creating a new one. It doesn't "save allocations". Always use the functional version `string-copy` unless you absolutely need to replace a string in-place. See Section 6.12.7 [String utilities], page 158.

`string-take` *s nchars* [Function]
`string-drop` *s nchars* [Function]
`string-take-right` *s nchars* [Function]
`string-drop-right` *s nchars* [Function]

[SRFI-13] {`srfi-13`} Returns the first *nchars*-character string of *s* (`string-take`) or the string without first *nchars* (`string-drop`). The `*-right` variation counts from the end of string. It is guaranteed that the returned string is always a copy of *s*, even no character is dropped.

```
(string-take "abcde" 2) ⇒ "ab"
(string-drop "abcde" 2) ⇒ "cde"

(string-take-right "abcde" 2) ⇒ "de"
(string-drop-right "abcde" 2) ⇒ "abc"
```

`string-pad` *s len :optional char start end* [Function]
`string-pad-right` *s len :optional char start end* [Function]

[SRFI-13] {`srfi-13`} If a string *s* is shorter than *len*, returns a string of *len* where *char* is padded to the left or right, respectively. If *s* is longer than *len*, the rightmost or leftmost *len* chars are taken. *Char* defaults to `#\space`. If *start* and *end* are provided, the substring of *s* is used as the source.

```
(string-pad "abc" 10)    ⇒ "       abc"
(string-pad "abcdefg" 3) ⇒ "efg"

(string-pad-right "abc" 10) ⇒ "abc       "

(string-pad "abcdefg" 10 #\+ 2 5)
  ⇒ "+++++++cde"
```

`string-trim` *s :optional char/char-set/pred start end* [Function]
`string-trim-right` *s :optional char/char-set/pred start end* [Function]

`string-trim-both` *s :optional char/char-set/pred start end* [Function]

[SRFI-13] {`srfi-13`} Removes characters that match *char/char-set/pred* from *s*. `String-trim` removes the characters from left of *s*, `string-trim-right` does from right, and `string-trim-both` does from both sides. *Char/char-set/pred* defaults to `#[\s]`, i.e. a char-set of whitespaces. If *start* and *end* are provided, the substring of *s* is used as the source.

```
(string-trim "   abc  ")        ⇒ "abc  "
(string-trim-right "   abc  ") ⇒ "   abc"
(string-trim-both "   abc  ")  ⇒ "abc"
```

11.5.5 String comparison

`string-compare` *s1 s2 proc< proc= proc> :optional start1 end1 start2 end2* [Function]

`string-compare-ci` *s1 s2 proc< proc= proc> :optional start1 end1 start2 end2* [Function]

[SRFI-13] {`srfi-13`} Compares two strings *s1* and *s2* codepoint-wise from left. When mismatch is found at the index *k* of *s1*, calls *proc<* with *k* if *s1*'s codepoint is smaller than the corresponding *s2*'s, or calls *proc>* if *s1*'s one is greater than *s2*'s. If two strings are the same, calls *proc=* with the index of the last compared position in *s1*.

```
(string-compare "abcd" "abzd"
                (^i `(< ,i)) (^i `(= ,i)) (^i `(> ,i)))
  ⇒ (< 2)

(string-compare "abcd" "abcd"
                (^i `(< ,i)) (^i `(= ,i)) (^i `(> ,i)))
  ⇒ (= 3)
```

The optional arguments restricts the range of the input strings; however, the index passed to one of the procedures is always an index from the beginning of *s1*.

```
(string-compare "zzabcdyy" "abcz"
   (^i `(< ,i)) (^i `(= ,i)) (^i `(> ,i)) 2 6 0 4)
 ⇒ (< 5)

(string-compare "zzabcdyy" "abcz"
   (^i `(< ,i)) (^i `(= ,i)) (^i `(> ,i)) 2 5 0 3)

 ⇒ (= 4)
```

The case-insensitive variant, `string-compare-ci`, compares each codepoint with character-wise case-folding. It won't consider special case folding such as German eszett.

`string=` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string<>` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string<` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string<=` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string>` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string>=` *s1 s2 :optional start1 end1 start2 end2* [Function]

[SRFI-13] {`srfi-13`} Compare two strings *s1* and *s2*. Optional arguments can limit the portion of strings to be compared. Comparison is done by character-wise.

Note: The builtin procedures `string=?` etc. can also be used for character-wise string comparison, but they take arguments differently. See Section 6.12.6 [String Comparison], page 158.

`string-ci=` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-ci<>` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-ci<` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-ci<=` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-ci>` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-ci>=` *s1 s2 :optional start1 end1 start2 end2* [Function]

[SRFI-13] {`srfi-13`} Compare two strings *s1* and *s2* in case-insensitive way. Optional arguments can limit the portion of strings to be compared. Case folding and comparison is done by character-wise, so they don't consider case folding that affects multiple characters.

Note: We have two other sets of string comparison operations, both are named as `string-ci=?` etc. The builtin version (see Section 6.12.6 [String Comparison], page 158) does character-wise comparison. The one in `gauche.unicode` uses full-string case conversion (see Section 9.35.3 [Full string case conversion], page 475). R7RS version is the latter.

`string-hash` *s :optional bound start end* [Function]
`string-hash-ci` *s :optional bound start end* [Function]

[SRFI-13] {`srfi-13`} (Note: Gauche has builtin `string-hash` and `string-ci-hash` according to SRFI-128. See Section 6.2.3 [Hashing], page 100, for the details. SRFI-13's API is upper-compatible to SRFI-128's. The underlying hash algorithm is the same as the builtin ones, so `string-hash` returns the same value as the builtin ones for the same string if optional arguments are omitted. On the other hand, the builtin `string-ci-hash` uses string case folding (e.g. German eszett and `SS` are the same), while SRFI-13's `string-hash-ci` uses character-wise case folding. Unless there's a strong reason, we recommend new code should use builtin SRFI-128 version instead of SRFI-13.)

Calculates hash value of a string *s*. For `string-hash-ci`, character-wise case folding is done before calculating the hash value.

If the optional *bound* argument is given, it must be a positive exact integer, and the return value is limited below it. The optional *start* and *end* arguments allows using that portion for calculation.

11.5.6 String Prefixes & Suffixes

`string-prefix-length` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-suffix-length` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-prefix-length-ci` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-suffix-length-ci` *s1 s2 :optional start1 end1 start2 end2* [Function]

[SRFI-13] {`srfi-13`} Returns the length of the longest common prefix/suffix of two strings, *s1* and *s2*. The optional arguments restrict the range of search. The `*-ci` variations use case folding character comparison.

```
(string-prefix-length "abacus" "abalone")   ⇒ 3
(string-prefix-length "machine" "umbrella") ⇒ 0
(string-suffix-length "peeking" "poking")   ⇒ 4

(string-prefix-length "obvious" "oblivious" 2 7 4 9)
  ⇒ 5
```

`string-prefix?` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-suffix?` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-prefix-ci?` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-suffix-ci?` *s1 s2 :optional start1 end1 start2 end2* [Function]

[SRFI-13] {`srfi-13`} Returns true iff *s1* is a prefix or suffix of *s2*, respectively. The optional arguments limit the range of *s1* and *s2* to look at. The `*-ci` variations use case folding character comparison.

```
(string-prefix? "sch" "scheme")   ⇒ #t
(string-prefix? "lisp" "scheme")  ⇒ #f

(string-suffix? "eme" "scheme")   ⇒ #t
(string-suffix? "eme" "lisp")     ⇒ #f

(string-prefix? "mit-scheme" "scheme-family" 4) ⇒ #t
```

11.5.7 String searching

`string-index` *s char/char-set/pred :optional start end* [Function]
`string-index-right` *s char/char-set/pred :optional start end* [Function]

[SRFI-13] {`srfi-13`} Looks for the first element in a string *s* that matches *char/char-set/pred*, and returns its index. If *char/char-set/pred* is not found in *s*, returns `#f`. Optional *start* and *end* limit the range of *s* to search.

```
(string-index "Aloha oe" #\a) ⇒ 4
(string-index "Aloha oe" #[Aa]) ⇒ 0
(string-index "Aloha oe" #[\s]) ⇒ 5
(string-index "Aloha oe" char-lower-case?) ⇒ 1
(string-index "Aloha oe" #\o 3) ⇒ 6
```

See also the Gauche built-in procedure `string-scan` (Section 6.12.7 [String utilities], page 158), if you need speed over portability.

`string-skip` *s char/char-set/pred :optional start end* [Function]
`string-skip-right` *s char/char-set/pred :optional start end* [Function]

[SRFI-13] {`srfi-13`} Looks for the first element that does not match *char/char-set/pred* and returns its index. If such element is not found, returns `#f`. Optional *start* and *end* limit the range of *s* to search.

`string-count` *s char/char-set/pred :optional start end* [Function]

[SRFI-13] {`srfi-13`} Counts the number of elements in *s* that matches *char/char-set/pred*. Optional *start* and *end* limit the range of *s* to search.

`string-contains` *s1 s2 :optional start1 end1 start2 end2* [Function]
`string-contains-ci` *s1 s2 :optional start1 end1 start2 end2* [Function]

[SRFI-13] {`srfi-13`} Looks for a string *s2* inside another string *s1*. If found, returns an index in *s1* from where the matching string begins. Returns `#f` otherwise. Optional *start1*, *end1*, *start2* and *end2* limits the range of *s1* and *s2*.

See also the Gauche built-in procedure `string-scan` (Section 6.12.7 [String utilities], page 158), if you need speed over portability.

11.5.8 String case mapping

`string-titlecase` *s :optional start end* [Function]
`string-titlecase!` *s :optional start end* [Function]
`string-upcase` *s :optional start end* [Function]
`string-upcase!` *s :optional start end* [Function]
`string-downcase` *s :optional start end* [Function]
`string-downcase!` *s :optional start end* [Function]

[SRFI-13] {`srfi-13`} Converts a string *s* to titlecase, upcase or downcase, respectively. These operations uses character-by-character mapping provided by `char-upcase` etc. That is, `string-upcase` and `string-downcase` can be understood as follow:

```
(string-upcase s)
```

```
  ≡ (string-map char-upcase s)
(string-downcase s)
  ≡ (string-map char-downcase s)
```

If you need full case mapping that handles the case when a character is mapped to more than one characters, use the procedures with the same name in `gauche.unicode` module (see Section 9.35.3 [Full string case conversion], page 475).

The linear-update version `string-titlecase!`, `string-upcase!` and `string-downcase!` destroys *s* to store the result. Note that in Gauche, using those procedures doesn't save anything, since string mutation is expensive by design. They are provided merely for completeness.

11.5.9 String reverse & append

`string-reverse` *s :optional start end* [Function]
`string-reverse!` *s :optional start end* [Function]

[SRFI-13] {`srfi-13`} Returns a string in which the character positions are reversed from *s*. `string-reverse!` modifies *s*.

```
(string-reverse "mahalo") ⇒ "olaham"
(string-reverse "mahalo" 3) ⇒ "ola"
(string-reverse "mahalo" 1 4) ⇒ "aha"

(let ((s (string-copy "mahalo")))
  (string-reverse! s 1 5)
  s)
  ⇒ "mlahao"
```

`string-concatenate` *string-list* [Function]

[SRFI-13] {`srfi-13`} Concatenates list of strings.

```
(string-concatenate '("humuhumu" "nukunuku" "apua" "‘a"))
  ⇒ "humuhumunukunukuapua‘a"
```

`string-concatenate/shared` *string-list* [Function]
`string-append/shared` *s . . .* [Function]

[SRFI-13] {`srfi-13`} "Shared" version of `string-concatenate` and `string-append`. In Gauche, these are just synonyms of them.

`string-concatenate-reverse` *string-list* [Function]
`string-concatenate-reverse/shared` *string-list* [Function]

[SRFI-13] {`srfi-13`} Reverses *string-list* before concatenation. "Shared" version works the same in Gauche.

11.5.10 String mapping

`string-map` *proc s :optional start end* [Function]
`string-map!` *proc s :optional start end* [Function]

[SRFI-13] {`srfi-13`} `string-map` applies *proc* on every character of *s*, and collects the results into a string and returns it. On the other hand, `string-map!` modifies *s*.

```
(string-map char-upcase "wikiwiki") ⇒ "WIKIWIKI"
(string-map char-upcase "wikiwiki" 4) ⇒ "WIKI"

(let ((s (string-copy "wikiwiki")))
  (string-map! char-upcase s 4)
  s)
  ⇒ "wikiWIKI"
```

`string-fold` *kons knil s :optional start end* [Function]
`string-fold-right` *kons knil s :optional start end* [Function]

[SRFI-13] {`srfi-13`} Like *fold* and *fold-right* (see Section 6.6.5 [Walking over lists], page 131), but works on a string instead of a list.

```
(string-fold cons '() "abcde")
  ⇒ (#\e #\d #\c #\b #\a)
(string-fold-right cons '() "abcde")
  ⇒ (#\a #\b #\c #\d #\e)
```

`string-unfold` *p f g seed :optional base make-final* [Function]

[SRFI-13] {`srfi-13`} A fundamental string builder. The *p*, *f* and *g* are procedures, taking the current seed value. The stop predicate *p* determines when to stop: If it returns a true value, string building stops. The mapping function *f* returns a character from the current seed value. The next seed function *g* returns a next seed value from the current seed value. The *seed* argument gives the initial seed value.

```
(string-unfold (^n (= n 10))
               (^n (integer->char (+ n 48)))
               (^n (+ n 1))
               0)
  ⇒ "0123456789"
```

The optional argument *base* is, when given, prepended to the result string. Another optional argument *make-final* is a procedure that takes the last return value of *g* and returns a string that becomes the suffix of the result string.

```
(string-unfold (^n (= n 10))
               (^n (integer->char (+ n 48)))
               (^n (+ n 1))
               0 "foo" x->string)
  ⇒ "foo012345678910"
```

`string-unfold-right` *p f g seed :optional base make-final* [Function]

[SRFI-13] {`srfi-13`} Another fundamental string builder. The meanings of arguments are the same as '`string-unfold`'. The only difference is that the string is build right-to-left. The optional *base*, if given, becomes the suffix of result, and the result of *make-final* becomes the prefix.

```
(string-unfold-right (^n (= n 10))
                     (^n (integer->char (+ n 48)))
                     (^n (+ n 1))
                     0 "foo" x->string)
  ⇒ "109876543210foo"
```

`string-for-each` *proc s :optional start end* [Function]

[SRFI-13] {`srfi-13`} Apply *proc* on each character of string *s*, from left to right. Optional *start* and *end* arguments limit the range of the input string.

`string-for-each-index` *proc s :optional start end* [Function]

[SRFI-13] {`srfi-13`} Call *proc* on each index of the string *s*.

11.5.11 String rotation

`xsubstring` *s from :optional to start end* [Function]

[SRFI-13] {`srfi-13`} Takes a substring of infinite repetition of string *s* between index *from* (inclusive) and index *to* (exclusive).

For example, if *s* is `"abcde"`, we repeat it infinitely to both sides. So *5n*-th character for integer *n* is always `#\a`, which extends negative *n* as well.

```
(xsubstring "abcde" 2 10)
  ⇒ "cdeabcde"
(xsubstring "abcde" -9 -2)
  ⇒ "bcdeabc"
```

`string-xcopy!` *target tstart s sfrom :optional sto start end* [Function]
[SRFI-13] {`srfi-13`}

11.5.12 Other string operations

`string-replace` *s1 s2 start1 end1 :optional start2 end2* [Function]
[SRFI-13] {`srfi-13`} Returns a new string whose content is a copy of a string *s1*, except the part beginning from the index *start1* (inclusive) and ending at the index *end1* (exclusive) are replaced by a string *s2*. When optional *start2* and *end2* arguments are given, *s2* is trimmed first according to them. The size of the *gap*, `(- end1 start1)`, doesn't need to be the same as the size of the inserted string. Effectively, this is the same as the following code.

```
(string-append (substring s1 0 start1)
               (substring s2 start2 end2)
               (substring s1 end1 (string-length s1)))
```

`string-tokenize` *s :optional token-set start end* [Function]
[SRFI-13] {`srfi-13`} Splits the string *s* into a list of substrings, where each substring is a maximal non-empty contiguous sequence of characters from the character set *token-set*. The default of *token-set* is `char-set:graphic` (see Section 6.11 [Character set], page 147).

See also Gauche's built-in `string-split` (see Section 6.12.7 [String utilities], page 158), which provides similar features but different criteria.

11.5.13 String filtering

`string-filter` *char/char-set/pred s :optional start end* [Function]
`string-delete` *char/char-set/pred s :optional start end* [Function]
[SRFI-13] {`srfi-13`} Returns a string consists of characters in a string *s* that passes (or don't pass) the test indicated by *char/char-set/pred*, respectively.

```
(string-filter char-upper-case? "Hello, World!")
  ⇒ "HW"

(string-delete char-upper-case? "Hello, World!")
  ⇒ "ello, orld!"

(string-delete #\l "Hello, World!")
  ⇒ "Heo, Word!"

(string-filter #[\w] "Hello, World!")
  ⇒ "HelloWorld"
```

Note: Srfi-13 was revised after finalization to switch the order of arguments *char/char-set/pred* and *s* was. At the time of finalization, the order was `(string-filter s pred)` and Gauche implemented it accordingly. However, most existing implementations follows the revised order, since that was what the srfi-13 reference implementation had.

So, from 0.9.4, we revised the API to comply the current srfi-13 spec, but we also accept the old order as well not to break the old code. We recommend the new code to use the new order.

11.5.14 Low-level string procedures

`string-parse-start+end` *proc s args* [Function]
`string-parse-final-start+end` *proc s args* [Function]
[SRFI-13] {`srfi-13`}

`let-string-start+end` (*start end* [*rest*]) *proc-exp s-exp args-exp body* ... [Macro]
[SRFI-13] {`srfi-13`}

`check-substring-spec` *proc s start end* [Function]
`substring-spec-ok?` *s start end* [Function]
[SRFI-13] {`srfi-13`}

`make-kmp-restart-vector` *s :optional c= start end* [Function]
[SRFI-13] {`srfi-13`}

`kmp-step` *pat rv c i c= p-start* [Function]
[SRFI-13] {`srfi-13`}

`string-kmp-partial-search` *pat rv s i :optional c= p-start s-start s-end* [Function]
[SRFI-13] {`srfi-13`}

11.6 `srfi-14` - Character-set library

`srfi-14` [Module]
SRFI-14 has become a part of R7RS large. See Section 10.3.6 [R7RS character sets], page 533.

11.7 `srfi-19` - Time data types and procedures

`srfi-19` [Module]
This SRFI defines various representations of time and date, and conversion methods among them.

On Gauche, time object is supported natively by `<time>` class (see Section 6.25.9 [Time], page 269). Date object is supported by `<date>` class described below.

11.7.1 Time types

Time type is represented by a symbol. This module defines the following constant variables that is bound to its name, for convenience.

`time-utc` [Constant]
[SRFI-19] {`srfi-19`} UTC time. Gauche's built-in `current-time` always returns this type (see Section 6.25.9 [Time], page 269).

`time-tai` [Constant]
[SRFI-19] {`srfi-19`} International Atomic Time. This time is a bit larger than UTC, due to the leap seconds.

`time-monotonic` [Constant]
[SRFI-19] {`srfi-19`} Implementation-dependent monotonically increasing time. In Gauche, this is the same as `time-tai`.

`time-duration` [Constant]
[SRFI-19] {`srfi-19`} Duration between two absolute time points.

`time-process` [Constant]
[SRFI-19] {`srfi-19`} CPU time in current process. Gauche calculates this from user time and system time returned by POSIX times(3).

`time-thread` [Constant]
[SRFI-19] {`srfi-19`} CPU time in current thread. In the current implementation, this is the same as `time-process`.

11.7.2 Time queries

`current-time` *:optional time-type* [Function]
[SRFI-19] {`srfi-19`} Extends Gauche built-in `current-time` (see Section 6.25.9 [Time], page 269) to take optional *time-type* argument to specify the desired time type. *time-type* must be one of the types described in Section 11.7.1 [SRFI-19 Time types], page 600.

`current-date` *:optional tz-offset* [Function]
[SRFI-19] {`srfi-19`} Returns the current date as an instance of `<date>` class (see Section 11.7.4 [SRFI-19 Date], page 602). If *tz-offset* is given, it must be an offset from UTC in number of seconds. If *tz-offset* is not given, returns the date in local time zone.

`current-julian-day` [Function]
[SRFI-19] {`srfi-19`} Returns the current julian day, a point in time as a real number of days since -4714-11-24T12:00:00Z (November 24, -4714 at noon, UTC).

`current-modified-julian-day` [Function]
[SRFI-19] {`srfi-19`} Returns the current modified julian day, a point in time as a real number of days since 1858-11-17T00:00:00Z (November 17, 1858 at midnight, UTC).

`time-resolution` [Function]
[SRFI-19] {`srfi-19`}

11.7.3 Time procedures

`make-time` *type nanoseconds seconds* [Function]
[SRFI-19] {`srfi-19`} Returns an instance of `<time>` class with specified initial values. Equivalent to `(make <time> :type type :second seconds :nanosecond nanoseconds)`.

(This function had been defined incorrectly before release 0.6.8; the arguments *seconds* and *nanoseconds* were switched. Please check your code if it uses `make-time`).

`time-type` *time* [Function]
`time-second` *time* [Function]
`time-nanosecond` *time* [Function]
`set-time-type!` *time type* [Function]
`set-time-second!` *time second* [Function]
`set-time-nanosecond!` *time nanosecond* [Function]
[SRFI-19] {`srfi-19`} Getter and setter of `<time>` object slots.

`copy-time` *time* [Function]
[SRFI-19] {`srfi-19`} Returns a new instance of `<time>` whose content is the same as given *time*

`time=?` *time0 time1* [Function]
`time<?` *time0 time1* [Function]
`time<=?` *time0 time1* [Function]
`time>?` *time0 time1* [Function]
`time>=?` *time0 time1* [Function]
[SRFI-19] {`srfi-19`} Compares two times. Types of both times must match.

`time-difference` *time0 time1* [Function]
`time-difference!` *time0 time1* [Function]

[SRFI-19] {`srfi-19`} Returns the difference of two times, in `time-duration` time. Types of both times must match. `Time-difference!` modifies *time0* to store the result.

`add-duration` *time0 time-duration* [Function]
`add-duration!` *time0 time-duration* [Function]
`subtract-duration` *time0 time-duration* [Function]
`subtract-duration!` *time0 time-duration* [Function]

[SRFI-19] {`srfi-19`} Adds or subtracts *time-duration* to or from *time0*. Type of returned time is the same as *time0*. Type of *time-duration* must be `time-duration`. `add-duration!` and `subtract-duration!` reuse *time0* to store the result.

11.7.4 Date

`<date>` [Class]

{`srfi-19`} Represents a date.

`nanosecond` [Instance Variable of `<date>`]

Nanosecond portion of the date by an integer between 0 and 999,999,999, inclusive.

`second` [Instance Variable of `<date>`]

Second portion of the date by an integer between 0 and 60, inclusive. (60 for leap second).

`minute` [Instance Variable of `<date>`]

Minute portion of the date by an integer between 0 and 59, inclusive.

`hour` [Instance Variable of `<date>`]

Hour portion of the date by an integer between 0 and 23, inclusive.

`day` [Instance Variable of `<date>`]

Day portion of the date by an integer between 0 and 31, inclusive. The actual upper bound of the day is determined by the year and the month. (Note: 1 is for the first day; 0 is allowed by the specification, but I don't see why).

`month` [Instance Variable of `<date>`]

Month portion of the date by an integer between 1 and 12, inclusive. 1 for January, 2 for February, and so on. (Note: this is different from POSIX's `<sys-tm>` convention).

`year` [Instance Variable of `<date>`]

Year portion of the date.

`zone-offset` [Instance Variable of `<date>`]

The number of seconds east of GMT for this timezone, by an integer.

`make-date` *nanosecond second minute hour day month year zone-offset* [Function]

[SRFI-19] {`srfi-19`} Makes a `<date>` object from the given values. Note: this procedure does not check if the values are in the valid range.

`date?` *obj* [Function]

[SRFI-19] {`srfi-19`} Returns true iff *obj* is a `<date>` object.

`date-nanosecond` *date* [Function]
`date-second` *date* [Function]
`date-minute` *date* [Function]
`date-hour` *date* [Function]

`date-day` *date* [Function]
`date-month` *date* [Function]
`date-year` *date* [Function]
`date-zone-offset` *date* [Function]
[SRFI-19] {`srfi-19`} Accessors.

`date-year-day` *date* [Function]
`date-week-day` *date* [Function]
`date-week-number` *date day-of-week-starting-week* [Function]
[SRFI-19] {`srfi-19`} Calculates the day number in the year (1 for January 1st), the day number in the week (0 for Sunday, 1 for Monday, ...), and the ordinal week of the year which holds this date, ignoring a first partial week, respectively.

Day-of-week-starting-week is the integer corresponding to the day of the week which is to be considered the first day of the week (Sunday=0, Monday=1, etc.).

`date->julian-day` *date* [Function]
`date->modified-julian-day` *date* [Function]
`date->time-monotonic` *date* [Function]
`date->time-tai` *date* [Function]
`date->time-utc` *date* [Function]
[SRFI-19] {`srfi-19`} Conversions from date to various date/time types.

`julian-day->date` *jd :optional tz-offset* [Function]
`julian-day->time-monotonic` *jd* [Function]
`julian-day->time-tai` *jd* [Function]
`julian-day->time-utc` *jd* [Function]
[SRFI-19] {`srfi-19`} Conversions from julian-day to various date/time types.

`modified-julian-day->date` *jd :optional tz-offset* [Function]
`modified-julian-day->time-monotonic` *jd* [Function]
`modified-julian-day->time-tai` *jd* [Function]
`modified-julian-day->time-utc` *jd* [Function]
[SRFI-19] {`srfi-19`} Conversions from modified julian-day to various date/time types.

`time-monotonic->date` *time :optional tz-offset* [Function]
`time-monotonic->julian-day` *time* [Function]
`time-monotonic->modified-julian-day` *time* [Function]
`time-monotonic->time-tai` *time* [Function]
`time-monotonic->time-tai!` *time* [Function]
`time-monotonic->time-utc` *time* [Function]
`time-monotonic->time-utc!` *time* [Function]
[SRFI-19] {`srfi-19`} Conversions from time-monotonic to various date/time types.

`time-tai->date` *time :optional tz-offset* [Function]
`time-tai->julian-day` *time* [Function]
`time-tai->modified-julian-day` *time* [Function]
`time-tai->time-monotonic` *time* [Function]
`time-tai->time-monotonic!` *time* [Function]
`time-tai->time-utc` *time* [Function]
`time-tai->time-utc!` *time* [Function]
[SRFI-19] {`srfi-19`} Conversions from time-tai to various date/time types.

`time-utc->date` *time :optional tz-offset* [Function]
`time-utc->julian-day` *time* [Function]

`time-utc->modified-julian-day` *time* [Function]
`time-utc->time-monotonic` *time* [Function]
`time-utc->time-monotonic!` *time* [Function]
`time-utc->time-tai` *time* [Function]
`time-utc->time-tai!` *time* [Function]

[SRFI-19] {`srfi-19`} Conversions from time-utc to various date/time types.

11.7.5 Date reader and writer

`date->string` *date :optional format-string* [Function]

[SRFI-19+] {`srfi-19`} Converts a `<date>` object to a string, according to the format specified by *format-string*. If *format-string* is omitted, `"~c"` is assumed.

A format string is copied to output, except a sequence begins with ~ which is replaced with the following rules:

`~~` A literal ~.

`~a` Locale's abbreviated weekday name (Sun...Sat).

`~A` Locale's full weekday name (Sunday...Saturday).

`~b` Locale's abbreviate month name (Jan...Dec).

`~B` Locale's full month name (January...December).

`~c` Locale's date and time (e.g., "Fri Jul 14 20:28:42-0400 2000").

`~d` Day of month, zero padded (01...31).

`~D` Date (mm/dd/yy).

`~e` Day of month, blank padded (1...31).

`~f` Seconds+fractional seconds, using locale's decimal separator (e.g. 5.2).

`~h` Same as ~b.

`~H` Hour, zero padded, 24-hour clock (00...23).

`~I` Hour, zero padded, 12-hour clock (01...12).

`~j` Day of year, zero padded.

`~k` Hour, blank padded, 24-hour clock (0...23).

`~l` Hour, blank padded, 12-hour clock (1...12).

`~m` Month, zero padded (01...12).

`~M` Minute, zero padded (00...59).

`~n` New line.

`~N` Nanosecond, zero padded.

`~p` Locale's AM or PM.

`~r` Time, 12 hour clock, same as "~I:~M:~S ~p".

`~s` Number of full seconds since "the epoch" (in UTC).

`~S` Second, zero padded (00...60).

`~t` Horizontal tab.

`~T` Time, 24 hour clock, same as "~H:~M:~S".

`~U` Week number of year with Sunday as first day of week (01...53).

`~V` Week number of year with Monday as first day of week (00...52).

`~w` Day of week (0...6).

`~W` Week number of year with Monday as first day of week (00...52).

`~x` Locale's date representation, for example: "07/31/00".

`~X` Locale's time representation, for example: "06:51:44".

`~y` Last two digits of year (00...99).

`~Y` Year.

`~z` Time zone in RFC-822 style.

`~1` ISO-8601 year-month-day format.

`~2` ISO-8601 hour-minute-second-timezone format.

`~3` ISO-8601 hour-minute-second format.

`~4` ISO-8601 year-month-day-hour-minute-second-timezone format.

`~5` ISO-8601 year-month-day-hour-minute-second format.

Note: currently Gauche doesn't honor process's locale setting, and it always formats the date as if the locale is "C". It may be changed in future, so you shouldn't rely on, for example, `~a` always formatted as "Sun".."Sat".

There's no portable way to ensure you'll get "C" locale formats since there's no standard way to set process's locale yet. However, Gauche provides a way to ensure the locale to be "C", as an extension to srfi-19. Insert `@` between `~` and the directive character, such as `~@a`.

`string->date` *string template-string* [Function]
[SRFI-19] {`srfi-19`}

11.8 `srfi-27` - Sources of Random Bits

`srfi-27` [Module]
This module provides SRFI-27 pseudo random generator interface, using Mersenne Twister algorithm (see Section 12.26 [Mersenne-Twister random number generator], page 704) as the backbone.

`random-integer` *n* [Function]
[SRFI-27] {`srfi-27`} Returns a random exact integer between [0, *n*-1], inclusive, using the default random source. To set a random seed for this procedure, use `random-source-randomize!` or `random-source-pseudo-randomize!` on `default-random-source`.

`random-real` [Function]
[SRFI-27] {`srfi-27`} Returns a random real number between (0, 1), exclusive, using the default random source. To set a random seed for this procedure, use `random-source-randomize!` or `random-source-pseudo-randomize!` on `default-random-source`.

`default-random-source` [Variable]
[SRFI-27] {`srfi-27`} Keeps the default random source that is used by `random-integer` and `random-real`.

`make-random-source` [Function]
[SRFI-27] {`srfi-27`} Creates and returns a new random source. In the current Gauche implementation, it is just a `<mersenne-twister>` object. It may be changed in the future implementation.

random-source? *obj* [Function]

[SRFI-27] {srfi-27} Returns #t if *obj* is a random source object.

random-source-state-ref *s* [Function]

random-source-state-set! *s state* [Function]

[SRFI-27] {srfi-27} Gets and sets the "snapshot" of the state of the random source *s*. *State* is an opaque object whose content depends on the backbone generator.

random-source-randomize! *s* [Function]

[SRFI-27] {srfi-27} Makes an effort to set the state of the random source *s* to a truly random state. The current implementation uses the current time and the process ID to set the random seed.

random-source-pseudo-randomize! *s i j* [Function]

[SRFI-27] {srfi-27} Changes the state of the random source *s* into the initial state of the (*i*, *j*)-th independent random source, where *i* and *j* are non-negative integers. This procedure can be used to reuse a random source *s* as large number of independent random source, indexed by two non-negative integers. Note that this procedure is entirely deterministic.

random-source-make-integers *s* [Function]

[SRFI-27] {srfi-27} Returns a procedure, that takes one integer argument *n* and returns a random integer between 0 and *n*-1 inclusive for every invocation, from the random source *s*.

random-source-make-reals *s :optional unit* [Function]

[SRFI-27] {srfi-27} Returns a procedure, that takes no argument and returns a random real between 0 and 1 exclusive for every invocation, from the random source *s*. If *unit* is given, the random real the returned procedure generates will be quantized by the given *unit*, where 0 < *unit* < 1.

11.9 srfi-29 - Localization

srfi-29 [Module]

This module implements the message localization mechanism defined in SRFI-29.

In fact, this module consists of two submodules, `srfi-29.bundle` and `srfi-29.format`. The module `srfi-29` extends both submodules. It is because srfi-29's definition of the `format` procedure is incompatible to Gauche's native `format` (thus Common Lisp's `format`) in the handling of `~@*` directive.

So I splitted the module into two, `srfi-29.format` which contains srfi-29's `format`, and `srfi-29.bundle` which contains the rest ("bundle" API). If a program wishes a complete compatibility of srfi-29, use `srfi-29` module, which overrides Gauche's native `format`. If a program just wants srfi-29's "bundle" API, but wants to keep Gauche's `format`, use `srfi-29.bundle`.

A localization feature is also provided by `text.gettext` module (see Section 12.54 [Localized messages], page 783), which is a preferable way of message localization in Gauche. This module is provided mainly for porting code that uses srfi-29 features.

Bundle specifier

A *bundle specifier* is an arbitrary list of symbols, but typically it takes the form like:

```
(package language country details ...)
```

Where *package* specifies the software package, *language* and *country* specifies language and country code, and *details* gives other informations like encoding.

The values for the default bundle specifier can be obtained by the following parameters.

`current-language` [Parameter]
`current-country` [Parameter]
`current-locale-details` [Parameter]

[SRFI-29] {`srfi-29`} The `current-language` and `current-country` parameters keep the ISO 639-1 language code and ISO 3166-1 country code respectively, both as symbols. The `current-locale-details` keeps a list of auxiliary local informations, such as encodings.

These parameters are initialized if `LANG` environment variable is set in the form of `lang_country.encoding` format. For example, if the `LANG` setting is `ja_JP.eucJP`, those parameters are `ja`, `jp`, and `(eucjp)`, respectively. If `LANG` is `C` or undefined, the default values are `en`, `us`, and `()`, respectively.

Bundle preparation

`declare-bundle!` *bundle-specifier association-list* [Function]

[SRFI-29] {`srfi-29`} Put the association list of template key (symbol) and the locale-specific message (string) into the bundle database, with *bundle-specifier* as the key.

Gauche currently supports only in-memory bundle database. That is, you have to call `declare-bundle!` within the application in order to lookup the localized messages.

`save-bundle!` *bundle-specifier* [Function]
`load-bundle!` *bundle-specifier* [Function]

[SRFI-29] {`srfi-29`} Since Gauche doesn't support persistent bundle database yet, these procedures does nothing and returns `#f`. (It is still conforming behavior of srfi-29).

Retrieving localized message

`localized-template` *package-name message-template-name* [Function]

[SRFI-29] {`srfi-29`} Retrieves localized message, associated with a symbol *message-template-name* in the package *package-name*.

Extended format procedure

`format` *format-string args* [Function]

[SRFI-29] {`srfi-29`} SRFI-29 extends SRFI-28's `format` procedure spec (which supports `~a`, `~s`, `~%` and `~~` directives), in order to support argument repositioning.

A directive `~N@*`, where N is an integer or can be omitted, causes the next directive to retrieve a value from *N*-th optional argument. The referenced value isn't consumed, and won't affect the processing of subsequent directives.

Although SRFI-28 spec is compatible to Gauche's native `format` (see Section 6.22.8 [Output], page 231), this SRFI-29 extension isn't. Specifically, the `~N@*` directive of Gauche's `format` changes the argument pointer to points *N*-th optional argument, thus it affects all the subsequent arguments.

Because of this incompatibility, this function is defined in a separate module, `srfi-29.format`. If you use `srfi-29`, which extends `srfi-29.bundle` and `srfi-29.format`, the `format` procedure will be overridden by srfi-29's `format` in your module. If you want to keep Gauche's native `format`, use `srfi-29.bundle` only.

11.10 `srfi-37` - args-fold: a program argument processor

`srfi-37` [Module]

This module implements `args-fold`, yet another procedure to process command-line arguments, defined in SRFI-37 ([SRFI-37], page 831).

Unlike `gauche.parseopt` (see Section 9.23 [Parsing command-line options], page 413), `args-fold` provides functional interface, i.e. the user's states are explicitly passed via parser's argument and return values, and also follows POSIX and GNU getopt guidelines, including long options.

`args-fold` *args options unrecognized-proc operand-proc :rest seeds* [Function]

{`srfi-37`} Processes program options *args* from left to right, according to given option specification *options*, and two procedures *unrecognized-proc* and *operand-proc*.

Options is a list of option objects, explained below. Each option object keeps the name(s) of the option, a flag to specify whether the option takes an argument or not, and a procedure to process that option (we'll call it *option procedure*).

`Args-fold` recognizes both single-character options (short options) and long options. A short option must begin with single hyphen (e.g. `-a`), while long option must begin with double hyphens (e.g. `--help`). Short options can be concatenated, e.g. `-abc` or `-a -b -c`. Both a short option and a long option can take required or optional arguments. Required short-option argument can appear with or without space after the option, e.g. `-afoo` or `-a foo`. Long-option argument can appear after character '`=`' or space, e.g. `--long=foo` or `--long foo`.

When `args-fold` encounters a command-line argument that cannot be an option argument, and doesn't begin with hyphen, the argument is treated as an *operand*. `Args-fold` allows operands and options to be interleaved. However, if `args-fold` encounters '`--`', the rest of arguments are treated as operands, regardless of beginning with hyphen or not.

When the given option matches one of option object in *options*, the option procedure is called as follows:

```
(option-proc option name arg seed ...)
```

where *option* is the matched option object, *name* is the string actually used to specify the option, *arg* is the option argument (or `#f` if there's none), and *seed* . . . is the user's state information. *Option-proc* must return as many arguments as *seed*s.

When `args-fold` encounters an option that doesn't match any of the option objects, it creates a new option object for the option and calls *unrecognized-proc* with the same arguments as *option-proc*.

When `args-fold` finds an operand, *operand-proc* is called as follows:

```
(operand-proc operand seed ...)
```

Operand-proc must return as many arguments as *seed*s.

The caller's state should be explicitly passed around seed arguments and return values. The initial seed values are *seeds* given to `args-fold`. The values returned from option procedure, *unrecognized-proc* and *operand-proc* are used as the seed arguments of next invocation of those procedures. The values returned from the last call to the procedures are returned from `args-fold`.

`option` *names require-arg? optional-arg? processor* [Function]

{`srfi-37`} Creates an option object with the passed properties.

Names is a list of characters and/or strings. A character is used for a short option, and a string is used for a long option.

Two flags, *require-arg?* and *optional-arg?* indicates whether the option should take an option argument, or may take an option argument.

Processor is the option processor procedure.

Note that, if an option argument is passed using '`=`' character, it is passed to the option procedure even if the option has `#f` in both *require-arg?* and *optional-arg?*. It is up to the option procedure to deal with the argument.

It should also be noted that the optional option argument for a short option is only recognized if it is given without whitespace after the short option. That is, if a short option 'd' is marked to take optional option argument, then '-dfoo' is interpreted as '-d' with argument 'foo', but '-d foo' is interpreted as '-d' without argument and an operand foo. If 'd' is marked to take required option argument, however, both are interpreted as '-d' with argument 'foo'.

`option?` *obj* [Function]
{srfi-37} Returns `#t` if *obj* is an option object, `#f` otherwise.

`option-names` *option* [Function]
`option-required-arg?` *option* [Function]
`option-optional-arg?` *option* [Function]
`option-processor` [Function]
{srfi-37} Returns the properties of an option object *option*.

A simple example:

```
(use srfi-37)

(define options
 (list (option '(#\d "debug") #f #t
              (lambda (option name arg debug batch paths files)
                (values (or arg "2") batch paths files)))
       (option '(#\b "batch") #f #f
              (lambda (option name arg debug batch paths files)
                (values debug #t paths files)))
       (option '(#\I "include") #t #f
              (lambda (option name arg debug batch paths files)
                (values debug batch (cons arg paths) files)))))

(define (main args)
  (receive (debug-level batch-mode include-paths files)
    (args-fold (cdr args)
               options
               (lambda (option name arg . seeds)         ; unrecognized
                 (error "Unrecognized option:" name))
               (lambda (operand debug batch paths files) ; operand
                 (values debug batch paths (cons operand files)))
               0      ; default value of debug level
               #f     ; default value of batch mode
               '()    ; initial value of include paths
               '()    ; initial value of files
               )
     (print "debug level = " debug-level)
     (print "batch mode = " batch-mode)
     (print "include paths = " (reverse include-paths))
     (print "files = " (reverse files))
     0))
```

11.11 `srfi-41` - Streams

`srfi-41` [Module]
SRFI-41 has become a part of R7RS large. See Section 10.3.11 [R7RS stream], page 546.

11.12 `srfi-42` - Eager comprehensions

`srfi-42` [Module]

This module provides a generic comprehension mechanism, which some other languages (e.g. Haskell and Python) offer as a built-in mechanism. It provides a rich set of operators so it can be used not only as a list generator but as a generic loop construct (actually, some may say it is as powerful/evil as Common Lisp's *loop* macro).

It runs eagerly as the name suggests, that is, if it generates a list, it creates the entire list when evaluated, instead of generate the elements *on demand*. Thus it can't represent an infinite sequence, which Haskell's comprehension naturally does. Gauche offers a few alternatives to deal with lazy, possibly infinite, sequences: See Section 6.19.2 [Lazy sequences], page 200, Section 9.11 [Generators], page 372, and Section 12.71 [Stream library], page 809.

Eager comprehension examples

Let's begin with some examples.

Generate a list of squares for the first five integers:

```
(list-ec (: i 5) (* i i)) ⇒ (0 1 4 9 16)
```

`list-ec` is a comprehension macro that generates a list. The first form `(: i 5)` is called a *qualifier*, which specifies a set of values to repeat over (here it is each integer from 0 below 5). The last form `(* i i)` is called a *body*, which is ordinary Scheme expression evaluated for each values specified by the *qualifier*.

A comprehension can have more than one qualifiers. Next example generate set of pair of numbers `(x y)`, where `x` is between 2 (inclusive) and 5 (exclusive), and `y` is between 1 (inclusive) and x (exclusive).

```
(list-ec (: x 2 5) (: y 1 x) (list x y))
  ⇒ ((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))
```

The qualifiers works as *nested*; that is, `(: x 2 5)` specifies to repeat the rest of the clauses—`(: y 1 x)` and `(list x y)`.

The above two examples can be written in Haskell as the followings:

```
[ i*i   | i <- [0..4] ]
[ (x,y) | x <- [2..4], y <- [1..x-1] ]
```

Note the differences: (1) In Haskell, the body expression to yield the elements comes first, followed by qualifiers (selectors). In srfi-42, the body expression comes last. (2) In srfi-42, range operator's lower bound is inclusive but its upper bound is exclusive.

List a set of numbers `(a b c d)`, where `a^3+b^3 = c^3+d^3`:

```
(define (taxi-number n)
  (list-ec (: a 1 n)
           (: b (+ a 1) n)
           (: c (+ a 1) b)
           (: d (+ c 1) b)
           (if (= (+ (expt a 3) (expt b 3))
                  (+ (expt c 3) (expt d 3))))
           (list a b c d)))
```

If you want to change values of more than one variable simultaneously, instead of nesting, you can bundle the qualifiers like this:

```
(list-ec (:parallel (: x '(a b c d)) (: y '(1 2 3 4)))
         (list x y))
  ⇒ ((a 1) (b 2) (c 3) (d 4))
```

You can generate not only a list, but other sequences:

```
(vector-ec (: i 5) i) ⇒ #(0 1 2 3 4)
(string-ec (: i 5) (integer->char (+ i 65))) ⇒ "ABCDE"
```

Or apply folding operations:

```
(sum-ec (: i 1 100) i)
  ⇒ 4950    ;; sum of integers from 1 below 100.
(product-ec (: i 1 10) i)
  ⇒ 362880 ;; ... and product of them.
```

Comprehension macros

Each comprehension takes the following form.

```
(comprehension-macro qualifier ... body)
```

It evaluates *body* repeatedly as specified by *qualifier* .... Depending on the type of comprehension, the results of *body* may be either collected to create an aggregate (list, vector, string, ...), folded by some operator (sum, product, min, max, ...), or simply discarded.

Each *qualifier* specifies how to repeat the following *qualifiers* and *body*. A *qualifier* can be a generational qualifier that yields a set of values to loop over, or a control qualifier that specify a condition to exclude some values. See the Qualifiers heading below.

A few comprehensions takes extra values before *qualifiers* or after *body*.

`do-ec` *qualifier ... body* [Macro]
[SRFI-42] {`srfi-42`} Repeats *body*. The results of *body* is discarded. This is for side-effecting operations.

`list-ec` *qualifier ... body* [Macro]
[SRFI-42] {`srfi-42`} Repeats *body* and collects the results into a list.

`append-ec` *qualifier ... body* [Macro]
[SRFI-42] {`srfi-42`} Repeats *body*, which must yield a list. Returns a list which is the concatenation of all lists returned by *body*.

`string-ec` *qualifier ... body* [Macro]
`string-append-ec` *qualifier ... body* [Macro]
[SRFI-42] {`srfi-42`} Repeats *body*, which must yield a character (in `string-ec`) or a string (in `string-append-ec`). Returns a string that consists of the results of *body*.

`vector-ec` *qualifier ... body* [Macro]
[SRFI-42] {`srfi-42`} Repeats *body* and collects the results into a vector.

`vector-of-length-ec` *k qualifier ... body* [Macro]
[SRFI-42] {`srfi-42`} This is like `vector-ec`, except that the length of the result vector is known to be *k*. It can be more efficient than `vector-ec`. Unless the comprehension repeats exactly `k` times, an error is signaled.

`sum-ec` *qualifier ... body* [Macro]
`product-ec` *qualifier ... body* [Macro]
[SRFI-42] {`srfi-42`} *body* must yield a numeric value. Returns sum of and product of the results, respectively.

`min-ec` *qualifier ... body* [Macro]
`max-ec` *qualifier ... body* [Macro]
[SRFI-42] {`srfi-42`} *body* must yield a numeric value. Returns maximum and minimum value of the results, respectively. *body* must be evaluated at least once, or an error is signaled.

`any?-ec` *qualifier . . . test* [Macro]
`every?-ec` *qualifier . . . test* [Macro]

[SRFI-42] {`srfi-42`} Evaluates *test* for each iteration, and returns `#t` as soon as it yields non-`#f` (for `any-ec?`), or returns `#f` as soon as it yields `#f` (for `every?-ec`). Unlink the comprehensions introduced above, these stop evaluating *test* as soon as the condition meets. If the qualifiers makes no iteration, `#f` and `#t` are returned, respectively.

`first-ec` *default qualifier . . . body* [Macro]
`last-ec` *default qualifier . . . body* [Macro]

[SRFI-42] {`srfi-42`} First initializes the result by the value of the expression *default*, then start iteration, and returns the value of the first and last evaluation of *body*, respectively. In fact, `first-ec` only evaluates *body* at most once.

These procedures are most useful when used with control qualifiers. For example, the following `first-ec` returns the *first* set of distinct integers (x, y, z), where $x*x+y*y+z*z$ becomes a square of another integer w.

```
(first-ec #f (:integers w) (: z 1 w) (: y 1 z) (: x 1 y)
          (if (= (* w w) (+ (* x x) (* y y) (* z z))))
          (list x y z w))
```

Note that the first qualifier, `(:integers w)`, generates infinite number of integers; if you use `list-ec` instead of `first-ec` it won't stop.

`fold-ec` *seed qualifier . . . expr proc* [Macro]
`fold3-ec` *seed qualifier . . . expr init proc* [Macro]

[SRFI-42] {`srfi-42`} Reduces the values produced by *expr*.

Suppose *expr* produces a sequence of values *x0*, *x1*, . . ., *xN*. `Fold-ec` calculates the following value:

```
(proc xN (...(proc x1 (proc x0 seed))...))
```

It's similar to `fold`, except that *proc* is evaluated within the scope of *qualifier* . . . so you can refer to the variables introduced by them. On the other hand, *seed* is outside of the scope of *qualifier*s.

`Fold-ec3` is almost the same but the initial value calculation. In `fold-ec3`, *seed* is only used when *qualifier*s makes no iteration. Otherwise it calculates the following value:

```
(proc xN (...(proc x1 (init x0))...))
```

Qualifiers

Generational qualifiers

This type of qualifiers generates (possibly infinite) values over which the rest of clauses iterate.

In the following descriptions, *vars* refers to either a single identifier, or a series of identifier and a form `(index identifier2)`. The single identifier in the former case and the first identifier in the latter case name the variable to which each generated value is bound. The *identifier2* in the latter case names a variable to which a series of integers, increasing with each generated element, is bound. See the following example:

```
(list-ec (: x '(a b c)) x)
  ⇒ (a b c)
(list-ec (: x (index y) '(a b c)) (cons x y))
  ⇒ ((a . 0) (b . 1) (c . 2))
```

`:` *vars arg1 args . . .* [EC Qualifier]

A generic dispatcher of generational qualifiers. An appropriate generational qualifier is selected based on the types of *arg1 args* . . . .

`:list` *vars arg1 args ...* [EC Qualifier]
`:vector` *vars arg1 args ...* [EC Qualifier]
`:uvector` *vars arg1 args ...* [EC Qualifier]
`:string` *vars arg1 args ...* [EC Qualifier]

Arg1 args ... should be all lists, vectors, uniform vectors or strings, respectively. Repeats the subsequent clauses while binding each element from those args bound to *vars*. (The `:uvector` qualifier is Gauche's extension.)

```
(list-ec (:string c "ab" "cd") c) ⇒ (#\a #\b #\c #\d)
```

If the arguments given to the generic qualifier `:` are all lists, vectors, uniform vectors or strings, then these qualifiers are used.

`:integers` *vars* [EC Qualifier]

Generates infinite series of increasing integers, starting from 0.

`:range` *vars stop* [EC Qualifier]
`:range` *vars start stop* [EC Qualifier]
`:range` *vars start stop step* [EC Qualifier]

Generates a series of exact integers, starting from *start* (defaults to 0) and stops below *stop*, stepping by *step* (defaults to 1). Giving a negative integer to *step* makes a decreasing series.

```
(list-ec (:range v 5) v)       ⇒ (0 1 2 3 4)
(list-ec (:range v 3 8) v)     ⇒ (3 4 5 6 7)
(list-ec (:range v 1 8 2) v)   ⇒ (1 3 5 7)
(list-ec (:range v 8 1 -2) v)  ⇒ (8 6 4 2)
```

If one, two or three exact integer(s) is/are given to the generic qualifier `:`, this qualifier is used.

`:real-range` *vars stop* [EC Qualifier]
`:real-range` *vars start stop* [EC Qualifier]
`:real-range` *vars start stop step* [EC Qualifier]

Generates a series of real numbers, starting from *start* (defaults to 0) and stops below *stop*, stepping by *step* (defaults to 1). If all the arguments are exact numbers, the result consists of exact numbers; if any of the arguments are inexact, the result consists of inexact numbers.

```
(list-ec (:real-range v 5.0) v)
  ⇒ (0.0 1.0 2.0 3.0 4.0)
(list-ec (:real-range v 1 4 1/3) v)
  ⇒ (1 4/3 5/3 2 7/3 8/3 3 10/3 11/3)
(list-ec (:real-range v 1 5.0 1/2) v)
  ⇒ (1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5)
```

If one, two or three real numbers is/are given to the generic qualifier `:`, and any one of them isn't an exact integer, then this qualifier is used.

`:char-range` *vars min max* [EC Qualifier]

Generates a series of characters, starting from *min* and ending at *max* (inclusive). The characters are enumerated in the order defined by `char<=?` (see Section 6.10 [Characters], page 143).

```
(list-ec (:char-range v #\a #\e) v)
  ⇒ (#\a #\b #\c #\d #\e)
```

If two characters are given to the generic qualifier `:`, this qualifier is used.

`:port` *vars port* [EC Qualifier]
`:port` *vars port read-proc* [EC Qualifier]

Generates a series of values read from an input port *port*, by the procedure *read-proc* (defaults to `read`). The series terminates when EOF is read.

```
(call-with-input-string "a \"b\" :c"
  (^p (list-ec (:port v p) v)))
  ⇒ (a "b" :c)
```

If one or two arguments are given to the generic qualifier : and the first one is an input port, then this qualifier is used.

`:generator` *vars gen* [EC Qualifier]

This is Gauche's extension and not defined in SRFI-42. *gen* must be a procedure with zero arguments. This qualifier repeats until *gen* returns EOF.

Gauche has a set of utilities to create and operate on such procedures; see Section 9.11 [Generators], page 372.

```
(use gauche.generator)
(list-ec (:generator v (grange 1 8)) v)
  ⇒ (1 2 3 4 5 6 7)
```

If one argument is given to the generic qualifier : and it is applicable without arguments, then this qualifier is used.

`:parallel` *generator . . .* [EC Qualifier]

This is used to run through multiple generators in parallel. It terminates when any one of *generator* is exhausted.

```
(list-ec (:parallel (: x '(a b c))
                    (: y "defg"))
  (cons x y))
 ⇒ ((a . #\d) (b . #\e) (c . #\f))

;; Compare with this:
(list-ec (: x '(a b c))
         (: y "defg")
  (cons x y))
 ⇒ ((a . #\d) (a . #\e) (a . #\f) (a . #\g)
    (b . #\d) (b . #\e) (b . #\f) (b . #\g)
    (c . #\d) (c . #\e) (c . #\f) (c . #\g))
```

`:let` *vars expr* [EC Qualifier]

`:while` *generator expr* [EC Qualifier]

`:until` *generator expr* [EC Qualifier]

`:dispatched` *vars dispatch arg1 args . . .* [EC Qualifier]

`:do` (*lb . . .*) *ne1?* (*ls . . .*) [EC Qualifier]
`:do` (*let* (*ob . . .*) *oc . . .*) (*lb . . .*) *ne1?* (*let* (*ib . . .*) *ic . . .*) *ne2?* (*ls . . .*) [EC Qualifier]

Control qualifiers

`if` *test* [EC Qualifier]

`not` *test* [EC Qualifier]
`and` *test . . .* [EC Qualifier]
`or` *test . . .* [EC Qualifier]

`begin` *command . . . expr* [EC Qualifier]

`nested` *qualifier . . .* [EC Qualifier]

11.13 srfi-43 - Vector library (legacy)

`srfi-43` [Module]

This module is effectively superseded by R7RS and `srfi-133`. There are a few procedures that are not compatible with R7RS and `srfi-133`, and this module remains to support legacy code that depends on them.

See Section 6.14 [Vectors], page 174, and see Section 11.30 [Vector library], page 632, for the "modern" versions of vector library. New code should use them.

The following procedures in srfi-43 are built-in. See Section 6.14 [Vectors], page 174, for the description.

```
make-vector     vector          vector?         vector-ref
vector-set!     vector-length   vector-fill!    vector-copy
vector-copy!    vector-append   vector->list    list->vector
reverse-list->vector
```

The following procedures in srfi-43 are supported by srfi-133. See Section 11.30 [Vector library], page 632, for the description.

```
vector-unfold          vector-unfold-right   vector-reverse-copy
vector-reverse-copy!   vector-concatenate    vector-empty?
vector=                vector-index          vector-index-right
vector-skip            vector-skip-right     vector-binary-search
vector-any             vector-every          vector-swap!
reverse-vector->list
```

We explain the procedures that are not listed above.

`vector-fold` *kons knil vec1 vec2 . . .* [Function]
`vector-fold-right` *kons knil vec1 vec2 . . .* [Function]

[SRFI-43] {`srfi-43`} Like `vector-fold` and `vector-fold-right` in `srfi-133`, but *kons* takes an extra argument, the current index, as its first argument. So *kons* must accept *n*+2 arguments, where *n* is the number of given vectors. It is called as `(kons <index> <cumulated-value> <elt1> <elt2> ...)`.

Gauche has `fold-with-index` (see Section 9.29.3 [Mapping over sequences], page 442) that can be used to fold vectors with index, but the argument order of *kons* is slightly different: It passes the index, each element from argument vectors, then cumulated values.

```
(use srfi-43)
(vector-fold list '() '#(a b c) '#(d e f))
  ⇒ (2 (1 (0 () a d) b e) c f)

(use gauche.sequence)
(fold-with-index list '() '#(a b c) '#(d e f))
  ⇒ (2 c f (1 b e (0 a d ())))
```

`vector-map` *f vec1 vec2 . . .* [Function]
`vector-map!` *f vec1 vec2 . . .* [Function]
`vector-for-each` *f vec1 vec2 . . .* [Function]
`vector-count` *f vec1 vec2 . . .* [Function]

[SRFI-43] {`srfi-43`} Like `vector-map` and `vector-for-each` of R7RS, and `vector-map!` and `vector-count` in `srfi-133`, except *f* takes an extra argument, the current index, as its first argument.

Gauche provides `vector-map-with-index`, `vector-map-with-index!` and `vector-for-each-with-index` which are the same as `srfi-43`'s `vector-map`, `vector-map!` and `vector-for-each`, respectively. See Section 6.14 [Vectors], page 174.

```
(vector-map list '#(a b c))
 ⇒ #((0 a) (1 b) (2 c))
(vector-map list '#(a b c) '#(d e f g))
 ⇒ #((0 a d) (1 b e) (2 c f))
(vector-count = '#(0 2 2 4 4))
 ⇒ 3
```

(Note: The `vector-count` example calls `=` with two arguments, the current index and the element, for each element of the input vector. So it counts the number of occasions when the element is equal to the index.)

The generic `map` and `for-each` in `gauche.collection` can be used on vectors, but the mapped procedure is called without index, and the result is returned as a list. `(vector-map f vec1 vec2 ...)` is operationally equivalent to `(map-to-with-index <vector> f vec1 vec2 ...)`. See Section 9.5 [Collection framework], page 344, and Section 9.29 [Sequence framework], page 441.

11.14 `srfi-55` - Requiring extensions

`srfi-55` [Module]

This module defines `require-extension` macro, a yet another way to write portable scripts. See Section 4.12 [Feature conditional], page 68, and Section 11.4 [Feature-based program configuration language], page 590, for other means of ensuring specific features.

This module is autoloaded when you use `require-extension`, so you don't need explicitly say `(use srfi-55)`; for portable scripts, you shouldn't.

`require-extension` *clause . . .* [Macro]

{`srfi-55`} Make extension(s) specified by *clause*s available in the rest of the program.

A *clause* takes the following form:

```
(extension-id extension-arg ...)
```

Currently, only `srfi` is supported as *extension-id*, and its arguments are SRFI numbers.

For example, the following form:

```
(require-extension (srfi 1 13 14))
```

Roughly corresponds to Gauche's `use` forms:

```
(use srfi-1)
(use srfi-13)
(use srfi-14)
```

11.15 `srfi-60` - Integers as bits

`srfi-60` [Module]

This srfi provides bit operations on integers, regarding them as 2's complement representation. It is compatible to SLIB's `logical` module.

The newer srfi-151 (see Section 11.34 [Bitwise operations], page 633) provides the same functionality and more, with more consistent naming. We recommend new code to use srfi-151, while we keep srfi-60 for the backward compatibility.

The following procedures are Gauche built-in. See Section 6.3.6 [Basic bitwise operations], page 121, for the description.

```
lognot          logand          logior          logxor
logtest         logcount        integer-length  logbit?
copy-bit        bit-field       copy-bit-field  ash
```

The following procedures are defined in srfi-151. See Section 11.34 [Bitwise operations], page 633, for the description.

`bitwise-not`	`bitwise-and`	`bitwise-ior`	`bitwise-xor`
`arithmetic-shift`	`bit-count`	`bitwise-if`	`bit-set?`
`copy-bit`	`first-set-bit`		

We describe procedures that are unique in srfi-60 below.

`bitwise-merge` *mask n0 n1* [Function]

[SRFI-60] {`srfi-60`} Same as `bitwise-if` (see Section 11.34 [Bitwise operations], page 633).

`any-bits-set?` *mask n* [Function]

[SRFI-60] {`srfi-60`} Same as builtin `logtest` (see Section 6.3.6 [Basic bitwise operations], page 121). It is also called `any-bit-set?` in srfi-151 (see Section 11.34 [Bitwise operations], page 633).

`log2-binary-factors` *n* [Function]

[SRFI-60] {`srfi-60`} It is also called as `first-set-bit` in this srfi, which is also in srfi-151 (see Section 11.34 [Bitwise operations], page 633). This is equivalent to Gauche's built-in `twos-exponent-factor` (see Section 6.3.6 [Basic bitwise operations], page 121).

`rotate-bit-field` *n count start end* [Function]

[SRFI-60] {`srfi-60`} This is equivalent to `bit-field-rotate` in srfi-151 (see Section 11.34 [Bitwise operations], page 633).

`reverse-bit-field` *n start end* [Function]

[SRFI-60] {`srfi-60`} This is equivalent to `bit-field-reverse` in srfi-151 (see Section 11.34 [Bitwise operations], page 633).

`integer->list` *n :optional len* [Function]

[SRFI-60] {`srfi-60`} Breaks *n* to each bits, representing `1` as `#t` and `0` as `#f`, LSB last, and returns a list of them. If a nonnegative integer *len* is given, it specifies the length of the result. If it is omitted, `(integer-length n)` is used.

```
(integer->list 10)   ⇒ (#t #f #t #f)
(integer->list 10 6) ⇒ (#f #f #t #f #t #f)
```

Srfi-151 has similar procedure `bits->list`, with a reversed bit order (LSB first) (see Section 11.34 [Bitwise operations], page 633).

`list->integer` *lis* [Function]

[SRFI-60] {`srfi-60`} Takes a list of boolean values, replaces the true value for `1` and the false value for `0`, and compose an integer regarding each value as a binary digit. If *n* is nonnegative integer, `(eqv? (list->integer (integer->list n)) n)` is true.

```
(list->integer '(#f #t #f #t #f)) ⇒ 10
```

Srfi-151 has similar procedure `list->bits`, with a reversed bit order (LSB first) (see Section 11.34 [Bitwise operations], page 633).

`booleans->integer` *bool . . .* [Function]

[SRFI-60] {`srfi-60`} ≡ `(list->integer (list bool ...))`

Srfi-151 has similar procedure `bits`, with a reversed bit order (LSB first) (see Section 11.34 [Bitwise operations], page 633).

11.16 `srfi-66` - Octet vectors

`srfi-66` [Module]

This module defines procedures to deal with `u8vector`s; they are almost a subset of `srfi-160` and `gauche.uvector` (see Section 9.36 [Uniform vectors], page 476, except one procedure, `u8vector-copy!`, which has different argument orders (unfortunate historical artifacts).

There's no reason to use this srfi except porting code that relies on `srfi-66`.

The following procedures are the same as `gauche.uvector`:

```
u8vector?        make-u8vector      u8vector
u8vector->list   list->u8vector
u8vector-length  u8vector-ref       u8vector-set!
u8vector=?       u8vector-compare   u8vector-copy
```

`u8vector-copy!` *src sstart target tstart n* [Function]

[SRFI-66] {`srfi-66`} Copy the content of an u8vector *src*, starting from *sstart* for *n* octets, into an u8vector *target* beginning from *tstart*. The *target* u8vector must be mutable.

Note that `gauche.uvector` has also `u8vector-copy!`, but its argument order is as follows, where *send* is `(+ sstart n)`:

```
(u8vector-copy! target tstart src sstart send)
```

Gauche's argument order is consistent with `vector-copy!` of R7RS, srfi-43 and srfi-133.

We recommend to use `srfi-66` only for porting third-party libraries to avoid confusion.

11.17 `srfi-69` - Basic hash tables

`srfi-69` [Module]

This module has been superseded by R7RS `scheme.hash-table` (see Section 10.3.7 [R7RS hash tables], page 536). New code should use it instead.

This is a thin adaptor on Gauche's built-in hashtables (see Section 6.15 [Hashtables], page 177). This is provided for the compatibility to the portable libraries; the hashtable object created by this module's `make-hash-table` is the same as the one created by Gauche's built-in, and you can pass the table to both APIs.

Here's a summary of difference between srfi-69 and Gauche's built-in hash table API:

- The constructor `make-hash-table`, as well as `alist->hash-table`, takes equality predicate and hash function, instead of a single comparator argument.
- The hash function passed to `srfi-69`'s `make-hash-table` takes two arguments, an object to calculate a hash value, and a positive integer that limits the range of the hash value.
- `Srfi-69`'s primary hash table accessor is `hash-table-ref`, which takes a thunk to be called when the table doesn't have an entry for the given key, while Gauche's `hash-table-get` takes a fallback value for that. `Srfi-69` also has `hash-table-ref/default`, which takes a fallback value like Gauche's `hash-table-get`, but the default value can't be omitted.
- The basic iterator of `srfi-69` is called `hash-table-walk`, which is Gauche's `hash-table-for-each`. The srfi name is chosen to avoid conflict with existing Scheme implementations.

The following procedures are the same as Gauche's built-in ones. See Section 6.15 [Hashtables], page 177, for the details.

```
hash-table?        hash-table-delete!   hash-table-exists?
hash-table-keys    hash-table-values    hash-table-fold
hash-table->alist  hash-table-copy
```

`make-hash-table` *:optional eq-pred hash-proc :rest args* [Function]

[SRFI-69] {`srfi-69`} Creates a new hashtable and returns it. This is the same name as Gauche's built-in procedure, but the arguments are different.

The *eq-pred* argument is an equality predicate; it takes two arguments and returns `#t` if two are the same, and `#f` if not. When omitted, `equal?` is used.

The *hash-proc* argument is a hash function. It takes two arguments: an object to hash, and a positive integer to limit the range of the hash value. (Note that Gauche's native hash functions takes only one argument.) When omitted, Gauche tries to choose appropriate hash function if `eq-pred` is known one (`eq?`, `eqv?`, `equal?`, `string=?` or `string-ci=?`). Otherwise we use Gauche's `hash` procedure, but there's no guarantee that it works appropriately; you should give suitable *hash-proc* if you pass custom *eq-pred*.

The returned hash table is an instance of Gauche's native hash table. You can pass it to Gauche's builtin procedures.

`Srfi-69` allows implementation-specific arguments *args* to be passed to `make-hash-table`. At this moment, Gauche ignores them.

`alist->hash-table` *alist :optional eq-pred hash-fn :rest args* [Function]

[SRFI-69] {`srfi-69`} Like Gauche's builtin `alist->hash-table`, but takes *eq-pred* and *hash-fn* separately, instead of a single comparator.

The *alist* argument is a list of pairs. The car of each pair is used for a key, and the cdr for its value.

See `make-hash-table` above for the description of *eq-pred*, *hash-fn* and *args*.

`hash-table-equivalence-function` *ht* [Function]
`hash-table-hash-function` *ht* [Function]

[SRFI-69] {`srfi-69`} Returns equivalence function and hash function of the hashtable *ht*.

The hash function returned from `hash-table-hash-function` takes two arguments, an object to hash and bound, a positive exact integer. Note that the function returned by `hash-table-hash-function` may not be `eq?` to the one you gave to `make-hash-table`.

`hash-table-ref` *ht key :optional thunk* [Function]

[SRFI-69] {`srfi-69`} Looks up the value corresponding to *key* in a hash table *ht*. If there's no entry for *key*, *thunk* is called without arguments. The default of *thunk* is to signal an error.

`hash-table-ref/default` *ht key default* [Function]

[SRFI-69] {`srfi-69`} Looks up the value corresponding to *key* in a hash table *ht*. This is like Gauche's `hash-table-get`, but *default* can't be omitted.

`hash-table-set!` *ht key val* [Function]

[SRFI-69] {`srfi-69`} This is the same as Gauche's `hash-table-put!`.

`hash-table-update!` *ht key proc :optional thunk* [Function]
`hash-table-update!/default` *ht key proc default* [Function]

[SRFI-69] {`srfi-69`}

`hash-table-size` *ht* [Function]

[SRFI-69] {`srfi-69`} Returns the number of entries in a hash table *ht*. The same as Gauche's `hash-table-num-entries`.

`hash-table-walk` *ht proc* [Function]

[SRFI-69] {`srfi-69`} For each entry in a hash table *ht*, calls *proc* with two arguments, a key and its value. It's the same as Gauche's `hash-table-for-each`.

`hash-table-merge!` *ht1 ht2* [Function]
[SRFI-69] {`srfi-69`} Add all entries in a hash table *ht2* into a hash table *ht1*, and returns *ht1*.

`hash` *obj :optional bound* [Function]
[SRFI-69] {`srfi-69`} Like Gauche's `hash`, except this one can take bound argument; if provided, it must be a positive integer, and the return value is limited between 0 and `(- bound 1)`, inclusive.

`string-hash` *obj :optional bound* [Function]
`string-ci-hash` *obj :optional bound* [Function]
[SRFI-69] {`srfi-69`} These are like `srfi-13`'s (see Section 11.5 [String library], page 591), except these don't take *start* and *end* argument.

`hash-by-identity` *obj :optional bound* [Function]
[SRFI-69] {`srfi-69`} This is Gauche's `eq-hash`, except this one can take bound argument.

11.18 `srfi-74` - Octet-addressed binary blocks

`srfi-74` [Module]
This module provides procedures to deal with *blob*, or a sequence of octets. In Gauche, a blob is simply an u8vector.

Most functionalities of this module is available in `binary.io` module (see Section 12.1 [Binary I/O], page 638), and in fact this module is a thin wrapper to it. We provide this module for the compatibility. If you're writing Gauche-specific code, we recommend to use `binary.io` directly.

`endianness` *e* [Macro]
[SRFI-74] {`srfi-74`} The argument *e* must be either `big`, `little`, or `native`. It expands to the implementation-specific endianness designator. In Gauche, the result is one of the symbols; see Section 6.3.7 [Endianness], page 123, for the details.

`make-blob` *size* [Function]
[SRFI-74] {`srfi-74`} Returns a freshly created blob that can hold *size* octets. In Gauche, this is the same as `(make-u8vector size)`.

`blob?` *obj* [Function]
[SRFI-74] {`srfi-74`} Returns `#t` if *obj* is a blob, `#f` otherwise. In Gauche, this is the same as `(u8vector? obj)`.

`blob-length` *blob* [Function]
[SRFI-74] {`srfi-74`} Returns the size of the blob, in octets. In Gauche, this is the same as `(u8vector-length blob)`.

`blob-uint-ref` *size endian blob pos* [Function]
`blob-sint-ref` *size endian blob pos* [Function]
[SRFI-74] {`srfi-74`} Read an unsigned or signed integer of *size* octets beginning at the position of *pos* from *blob*, respectively.

These are wrappers of `(get-uint size blob pos endian)` and `(get-sint size blob pos endian)` in `binary.io` module (see Section 12.1 [Binary I/O], page 638), except that `blob-uint-ref`/`blob-sint-ref` only accept u8vector as *blob*.

`blob-uint-set!` *size endian blob pos val* [Function]
`blob-sint-set!` *size endian blob pos val* [Function]

[SRFI-74] {`srfi-74`} Store an unsigned or signed integer *val* of *size* octets into *blob* starting at the position of *pos*, respectively.

These are wrappers of `(put-uint! size blob pos val endian)` and `(put-sint! size blob pos val endian)` in `binary.io` module (see Section 12.1 [Binary I/O], page 638), except that `blob-uint-set!/blob-sint-set!` only accept u8vector as *blob*.

`blob-u8-ref` *blob pos* [Function]
`blob-u8-set!` *blob pos val* [Function]
`blob-s8-ref` *blob pos* [Function]
`blob-s8-set!` *blob pos val* [Function]

[SRFI-74] {`srfi-74`} Get/set an unsigned or signed integer as a octet at *pos* from/to *blob*.

These are wrappers of `get-u8`, `put-u8!`, `get-s8` and `put-s8!` in `binary.io`, respectively.

`blob-u16-ref` *endian blob pos* [Function]
`blob-u16-set!` *endian blob pos val* [Function]
`blob-s16-ref` *endian blob pos* [Function]
`blob-s16-set!` *endian blob pos val* [Function]
`blob-u32-ref` *endian blob pos* [Function]
`blob-u32-set!` *endian blob pos val* [Function]
`blob-s32-ref` *endian blob pos* [Function]
`blob-s32-set!` *endian blob pos val* [Function]
`blob-u64-ref` *endian blob pos* [Function]
`blob-u64-set!` *endian blob pos val* [Function]
`blob-s64-ref` *endian blob pos* [Function]
`blob-s64-set!` *endian blob pos val* [Function]

[SRFI-74] {`srfi-74`} Get/set an unsigned or signed integer of the indicated length at *pos* from/to *blob*, using the specified *endian*.

These are wrappers of corresponding `get-XX` and `put-XX!` in `binary.io`; note that the argument orders differ, though.

`blob-u16-native-ref` *blob pos* [Function]
`blob-u16-native-set!` *blob pos val* [Function]
`blob-s16-native-ref` *blob pos* [Function]
`blob-s16-native-set!` *blob pos val* [Function]
`blob-u32-native-ref` *blob pos* [Function]
`blob-u32-native-set!` *blob pos val* [Function]
`blob-s32-native-ref` *blob pos* [Function]
`blob-s32-native-set!` *blob pos val* [Function]
`blob-u64-native-ref` *blob pos* [Function]
`blob-u64-native-set!` *blob pos val* [Function]
`blob-s64-native-ref` *blob pos* [Function]
`blob-s64-native-set!` *blob pos val* [Function]

[SRFI-74] {`srfi-74`} Get/set an unsigned or signed integer of the indicated length at *pos* from/to *blob*, using the native endianness.

These are wrappers of corresponding `get-XX` and `put-XX!` in `binary.io`; note that the argument orders differ, though.

`blob=?` *blob1 blob2* [Function]

[SRFI-74] {`srfi-74`} This is the same as `u8vector=?` in `gauche.uvector`.

`blob-copy!` *src sstart target tstart n* [Function]

[SRFI-74] {`srfi-74`} Copy *n* octets from the source blob *src* starting from *sstart* into the target blob *target* starting from *tstart*.

Note that the order of arguments differs from other `*-copy!` procedures (e.g. R7RS's `string-copy!` and `vector-copy!`, and `gauche.uvector`'s `u8vector-copy!`)), which have the following signature: `(*-copy! target tstart src sstart send)`

`blob-copy` *blob* [Function]

[SRFI-74] {`srfi-74`} Returns a fresh copy of *blob*. The same as `u8vector-copy` in `gauche.uvector`.

`blob->u8-list` *blob* [Function]
`u8-list->blob` *list* [Function]

[SRFI-74] {`srfi-74`} Wrappers of `u8vector->list` and `list->u8vector`, except those don't take optional start/end arguments.

`blob->uint-list` *size endian blob* [Function]
`blob->sint-list` *size endian blob* [Function]

[SRFI-74] {`srfi-74`} Read a sequence of unsigned or signed integers of *size* octets from *blob* with *endian*, and returns them as a list.

```
(blob->uint-list 3 (endianness big) '#u8(0 0 1 0 0 2 0 0 3))
  ⇒ (1 2 3)
```

`uint-list->blob` *size endian list* [Function]
`sint-list->blob` *size endian list* [Function]

[SRFI-74] {`srfi-74`} Convert a *list* of unsigned or signed integers to a blob. The resulting blob has `(* size (length list))` octets. Each integer occupies *size* octets.

```
(uint-list->blob 3 (endianness little) '(1 2 3))
  ⇒ #u8(1 0 0 2 0 0 3 0 0)
```

11.19 `srfi-98` - Accessing environment variables

`srfi-98` [Module]

This srfi defines a portable way to access the underlying system's environment variables. Gauche supports such procedures built-in (see Section 6.25.3 [Environment inquiry], page 248), but portable programs may want to use srfi API instead.

`get-environment-variable` *name* [Function]

[SRFI-98] {`srfi-98`} Returns a string value of an environment variable named by a string *name*. If the named environment variable doesn't exist, `#f` is returned.

This is equivalent to `sys-getenv`.

```
(get-environment-variable "PATH")
  ⇒ "/bin:/usr/sbin:/usr/bin"
```

`get-environment-variables` [Function]

[SRFI-98] {`srfi-98`} Returns an assoc list of the name and the value of each environment variable.

This is equivalent to `sys-environ->alist` without the optional argument.

```
(get-environment-variables)
  ⇒ (("PATH" . "/bin:/usr/sbin:/usr/bin")
      ...)
```

11.20 srfi-106 - Basic socket interface

`srfi-106` [Module]

A portable basic socket interface.

Although comprehensive network API is provided by `gauche.net` (see Section 9.20 [Networking], page 398), it is Gauche-specific. This srfi provides a small subset of socket operations, but it offers a portable way to create applications that needs simple networking.

Note that some procedures have the same name as the ones in `gauche.net`, but the interface may differ.

A socket object created by this srfi's API is an instance of Gauche's `<socket>`, so it can be passed to the API in `gauche.net` and vice versa.

The following procedures are exactly the same as defined in `gauche.net`. See Section 9.20 [Networking], page 398, for the details.

```
socket-accept        socket-shutdown      socket-close
socket-input-port    socket-output-port
```

Socket object

`make-client-socket` *node service :optional ai-family ai-socktype ai-flags ai-protocol* [Function]

[SRFI-106] {`srfi-106`} Creates and returns a socket to communicate with the node *node* and *service*. If the socket type is connection-oriented (that is, *ai-socktype* is `*sock-stream*`, which is the default), the returned socket is already connected.

Both *node* and *service* must be strings. The *node* argument is passed to `getaddrinfo(3)` to resolve to the server IP address(es). A service name solely consists of decimal digits is interpreted as a port number.

The default value of optional arguments are as follows: `*af-inet*` for *ai-family*, `*sock-stream*` for *ai-socktype*, `(socket-merge-flags *ai-v4mapped* *ai-addrconfig*)` for *ai-flags*, and `*ipproto-ip*` for *ai-protocol*. See below for valid flag values.

This API differs from `make-client-socket` in `gauche.net`.

```
(make-client-socket "127.0.0.1" "80")
 ⇒ a <socket> connected to port 80 of localhost
```

`make-server-socket` *service :optional ai-family ai-socktype ai-protocol* [Function]

[SRFI-106] {`srfi-106`} Creates and returns a server socket that binds and listens at the port specified by *service*, which must be a string. A service name solely consists of decimal digits is interpreted as a port number.

The default value of optional arguments are as follows: `*af-inet*` for *ai-family*, `*sock-stream*` for *ai-socktype*, and `*ipproto-ip*` for *ai-protocol*. See below for valid flag values.

This API differs from `make-server-socket` in `gauche.net`.

`socket?` *obj* [Function]

[SRFI-106] {`srfi-106`} Equivalent to `(is-a? obj <socket>)`.

Communication

`socket-send` *socket u8vector :optional flags* [Function]

[SRFI-106] {`srfi-106`} Almost same as `socket-send` in `gauche.net`, except that this procedure only accepts a u8vector as the message. (The one in `gauche.net` can take a string as well.)

Returns the number of octets that are actually sent.

`socket-recv` *socket size :optional flags* [Function]

[SRFI-106] {srfi-106} This is like `socket-recv` in `gauche.net`, except that this procedure returns the received data in u8vector, instead of a string. If the peer has shut down the connection, this procedure returns an empty u8vector, `#u8()`.

The *size* argument specifies the maximum size of the receiving data. The returned vector may be shorter if that much data is received.

Flags

The srfi provides common names for constants of typical socket flags, as well as macros that map symbolic name(s) to the flags.

`socket-merge-flags` *flag ...* [Function]

[SRFI-106] {srfi-106} Merge bitwise flags. This is simply `logior` in Gauche.

`socket-purge-flags` *base-flag flag ...* [Function]

[SRFI-106] {srfi-106} Drop the bitwise flags in *base-flag* that are set in *flag* ....

Address family

`*af-inet*`	`AF_INET`
`*af-inet6*`	`AF_INET6`
`*af-unspec*`	`AF_UNSPEC`

`address-family` *name* [Macro]

{srfi-106} *Name* can be either one of symbols `inet`, `inet6`, or `unspec`, and the macro expands into the value of `*af-inet*`, `*af-inet6*` or `*af-unspec*`, respectively.

If *name* is other object, an error is signaled.

Socket domain

`*sock-stream*`	`SOCK_STREAM`
`*sock-dgram*`	`SOCK_DGRAM`

`socket-domain` *name* [Macro]

{srfi-106} *Name* can be either one of symbols `stream` or `datagram`, and the macro expands into the value of `*sock-stream*` and `*sock-dgram*`, respectively.

If *name* is other object, an error is signaled.

Address info

`*ai-canonname*`	`AI_CANONNAME`
`*ai-numerichost*`	`AI_NUMERICHOST`
`*ai-v4mapped*`	`AI_V4MAPPED`
`*ai-all*`	`AI_ALL`
`*ai-addrconfig*`	`AI_ADDRCONFIG`

`address-info` *name ...* [Macro]

{srfi-106} Maps combination of names `canoname`, `numerichost`, `v4mapped`, `all` and `addrconfig` to the combination of corresponding flags.

An error is signaled if other symbols are passed. (Note: `canoname` for `*ai-canonname*`).

Protocol

`*ipproto-ip*`	`IPPROTO_IP`
`*ipproto-tcp*`	`IPPROTO_TCP`
`*ipproto-udp*`	`IPPROTO_UDP`

`ip-protocol` *name* [Macro]

{`srfi-106`} Maps one of names `ip`, `tcp`, and `udp` to the corresponding flag value. An error is signaled if other symbol is passed.

Message type

`*msg-none*`	0
`*msg-peek*`	`MSG_PEEK`
`*msg-oob*`	`MSG_OOB`
`*msg-waitall*`	`MSG_WAITALL`

`message-type` *name ...* [Macro]

{`srfi-106`} Maps combination of names `none`, `peek`, `oob` and `wait-all` to the combination of corresponding flags.

An error is signaled if other symbols are passed. (Note: `wait-all` for `*msg-waitall*`).

Shutdown method

`*shut-rd*`	`SHUT_RD`
`*shut-wr*`	`SHUT_WR`
`*shut-rdwr*`	`SHUT_RDWR`

`shutdown-method` *name ...* [Macro]

{`srfi-106`} Maps combination of names `read` and `write` to the combination of corresponding flags.

An error is signaled if other symbols are passed.

11.21 `srfi-111` - Boxes

`srfi-111` [Module]

SRFI-111 has become a part of R7RS large. See Section 10.3.12 [R7RS boxes], page 547.

11.22 `srfi-112` - Environment inquiry

`srfi-112` [Module]

This srfi provides a portable way to obtain runtime information.

`implementation-name` [Function]

[SRFI-112] {`srfi-112`} Returns a string `"Gauche"`.

`implementation-version` [Function]

[SRFI-112] {`srfi-112`} Returns a string of Gauche's version. The same as `gauche-version` (see Section 6.25.3 [Environment inquiry], page 248).

`cpu-architecture` [Function]

[SRFI-112] {`srfi-112`} Returns a string of CPU architecture info, such as `"x86_64"`. Same as the `machine` field of the return value of `sys-uname` (see Section 6.25.8 [System inquiry], page 266).

`machine-name` [Function]

[SRFI-112] {`srfi-112`} Returns the host name. Same as the `nodename` field of the return value of `sys-uname`. (see Section 6.25.8 [System inquiry], page 266).

`os-name` [Function]

[SRFI-112] {`srfi-112`} Returns the OS name. Same as the `sysname` field of the return value of `sys-uname`.

`os-version` [Function]

[SRFI-112] {`srfi-112`} Returns the OS version. Same as the `release` field of the return value of `sys-uname`.

Here's an example of output:

```
gosh> (implementation-name)
"Gauche"
gosh> (implementation-version)
"0.9.5"
gosh> (cpu-architecture)
"x86_64"
gosh> (machine-name)
"scherzo"
gosh> (os-name)
"Linux"
gosh> (os-version)
"3.2.0-89-generic"
```

11.23 `srfi-113` - Sets and bags

`srfi-113` [Module]

SRFI-113 has become a part of R7RS large. See Section 10.3.5 [R7RS sets], page 525.

11.24 `srfi-114` - Comparators

`srfi-114` [Module]

This module is provided for the compatibility of code using srfi-114. The new code should use srfi-128, which is fully built-in.

The following procedures are built-in. See Section 6.2.4 [Basic comparators], page 103, for the detailed documentation. Those are also exported from `srfi-114` for the compatibility.

Predicates `comparator?`,

Standard comparators
: `boolean-comparator`, `char-comparator`, `char-ci-comparator`, `string-comparator`, `string-ci-comparator`, `symbol-comparator`, `exact-integer-comparator`, `integer-comparator`, `rational-comparator`, `real-comparator`, `complex-comparator`, `number-comparator`, `pair-comparator`, `list-comparator`, `vector-comparator`, `bytevector-comparator`, `uvector-comparator`

The default comparator
: `default-comparator`

Wrapped equality predicates
: `eq-comparator`, `eqv-comparator`, `equal-comparator`

Accessors `comparator-equality-predicate`, `comparator-comparison-procedure`, `comparator-hash-function`

Primitive applicators
`comparator-test-type`, `comparator-check-type`, `comparator-compare`, `comparator-hash`

Comparison predicates
`=?`, `<?`, `<=?`, `>?`, `>=?`

Basic comparator interface

`make-comparator` *type-test equal compare hash :optional name* [Function]

[SRFI-114+] {`srfi-114`} This is SRFI-114 style comparator constructor. The optional *name* argument is Gauche's extension.

This is the same as built-in `make-comparator/compare`. See Section 6.2.4 [Basic comparators], page 103, for the details.

Do not confuse this with built-in (SRFI-128) `make-comparator`; if you `(use srfi-114)`, this one shadows the built-in one.

Note that a comparator works for both SRFI-114 and SRFI-128 procedures, regardless of how it is constructed.

`comparator-comparison-procedure?` *c* [Function]
`comparator-hash-function?` *c* [Function]

[SRFI-114] {`srfi-114`} Returns true iff a comparator *c* can be used to order objects or to hash them, respectively. These are aliases of built-in `comparator-ordered?` and `comparator-hashable?`.

`comparator-type-test-procedure` *c* [Function]

[SRFI-114] {`srfi-114`} Returns type test predicate of a comparator *c*. This is an alias of built-in `comparator-type-test-predicate`.

`comparator-equal?` *c a b* [Function]

[SRFI-114] {`srfi-114`} Checks equality of *a* and *b* using the equality predicate of a comparator *c*. This can be also written in `=?`, which is built-in (see Section 6.2.4.2 [Comparator predicates and accessors], page 104).

```
(=? c a b)
```

Auxiliary comparator constructors

`make-inexact-real-comparator` *epsilon rounding nan-handling* [Function]

[SRFI-114] {`srfi-114`} Returns a comparator for inexact real numbers, taking into account of errors and NaNs.

The basic idea is that we compare two finite real numbers after rounding them to *epsilon* interval, which must be a nonnegative real number. (Note that it's not to compare two numbers "close enough", as often being done to compare inexact numbers. "Close enough" scheme won't be transitive.)

The rounding mode is specified by the *rounding* argument. It can be either one of the symbols `round`, `ceiling`, `floor` or `truncate`, or a procedure that takes two arguments, a real number and an epsilon, and returns the rounded result of the first argument according to the given epsilon.

The *nan-handling* argument determines how to handle the case to compare NaN and non-NaN numbers. (If both are NaNs, this comparator regards them as equal). It can be either one of the followings:

`min` If it's a symbol `min`, NaN is compared as smaller than all other real numbers, even than `-inf.0`.

`max` If it's a symbol `min`, NaN is compared as greater than all other real numbers, even than `+inf.0`.

`error` If it's a symbol `error`, an error is signaled.

a procedure taking one argument
The procedure is invoked with the real number which is not NaN. If it ever returns, it must return either 1, 0 or -1, for it's used as the result of the comparison procedure of the comparator. However, since the procedure doesn't know *which* argument is non-NaN, it's hard to have consistent semantics; the best bet is to throw a custom error.

```
(define c (make-inexact-real-comparator 0.1 'round 'error))

(comparator-compare c 0.112 0.098) ⇒ 0
(comparator-compare c 0.131 0.172) ⇒ -1
```

Note: Rounding to the nearest epsilon interval would involve scaling inexact numbers, and that may reveal small difference between the actual number and its notation. For example, an inexact real number denoted as `0.15` is actually slightly smaller than `15/100`, and rounding with epsilon `0.1` would result `0.1`, not `0.2`.

`make-car-comparator` *cmpr* [Function]
`make-cdr-comparator` *cmpr* [Function]

[SRFI-114] {`srfi-114`} Returns comparators that accept pairs, and compare them with their car or cdr by *cmpr*, respectively.

Using `make-key-comparator`, these cam be written as follows (see Section 6.2.4.4 [Combining comparators], page 108, for `make-key-comparator`).

```
(define (make-car-comparator cmpr)
  (make-key-comparator cmpr pair? car))

(define (make-cdr-comparator cmpr)
  (make-key-comparator cmpr pair? cdr))
```

`make-list-comparator` *element-comparator* [Function]
`make-vector-comparator` *element-comparator* [Function]
`make-bytevector-comparator` *element-comparator* [Function]

{`srfi-114`} [SRFI-114] Returns a new comparator that compares lists, vectors and bytevectors element-wise using *element-comparator*, respectively. These are more general versions of `list-comparator`, `vector-comparator` and `bytevector-comparator`, which use `default-comparator` as *element-comparator*.

For a list comparator, it is an error to pass improper lists.

Note that comparing sequences of different lengths is slightly different between lists and vector/bytevectors. List comparator uses "dictionary" order, so `(1 3)` comes after `(1 2 3)`, assuming elements are compared numerically. For vectors and bytevectors, shorter one always precedes the other, so `#(1 3)` comes before `#(1 2 3)`.

make-listwise-comparator *type-test element-comparator empty? head tail* [Function]

[SRFI-114] {srfi-114} More general version of make-list-comparator. Returns a comparator that compares structures which can be traversed using three procedures, *empty?*, *head* and *tail*. Each of those procedure receives a structure to be compared, and *empty?* must return #t iff the structure is empty, *head* must return the first element in the structure, and *tail* must return the same type of structure containing all the elements but the head. The *type-test* predicate checks if the arguments passed to the comparator to be a suitable structure.

That is, make-list-comparator can be written in make-listwise-comparator as follows.

```
(make-list-compartator element-comparator)
  ≡
  (make-listwise-comparator list? element-comparator null? car cdr)
```

This can be used to compare list-like structures. For example, the following call returns a comparator that compares elements of two lazy streams (see Section 12.71 [Stream library], page 809).

```
(make-listwise-comparator stream?
                          element-comparator
                          stream-null?
                          stream-car
                          stream-cdr)
```

make-vectorwise-comparator *type-test element-comparator length ref* [Function]

[SRFI-114] {srfi-114} More general version of make-vector-comparator. Returns a comparator that compares structures which can be traversed using two procedures, *length* and *ref*. The *length* procedure must return the number of elements in the structure. The *ref* procedure receives a structure and a nonnegative exact integer index *k*, and must return *k*-th element of the structure.

That is, the following equivalence holds:

```
(make-vector-comparator element-comparator)
  ≡
  (make-vectorwise-comparator vector? element-comparator
                              vector-length vector-ref)

(make-bytevector-comparator element-comparator)
  ≡
  (make-vectorwise-comparator u8vector? element-comparator
                              u8vector-length u8vector-ref)
```

make-pair-comparator *car-comparator cdr-comparator* [Function]

[SRFI-114] {srfi-114} Creates a comparator that compares pairs, with their cars by *car-comparator* and their cdrs by *cdr-comparator*.

make-improper-list-comparator *element-comparator* [Function]

[SRFI-114] {srfi-114} This may be understood as recursive pair comparator; if objects to be compared are pairs, we recurse their cars then their cdrs. If objects to be compared are not pairs, we use *element-comparator* to compare them.

make-selecting-comparator *comparator1 comparator2 . . .* [Function]

[SRFI-114] {srfi-114} This creates a comparator that works any one of the given comparators; the objects to be compared are type-tested with each of the comparators in order, and the first comparator that accepts all objects will be used.

`make-refining-comparator` *comparator1 comparator2 . . .* [Function]

[SRFI-114] {`srfi-114`} This is similar to `make-selecting-comparator`, except that if the first comparator that accepts given objects to compare finds they are equal (or 0 by the comparison procedure), it tries other comparators down the list, if any.

`make-reverse-comparator` *comparator* [Function]

[SRFI-114] {`srfi-114`} Returns a comparator that just reverses the comparison order of *comparator*.

`make-debug-comparator` *comparator* [Function]

[SRFI-114] {`srfi-114`}

Comparison procedure constructors

`make-comparison<` *lt-pred* [Function]
`make-comparison>` *gt-pred* [Function]
`make-comparison<=` *le-pred* [Function]
`make-comparison>=` *ge-pred* [Function]
`make-comparison=/<` *eq-pred lt-pred* [Function]
`make-comparison=/>` *eq-pred gt-pred* [Function]

[SRFI-114] {`srfi-114`} Utility procedures to create a comparison procedure (the one returns -1, 0, or 1) from the given predicate. For example, `make-comparison<` can be defined as follows:

```
(define (make-comparison< pred)
  (^[a b] (cond [(pred a b) -1]
                [(pred b a) 1]
                [else 0])))
```

Comparison syntax

`if3` *expr less equal greater* [Macro]

[SRFI-114] {`srfi-114`} Three-way `if`: Evaluates *expr*, and then evaluates either one of *less*, *equal*, or *greater*, depending on the value of *expr* is either less than zero, equal to zero, or greater than zero, respectively.

`if=?` *expr consequent :optional alternate* [Macro]
`if<?` *expr consequent :optional alternate* [Macro]
`if>?` *expr consequent :optional alternate* [Macro]
`if<=?` *expr consequent :optional alternate* [Macro]
`if>=?` *expr consequent :optional alternate* [Macro]
`if-not=?` *expr consequent :optional alternate* [Macro]

[SRFI-114] {`srfi-114`} Conditional evaluation according to comparison expression *expr*; that is, `if`*OP*`?` evaluates *consequent* if `(`*OP*` expr 0)` is true, otherwise it evaluates *alternate* when provided.

```
(if<? (compare 10 20) 'yes)      ⇒ yes
(if>=? (compare 10 20) 'yes 'no) ⇒ no
```

Comparison predicate constructors

`make=?` *comparator* [Function]
`make<?` *comparator* [Function]
`make>?` *comparator* [Function]
`make<=?` *comparator* [Function]

`make>=?` *comparator* [Function]

[SRFI-114] {`srfi-114`}

```
((make=? comparator) obj1 obj2 obj3 ...)
  ≡ (=? comparator obj1 obj2 obj3 ...)
```

Interval comparison predicates

`in-open-interval?` [*comparator*] *obj1 obj2 obj3* [Function]
`in-closed-interval?` [*comparator*] *obj1 obj2 obj3* [Function]
`in-open-closed-interval?` [*comparator*] *obj1 obj2 obj3* [Function]
`in-closed-open-interval?` [*comparator*] *obj1 obj2 obj3* [Function]

[SRFI-114] {`srfi-114`} Check if *obj1*, *obj2* and *obj3* has the following relationships:

```
(and (op1 obj1 obj2) (op2 obj2 obj3))
```

Where each of *op1* and *op2* can be `(make<? comparator)` (if that end is *open*), or `(make<=? comparator)` (if that end is *closed*).

When *comparator* is omitted, the default comparator is used.

```
(use srfi-42)
(list-ec (: x 0 5) (list x (in-closed-open-interval? 1 x 3)))
  ⇒ ((0 #f) (1 #t) (2 #t) (3 #f) (4 #f))
```

Min/max comparison procedures

`comparator-min` and `comparator-max` are the same as `srfi-162`. See Section 11.39 [Comparator sublibrary], page 636.

11.25 `srfi-115` - Sets and bags

`srfi-115` [Module]

SRFI-115 has become a part of R7RS large. See Section 10.3.16 [R7RS regular expressions], page 552.

11.26 `srfi-117` - Queues based on lists

`srfi-117` [Module]

SRFI-117 has become a part of R7RS large. See Section 10.3.13 [R7RS list queues], page 547.

11.27 `srfi-118` - Simple adjustable-size strings

`srfi-118` [Module]

This SRFI defines two string mutating operations that can change the length of the string: `string-append!` and `string-replace!`.

Note that, in Gauche, the body of strings is immutable; when you mutate a string, Gauche creates a fresh new string body and just switch a pointer in the original string to point the new string body. So it is not a problem to implement this SRFI in Gauche, but it also means you won't get any performance benefit by using these operations. Using immutable counterparts (`string-append` and `string-replace`) gives you the same performance. (Be aware that the interface is slightly different from the immutable versions.)

We provide this module only for the compatibility. Gauche-specific programs should stay away from this module. Particularly, avoid code like the example in SRFI-118 document (build a string by `append!`-ing small chunks at a time)—they're quadratic on Gauche.

`string-append!` *string values . . .* [Function]

[SRFI-118] {`srfi-118`} The *string* argument must be a mutable string. Modify *string* by appending *values*, each of which is either a character or a string.

```
(rlet1 a (string-copy "abc")
  (string-append! a #\X "YZ"))
 ⇒ "abcXYZ"
```

`string-replace!` *dst dst-start dst-end src :optional src-start src-end* [Function]

[SRFI-118] {`srfi-118`} The *dst* argument must be a mutable string. Replace *dst* between *dst-start* (inclusive) and *dst-end* (exclusive) with a string *src*. The optional arguments *src-start* and *src-end* limits the region of *src* to be used.

Be aware that the order of arguments differ from SRFI-13's `string-replace` (see Section 11.5.12 [SRFI-13 other string operations], page 599); `string-replace!` resembles to `string-copy!` (also in SRFI-13), rather than `string-replace`.

```
(rlet1 a (string-copy "abc")
  (string-replace! a 1 2 "XYZ"))
 ⇒ "aXYZc"
```

11.28 `srfi-127` - Lazy sequence (srfi)

`srfi-127` [Module]

SRFI-127 has become a part of R7RS large. See Section 10.3.10 [R7RS lazy sequences], page 544.

11.29 `srfi-132` - Sort library

`srfi-132` [Module]

SRFI-132 has become a part of R7RS large. See Section 10.3.4 [R7RS sort], page 522.

11.30 `srfi-133` - Vector library

`srfi-133` [Module]

SRFI-133 has become a part of R7RS large. See Section 10.3.2 [R7RS vectors], page 517.

11.31 `srfi-141` - Integer division

`srfi-141` [Module]

SRFI-141 has become a part of R7RS large. See Section 10.3.18 [R7RS integer division], page 574.

11.32 `srfi-143` - Fixnums

`srfi-143` [Module]

SRFI-143 has become a part of R7RS large. See Section 10.3.20 [R7RS fixnum], page 579.

11.33 `srfi-146` - Mappings and hashmaps

`srfi-146` [Module]

`srfi-146.hash` [Module]

SRFI-146 has become a part of R7Rs large. See Section 10.3.17 [R7RS mappings], page 563.

11.34 `srfi-151` - Bitwise operations

`srfi-151` [Module]

SRFI-151 has become a part of R7Rs large. See Section 10.3.19 [R7RS bitwise operations], page 575.

11.35 `srfi-152` - String library (reduced)

`srfi-152` [Module]

This is an improved version of `srfi-13`. The spec is actually smaller than srfi-13 (hence 'reduced' in the title) by removing bells and whistles. The consistency with R7RS and recent srfis are also considered.

The following are built-in. See Section 6.12 [Strings], page 153, for the details.

```
string?          make-string      string           string-length
string->vector   string->list     vector->string   list->string
string-ref       string-set!      substring        string-copy
string=?         string<?         string<=?        string>?
string>=?        string-append    string-join      string-split
read-string      write-string     string-fill!
```

The following procedures are defined to use Unicode string case folding, and `gauche.unicode` module provides them. See Section 9.35.3 [Full string case conversion], page 475, for the details. Note that Gauche's built-in versions uses character-wise case folding, which differs from string case folding on some characters (German eszett, for example).

```
string-ci=?  string-ci<?  string-ci<=?  string-ci>?  string-ci>=?
```

The following procedures are defined in srfi-13. See Section 11.5 [String library], page 591, for the details. Note: Some of those procedures in srfi-13 that require predicate allows a char-set or a character to be passed instead (e.g. `string-filter`). In srfi-152, only predicate is allowed. Our srfi-152 implementation shares the same procedure with srfi-13, so they accept the same arguments as srfi-13's, but such code won't be portable as srfi-152.

```
string-null?       string-any           string-every         string-tabulate
string-unfold      string-unfold-right        reverse-list->string
string-take        string-drop          string-take-right    string-drop-right
string-pad         string-pad-right     string-trim          string-trim-right
string-trim-both   string-replace       string-prefix-length string-suffix-length
string-prefix?     string-suffix?       string-index         string-index-right
string-skip        string-skip-right    string-contains      string-concatenate
string-concatenate-reverse              string-fold          string-fold-right
string-count       string-filter        string-copy!
```

The following procedures are the same with the ones in R7RS `scheme.base` module (see Section 10.2.2 [R7RS base library], page 504). Note that srfi-13 defines different procedures with the same name.

```
string-map         string-for-each
```

We describe procedures unique to this module below.

`string-remove` *pred string :optional start end* [Function]

[SRFI-152] {`srfi-152`} Returns a substring of *string* between *start* and *end*, except characters that satisfy *pred*. In other words, it is `(string-filter (complement pred) string start end)`.

This is called *string-delete* in `srfi-13`. Being changed to take only a predicate (but not a character), it is renamed for the consistency of other srfi (e.g. `filter`, `remove` and `delete` in srfi-1.)

```
(string-remove char-whitespace?
               "Quick fox jumps over the lazy dog"
               3 22)
 ⇒ "ckfoxjumpsovert"
```

`string-replicate` *string from to :optional start end* [Function]

[SRFI-152] {`srfi-152`} Extended substring. It is called `xsubstring` in srfi-13, but renamed for the consistency.

Extract a substring of *string* between *start* and *end*, and conceptually create a bidirectional infinite string by repeating the substring to both direction. For example, suppose *string* is `"abcde"`, *start* is 1, and *end* is 4. So we repeat the substring `"bcd"`, with one `b` falling on the index zero:

```
... b  c  d  b  c  d  b  c  d  b  c  d  b ...
   -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6
```

Then we extract a substring between *from* and *to* out of this infinite string.

```
(string-replicate "abcde" 2 10 1 4)
  ⇒ "dbcdbcdb"
(string-replicate "abcde" -5 -3 1 4)
  ⇒ "cdbcdbcd"
```

`string-segment` *string k* [Function]

[SRFI-152] {`srfi-152`} Splits *string* by every *k* characters and returns a list of those strings. The last string may be shorter than *k*.

```
(string-segment "abcdefghijklmn" 3)
 ⇒ ("abc" "def" "ghi" "jkl" "mn")
```

We have a similar procedure on lists, `slices` (see Section 6.6.4 [List accessors and modifiers], page 127).

`string-contains-right` *string1 string2 :optional start1 end1 start2 end2* [Function]

[SRFI-152] {`srfi-152`} Like `string-contains`, looks for a needle *string2* from a haystack *string1*, but if it is found, returns the start index of the *last* match, instead of the first match. The returned index is in *string1*. The optional arguments limit the range of a needle and a haystack. If a needle isn't found, `#f` is returned.

An edge case: If a needle is empty (e.g. *string2* is empty, or *start2* = *end2*), it always matches right after the haystack, so *end1* is returned.

```
(string-contains-right "Little Lisper" "Li")
  ⇒ 7
```

`string-take-while` *string pred :optional start end* [Function]
`string-take-while-right` *string pred :optional start end* [Function]

[SRFI-152] {`srfi-152`} Returns the longest prefix or suffix of *string* in which all characters satisfy *pred*.

Note: The order of *pred* and the source object is different from other `take-while`-style procedures, such as `take-while` (Section 10.3.1 [R7RS lists], page 512), `ideque-take-while` (Section 10.3.8 [R7RS immutable deques], page 540), and `lseq-take-while` (Section 10.3.10 [R7RS lazy sequences], page 544).

string-drop-while *string pred :optional start end* [Function]
string-drop-while-right *string pred :optional start end* [Function]

[SRFI-152] {srfi-152} Returns the longest prefix or suffix of *string* in which all characters does not satisfy *pred*.

Note: The order of *pred* and the source object is different from other `drop-while`-style procedures, such as `drop-while` (Section 10.3.1 [R7RS lists], page 512), `ideque-drop-while` (Section 10.3.8 [R7RS immutable deques], page 540), and `lseq-drop-while` (Section 10.3.10 [R7RS lazy sequences], page 544).

string-span *string pred :optional start end* [Function]
string-break *string pred :optional start end* [Function]

[SRFI-152] {srfi-152} Find the longest prefix of *string* between *start* and *end* in which all characters satisy / do not satisfy *pred*, and returns the prefix and the rest of substring as two values.

```
(string-break "foo@example.com" (cut eqv? <> #\@))
 ⇒ "foo" and "@example.com"

(string-break "foo@example.com" (cut eqv? <> #\@) 1 10)
 ⇒ "oo" "@exampl"

;; This is Gauche specific - a char-set can work as a predicate:
(string-span "VAR_1 = $VAR_2" #[\w])
 ⇒ "VAR_1" and " = $VAR_2"
```

Note: The order of *pred* and the source object is different from `span` and `break` in `scheme.list` (see Section 10.3.1 [R7RS lists], page 512).

11.36 `srfi-154` - First-class dynamic extents

srfi-154 [Module]

This module provides a convenient way to reify the dynamic environment. A continuation captured by `call/cc` includes the dynamic environment, as well as the control flow. Sometimes you only want the dynamic environment part. A dynamic extent is a reified dynamic environment.

Let's see an example. You want a procedure that prints out a message to the error port, so you wrote this `print-error`:

```
(define print-error
  (lambda (msg) (display msg (current-error-port))))
```

However, if `print-error` is called while the current error port is altered, it is affected. It is supposed to be so, that's the point of `current-error-port`.

```
(call-with-output-string
  (^p (with-error-to-port p (^[] (print-error "abc\n")))))
 ⇒ "abc\n"
```

If you do want `print-error` to use the error port at the time it is defined, you have to extract the dynamic value at the moment.

```
(define print-error
  (let1 eport (current-error-port)
    (lambda (msg) (display msg eport))))
```

This would be quickly cumbersome when you need to capture multiple dynamic values, or the original `print-error` is called indirectly and you can't modify it the way shown above.

Using `dynamic-lambda` addresses this issue. It not only captures the lexical environment, but also the dynamic environment when it is evaluated. So the `current-error-port` in its body returns the current error port at the time of `print-error` being defined, not when it is called:

```
(define print-error
  (dynamic-lambda (msg) (display msg (current-error-port))))
```

`current-dynamic-extent` [Function]

[SRFI-154] {`srfi-154`} Returns a dynamic-extent object that reifies the current dynamic environment.

`dynamic-extent?` *obj* [Function]

[SRFI-154] {`srfi-154`} Returns `#t` iff *obj* is a dynamic-extent object.

`with-dynamic-extent` *dynext thunk* [Function]

[SRFI-154] {`srfi-154`} Calls the *thunk* in the dynamic extent *dynext*, and returns the values yielded by *thunk*.

`dynamic-lambda` *formals body . . .* [Macro]

[SRFI-154] {`srfi-154`} Like `lambda`, but this not only captures the lexical environment, but also the dynamic environment. Yields a procedure.

Note: Since `dynamic-lambda` needs to swap the dynamic environment after executing *body*, the last expression of *body* isn't called in the tail context even if the procedure created by `dynamic-lambda` is called in tail context.

11.37 `srfi-158` - Generators and accumulators

`srfi-158` [Module]

SRFI-158 has become a part of R7RS large. See Section 10.3.9 [R7RS generators], page 542,

11.38 `srfi-160` - Homogeneous numeric vector libraries

`srfi-160` [Module]

This is an enhancement of srfi-4, Homogeneous vectors (see Section 11.2 [Homogeneous vectors], page 590), and then become a part of R7RS-large (see Section 10.3.3 [R7RS uniform vectors], page 522).

In Gauche, all the procedures in this module are provided as a part of `gauche.uvector` module (see Section 9.36 [Uniform vectors], page 476, and described there.

This module only exports the procedures defined in `srfi-160`.

11.39 `srfi-162` - Comparator sublibrary

`srfi-162` [Module]

This is a supplement of srfi-128, comparators. It provides a few comparator procedures, as well as several useful pre-defined comparators, listed below. Theese pre-defined comparators are already built in Gauche, so see Section 6.2.4.3 [Predefined comparators], page 106, for the details.

```
default-comparator    boolean-comparator    real-comparator
char-comparator       char-ci-comparator    string-comparator
string-ci-comparator  pair-comparator       list-comparator
vector-comparator     eq-comparator         eqv-comparator
equal-comparator
```

`comparator-min` *comparator obj obj2 . . .* [Function]
`comparator-max` *comparator obj obj2 . . .* [Function]

[SRFI-162] {`srfi-162`} Find the object in *obj1 obj2 . . .* that is minimum or maximum compared by *comparator*. Note: Srfi-114 provides the same procedures.

```
(comparator-min list-comparator '(a c b) '(a d) '(a c))
  ⇒ (a c)
```

`comparator-min-in-list` *comparator list* [Function]
`comparator-min-in-list` *comparator list* [Function]

[SRFI-162] {`srfi-162`} Find the object in *list* that is minimum or maximum compared by *comparator*. It is an error if *list* is empty.

11.40 `srfi-173` - Hooks (srfi)

`srfi-173` [Module]

This module provides hooks, which manages a list of closures to be called.

It is based on Guile's hooks, which Gauche supports in `gauche.hook` module (see Section 9.12 [Hooks], page 383). This srfi is a thin layer on top of `gauche.hook`.

The following procedures are the same as `gauche.hook`:

```
make-hook      hook?      hook->list
```

`hook-add!` *hook proc* [Function]

[SRFI-173] {`srfi-173`} Add a procedure *proc* to a hook *hook*. The procedure must accept the number of arguments that matches the arity of the hook. It is the same as `(add-hook! hook proc)` of `gauche.hook`.

`hook-delete!` *hook proc* [Function]

[SRFI-173] {`srfi-173`} Delete *proc* from *hook*. If *proc* isn't in *hook*, do nothing. It is the same as `(delete-hook! hook proc)` of `gauche.hook`.

`hook-reset!` *hook* [Function]

[SRFI-173] {`srfi-173`} Remove all the procedures registered in `hook`. It is the same as `(reset-hook! hook)` of `gauche.hook`.

`hook-run` *hook args . . .* [Function]

[SRFI-173] {`srfi-173`} Apply all the procedures from *hook* to the *args*. The order of the procedure isn't specified in the srfi, though Gauche preserves the order (see `run-hook` (see Section 9.12 [Hooks], page 383).

`list->hook` *arity list* [Function]

[SRFI-173] {`srfi-173`} Creates a new hook with specified *arity* (which is a non-negative exact integer), which has the procedures in *list*. All the procedures must accept *arity* number of arguments.

`list->hook!` *hook list* [Function]

[SRFI-173] {`srfi-173`} Replace the list of procedures in *hook* with procedures in *list*. All the procedures must accept the number of arguments the same as *hook*'s arity.

12 Library modules - Utilities

12.1 `binary.io` - Binary I/O

`binary.io` [Module]

This module provides basic procedures to perform binary I/O of numeric data. Each datum can be read from or written to a port, and got from or put to a uniform vector (see Section 9.36 [Uniform vectors], page 476). For structured binary data I/O, more convenient `pack` utility is implemented on top of this module (see Section 12.2 [Packing binary data], page 641). You might want to use this module directly if you need speed or want a flexible control of endianness.

See also Section 9.36 [Uniform vectors], page 476, which provides binary block I/O.

Endianness

Most procedures of this module take an optional *endian* argument, specifying the byte order of the binary input. It must be either one of symbols `big-endian`, `little-endian`, or `arm-little-endian`. If the endian argument is omitted, the current value of the builtin parameter `default-endian` is used (see Section 6.3.7 [Endianness], page 123). (For 8-bit I/O procedures like `read-u8` the endian argument has no effect, but is accepted for consistency).

I/O using port

`read-u8` *:optional port endian* [Function]
`read-u16` *:optional port endian* [Function]
`read-u32` *:optional port endian* [Function]
`read-u64` *:optional port endian* [Function]

{`binary.io`} Reads 8, 16, 32 or 64 bit unsigned integer from *port* with specified endian, respectively. If *port* is omitted, current input port is used. If *port* reaches EOF before a complete integer is read, EOF is returned.

`read-s8` *:optional port endian* [Function]
`read-s16` *:optional port endian* [Function]
`read-s32` *:optional port endian* [Function]
`read-s64` *:optional port endian* [Function]

{`binary.io`} Reads 8, 16, 32 or 64 bit 2's complement signed integer from *port* with specified endian, respectively. If *port* is omitted, current input port is used. If *port* reaches EOF before a complete integer is read, EOF is returned.

`read-uint` *size :optional port endian* [Function]
`read-sint` *size :optional port endian* [Function]

{`binary.io`} More flexible version. Reads *size*-octet unsigned or signed integer from *port* with specified endian. If *port* reaches EOF before a complete integer is read, EOF is returned.

`read-ber-integer` *:optional port* [Function]

{`binary.io`} Reads BER compressed integer a la X.209. A BER compressed integer is an unsigned integer in base 128, most significant digit first, where the high bit is set on all but the final (least significant) byte.

`write-u8` *val :optional port endian* [Function]
`write-u16` *val :optional port endian* [Function]
`write-u32` *val :optional port endian* [Function]

`write-u64` *val :optional port endian* [Function]
{`binary.io`} Writes a nonnegative integer *val* as 8, 16, 32 or 64 bit unsigned integer to *port* with specified endian, respectively. *Val* must be within the range of integers representable by the specified bits. When *port* is omitted, current output port is used.

`write-s8` *val :optional port endian* [Function]
`write-s16` *val :optional port endian* [Function]
`write-s32` *val :optional port endian* [Function]
`write-s64` *val :optional port endian* [Function]
{`binary.io`} Writes an integer *val* as 8, 16, 32 or 64 bit as 2's complement signed integer to *port* with specified endian, respectively. *Val* must be within the range of integers representable by the specified bits. When *port* is omitted, current output port is used.

`write-uint` *size val :optional port endian* [Function]
`write-sint` *size val :optional port endian* [Function]
{`binary.io`} More flexible version. Writes an integer *val* as unsigned or signed integer of *size* bytes to *port* with specified endian. When *port* is omitted, current output port is used.

`write-ber-integer` *val :optional port* [Function]
{`binary.io`} Writes a nonnegative integer *val* in BER compressed integer to *port*. See `read-ber-integer` above for BER format.

`read-f16` *:optional port endian* [Function]
`read-f32` *:optional port endian* [Function]
`read-f64` *:optional port endian* [Function]
{`binary.io`} Reads 16, 32, or 64-bit floating point numbers, respectively. 32bit is IEEE754 single-precision, and 64bit is IEEE754 double-precision numbers. 16-bit floating point number consists of 1-bit sign, 5-bit exponent and 10-bit mantissa, as used in some HDR image format.

If *port* is omitted, current input port is used. If *port* reaches EOF before a complete number is read, EOF is returned.

`write-f16` *val :optional port endian* [Function]
`write-f32` *val :optional port endian* [Function]
`write-f64` *val :optional port endian* [Function]
{`binary.io`} Writes a real number *val* to *port* in 16, 32, or 64-bit floating point number, respectively. If *port* is omitted, current output port is used.

I/O using uniform vectors

In the following routines, the argument *uv* can be any type of uniform vector; if it is not a `u8vector`, it is treated as if `(uvector-alias <u8vector> uv)` is called—that is, it reads directly from the memory image that holds the uvector's content. The *pos* argument specifies the byte position from the beginning of the memory area (it is always byte position, regardless of the uniform vector's element size).

`get-u8` *uv pos :optional endian* [Function]
`get-u16` *uv pos :optional endian* [Function]
`get-u32` *uv pos :optional endian* [Function]
`get-u64` *uv pos :optional endian* [Function]
`get-s8` *uv pos :optional endian* [Function]
`get-s16` *uv pos :optional endian* [Function]
`get-s32` *uv pos :optional endian* [Function]
`get-s64` *uv pos :optional endian* [Function]
`get-f16` *uv pos :optional endian* [Function]

`get-f32` *uv pos :optional endian* [Function]
`get-f64` *uv pos :optional endian* [Function]

{`binary.io`} Reads a number of a specific format from a uniform vector *uv*, starting at a byte position *pos*. An error is signaled if the specified position makes reference outside of the uniform vector's content. Returns the read number.

`get-u16be` *uv pos* [Function]
`get-u16le` *uv pos* [Function]
`get-u32be` *uv pos* [Function]
`get-u32le` *uv pos* [Function]
`get-u64be` *uv pos* [Function]
`get-u64le` *uv pos* [Function]
`get-s16be` *uv pos* [Function]
`get-s16le` *uv pos* [Function]
`get-s32be` *uv pos* [Function]
`get-s32le` *uv pos* [Function]
`get-s64be` *uv pos* [Function]
`get-s64le` *uv pos* [Function]
`get-f16be` *uv pos* [Function]
`get-f16le` *uv pos* [Function]
`get-f32be` *uv pos* [Function]
`get-f32le` *uv pos* [Function]
`get-f64be` *uv pos* [Function]
`get-f64le` *uv pos* [Function]

{`binary.io`} These are big-endian (`be`) or little-endian (`le`) specific versions of `get-*` procedures. In speed-sensitive code, you might want to use these to avoid the overhead of optional-argument handling.

`get-uint` *size uv pos :optional endian* [Function]
`get-sint` *size uv pos :optional endian* [Function]

{`binary.io`} Read *size* octets from uvector *uv*, starting from *pos*-th octet, as an unsigned or signed integer, respectively.

```
(get-uint 3 '#u8(1 2 3 4) 1 'big-endian)
  ⇒ 131884 ; #x020304

(get-sint 3 '#u9(1 2 3 #xff) 1 'little-endian)
  ⇒ -64766 ; sign extended #xff0302
```

`put-u8!` *uv pos val :optional endian* [Function]
`put-u16!` *uv pos val :optional endian* [Function]
`put-u32!` *uv pos val :optional endian* [Function]
`put-u64!` *uv pos val :optional endian* [Function]
`put-s8!` *uv pos val :optional endian* [Function]
`put-s16!` *uv pos val :optional endian* [Function]
`put-s32!` *uv pos val :optional endian* [Function]
`put-s64!` *uv pos val :optional endian* [Function]
`put-f16!` *uv pos val :optional endian* [Function]
`put-f32!` *uv pos val :optional endian* [Function]
`put-f64!` *uv pos val :optional endian* [Function]

{`binary.io`} Writes a number *val* into a uniform vector *uv* in a specific format, starting at a byte position *pos*. An error is signaled if the specified position makes reference outside of the uniform vector's content.

`put-u16be!` *uv pos val* [Function]
`put-u16le!` *uv pos val* [Function]
`put-u32be!` *uv pos val* [Function]
`put-u32le!` *uv pos val* [Function]
`put-u64be!` *uv pos val* [Function]
`put-u64le!` *uv pos val* [Function]
`put-s16be!` *uv pos val* [Function]
`put-s16le!` *uv pos val* [Function]
`put-s32be!` *uv pos val* [Function]
`put-s32le!` *uv pos val* [Function]
`put-s64be!` *uv pos val* [Function]
`put-s64le!` *uv pos val* [Function]
`put-f16be!` *uv pos val* [Function]
`put-f16le!` *uv pos val* [Function]
`put-f32be!` *uv pos val* [Function]
`put-f32le!` *uv pos val* [Function]
`put-f64be!` *uv pos val* [Function]
`put-f64le!` *uv pos val* [Function]

{`binary.io`} These are big-endian (`be`) or little-endian (`le`) specific versions of `put-*` procedures. In speed-sensitive code, you might want to use these to avoid the overhead of optional-argument handling.

`put-uint!` *size uv pos val :optional endian* [Function]
`put-sint!` *size uv pos val :optional endian* [Function]

{`binary.io`} Write an unsigned or signed integer *val* into an uvector *uv* starting from *pos*-th octet, for *size* octets, respectively.

Compatibility notes

`read-u8` etc. were called `read-binary-uint8` etc., and `read-f32` and `read-f64` were called `read-binary-float` and `read-binary-double`, respectively. These old names are still supported for the backward compatibility but their use is deprecated. The reason of the changes is for brevity and for consistency with the uniform vectors.

12.2 `binary.pack` - Packing binary data

`binary.pack` [Module]

This module provides an interface for packing and unpacking (writing and reading) binary data with templates. The functionality was inspired largely by the Perl pack/unpack functions, with comparison of similar features from other languages, however an effort was made to make it more general and more efficient, to be usable for database-like processing. To that end, the most notable differences are that any packable value is unpackable (and vice versa), and the default behavior is to pack and unpack using port I/O, so you can seek in a large file and unpack from it. Also, templates may be stored as dispatch closures to pack, unpack or even skip over values without re-parsing the template.

`pack` *template list :key output to-string?* [Function]

{`binary.pack`} Writes the values in *list* to the current output port, according to the format specified by the string *template*. The template string is a series of single character codes, optionally followed by a numeric count (which defaults to 1). The format characters can generally be divided into string types, which interpret the count as a string byte size, and object types, which treat the count as a repetition indicator. The count may be specified as the character `*`, which means to use the full size of the string for string types, and use all remaining values for object types. Counts may also be specified as a template enclosed in brackets, which means the count is the byte size of the enclosed template. For example, `x[L]` skips a long. The special format character `/` may be used to indicate a structure where the packed data contains a dynamic count followed by the value itself. The template is written as `<count-item>/<value-item>`, where `<count-item>` is any template character to be interpreted as a numeric count, and `<value-item>` is any other template character to use this count. If a normal count is given after `<value-item>` it is ignored. The format character `@` may be used with a count to pad to an absolute position since the start of the template. Sub-templates may be grouped inside parentheses. If angle-brackets are used, then they also behave as group operators but recursively operate on nested lists. The string types:

`a` An arbitrary incomplete string, null padded.

`A` A text string, space padded.

`Z` A null terminated (ASCIZ) string, null padded.

`b` A bit string (ascending bit order inside each byte).

`B` A bit string (descending bit order inside each byte).

`h` A hex string (low nybble first).

`H` A hex string (high nybble first).

The object types:

`c` A signed 8bit integer.

`C` An unsigned 8bit integer.

`s` A signed short (16 bit) value.

`S` An unsigned short (16 bit) value.

`i` A signed integer (>= 32 bit) value.

`I` An unsigned integer (>= 32 bit) value.

`l` A signed long (32 bit) value.

`L` An unsigned long (32 bit) value.

`n, n!` An unsigned and signed short (16 bit) in "network" (big-endian) order.

`N, N!` An unsigned and signed long (32 bit) in "network" (big-endian) order.

`v, v!` An unsigned and signed short (16 bit) in "VAX" (little-endian) order.

`V, V!` An unsigned and signed long (32 bit) in "VAX" (little-endian) order.

`q` A signed quad (64 bit) value.

`Q` An unsigned quad (64 bit) value.

`f` A single-precision float in the native format.

`d` A double-precision float in the native format.

`w` A BER compressed integer. An unsigned integer in base 128, most significant digit first, where the high bit is set on all but the final (least significant) byte. Thus any size integer can be encoded, but the encoding is efficient and small integers don't take up any more space than they would in normal char/short/int encodings.

`x` A null byte.

`o` An sexp, handled with `read` and `write`.

If the optional keyword *:output* is given that port is used instead of the current output port. If *:to-string?* is given and true, then pack accumulates and returns the output as a string.

Note that the returned string may be an incomplete string if the packed string contains a byte sequence invalid as a character sequence.

```
(pack "CCCC" '(65 66 67 68) :to-string? #t)
 ⇒ "ABCD"

(pack "C/a*" '("hello") :to-string? #t)
 ⇒ "\x05hello"
```

`unpack` *template :key :input :from-string* [Function]

{`binary.pack`} The complement of pack, unpack reads values from the current input port assuming they've been packed according to the string template and returns the values as a list. unpack accepts the same format strings as pack. Further, the following tautology holds:

```
(equal? x (unpack fmt :from-string (pack fmt x :to-string? #t)))
```

for any list *x* and format string *fmt*. The only exceptions to this are when the template includes a `*` and when the `o` template is used, since Scheme numeric literals cannot be reliably delimited (though future versions of `pack` may circumvent this by registering a new read syntax).

If the optional keyword *:input* is given that port is used instead of the current input port. If *:from-string* is given, then pack reads input from that string.

```
(unpack "CCCC" :from-string "ABCD")
 ⇒ '(65 66 67 68)

(unpack "C/a*" :from-string "\x05hello")
 ⇒ '("hello")
```

Note: in the current version, `@` in `unpack` template has a bug and does not work as supposed. It will be fixed in the future version.

`unpack-skip` *template :key :input* [Function]

{`binary.pack`} unpack-skip is the same as unpack except it does not return the values. In some cases, particularly with fixed-size templates, this can be much more efficient when you just want to skip over a value.

`make-packer` *template* [Function]

{`binary.pack`} The low-level interface. This function returns a dispatch closure that can be used to pack, unpack and skip over the same cached template. The dispatch closure accepts symbol methods as follows:

`'pack list`
pack the items in list to the current output port.

`'unpack` unpack items from the current input port.

`'skip` skip items from the current input port.

`'packer` return the cached 'pack closure

`'unpacker`
return the cached 'unpack closure.

`'skipper` return the cached 'skip closure.

`'length` return the known fixed length of the template.

'variable-length?
return #t if the template has variable length elements.

12.3 compat.chibi-test - Running Chibi-scheme test suite

compat.chibi-test [Module]

Quite a few srfis come with test suites that's to be run with Chibi Scheme test framework. This module enables Gauche to run the test code as is.

chibi-test *code* ... [Macro]

Run *code* ..., while translating Chibi test framework to Gauche's.

A typical usage is to write a wrapper that includes the original test code (suppose it's called `test-suite.scm`):

```
(use gauche.test)
(test-start "running test-suite.scm")
(chibi-test
  (include "test-suite.scm"))
(test-end)
```

Chibi's test directives are translated to Gauche's test directives (see Section 9.32 [Unit testing], page 452, for Gauche's test framework).

The main thing is that Chibi allows expressions and definitions to be intermingled within a body, while Gauche only allows all definitions before expressions within a body. We expand such body into nested `let` by `chibi-test` macro. Chibi test macros (e.g. `test-assert`) are defined as local macros in `chibi-test` expansion, which expand into `gauche.test` macros.

Note that we ignore `use` forms inside `chibi-test`; we might want to use different modules that work better in Gauche. Necessary modules need to be `use`'d before you call `chibi-test`.

You may want to check out `test/srfi.scm` in Gauche source tree for the use case.

12.4 compat.norational - Rational-less arithmetic

compat.norational [Module]

Until release 0.8.7, Gauche didn't have exact rational numbers. It was able to read the rational number literals such as `2/3`, but they are immediately coerced to inexact real numbers (except when it represents a whole integer). And if you divided an exact integer by another exact integer, the result could be coerced to an inexact real if the result wasn't a whole integer.

As of 0.8.8, this is not the case anymore. Exact division always yields exact result, except when the divisor is zero.

```
(/ 2 3)  ⇒ 2/3
(/ 5)    ⇒ 1/5
(/ 4 2)  ⇒ 2
```

This is more precise, but has one drawback: exact rational arithmetic is much slower than the integer and inexact real arithmetic. If you inadvertently produce a rational number in the early stage of calculation, and continue to apply exact arithmetic, performance would be degraded miserably.

The proper way to solve this is to insert `exact->inexact` to appropriate places. However, to ease the transition, you can just import this module and the division `/` behaves in the way it used to.

```
(use compat.norational)

(/ 2 3)  ⇒ 0.6666666666666666
```

```
(/ 5)     ⇒ 0.2
(/ 4 2)   ⇒ 2
```

The effect is not global, but only to the modules you explicitly import `compat.norational`.

This module only redefines `/`. So if your code has exact rational literals, they are treated as exact rationals rather than coerced to inexact reals. You should prefix rational literals with `#i` to force Gauche to coerce them to inexact reals:

```
gosh> 1/3
1/3
gosh> #i1/3
0.3333333333333333
```

12.5 `control.job` - A common job descriptor for control modules

`control.job` [Module]

This module provides a `job` record type, a lightweight structure to be used in the control flow subsystems (`control.*` modules). Currently the only user is `control.thread-pool`, but some other modules are planned to use `job` records.

A `job` record may be returned to an application by other `control.*` modules so that the application can keep track of the job. It's not meant for general use, however. An application isn't supposed to create a new job, or to modify its content; it can just query the job's properties.

In this section we only describe procedures an application needs to know. The interface for control subsystems is still fluid and may be changed as more subsystems are developed.

Different control flow subsystems may use job structure differently. This section only describes the common properties. Check the individual control flow module to know how to handle returned job objects.

`job` [Record type]

{`control.job`} A record type denotes the job. Applications should treat it as an opaque structure.

`job?` *obj* [Function]

{`control.job`} Returns `#t` iff *obj* is a job record, `#f` otherwise.

`job-status` *job* [Function]

{`control.job`} Returns the status of the job. It may be either one of the followings.

`#f` Newborn or orphaned job. Usually an application won't see a job in this status.

`acknowledged`
A job is recognized by a control flow library, but haven't yet been run.

`running` A job is being processed.

`done` A job is finished. An application can retrieve its result by `job-result`.

`error` A job is terminated by an error. An application can retrieve the error causing condition by `job-result`.

`killed` A job is killed by external force. An application can retrieve the reason of kill (which is specific to a particular control flow subsystem) by `job-result`.

`job-result` *job* [Function]

{`control.job`} If the job is in `done` status, it returns the result of the job. If the job is in `error` status, it returns the condition object that describes the error. If the job is in `killed` status, it returns an object describing the reason of kill. The details of the object depends on a particular control flow library. Calling `job-result` on a job in any other status may return anything; you can't rely on the result.

`job-wait` *job :optional timeout timeout-val* [Function]

{`control.job`} Suspends the calling thread until the job becomes either `done`, `error` or `killed` status. If the job is already in one of those status, it returns immediately. Returns job's status.

If *timeout* is given and not `#f`, it must be a valid timeout spec (a `<time>` object that represents an absolute time point, or a real number that represents a relative time in seconds.) The meaning of *timeout* is the same as in `mutex-unlock!` (see Section 9.33.3 [Synchronization primitives], page 462). Once the timeout reaches, `job-wait` returns no matter how the job's status is, and returns the value specified to *timeout-val*, which defaults to `#f`.

Depending on the control flow subsystem, jobs created by it may not be waitable; check out each subsystem's documentation for the details.

`job-acknowledge-time` *job* [Function]
`job-start-time` *job* [Function]
`job-finish-time` *job* [Function]

{`control.job`} If the control flow subsystem keeps track of timestamps, these procedure returns the time (in `<time>` objects) when the job is acknowledged, started and finished (either normally, or abnormally by an error or by being killed). If the job hasn't reached to certain status, `#f` is returned instead.

If the subsystem does not track timestamps, these procedures always returns `#f`.

12.6 `control.thread-pool` - Thread pools

`control.thread-pool` [Module]

Provides thread pools. Only available when Gauche is compiled with pthreads support.

`<thread-pool>` [Class]

{`control.thread-pool`} A class for thread pool objects. It maintains a set of worker threads, and let them work on the jobs you ask to do asynchronously.

Currently the size of pool (number of threads) is fixed and you have to specify it when creating a pool. In future we might add a feature to grow or shrink the pool.

You can also set maximum backlog of the job queue. You cannot put a job when the queue already reaches the max length (see `add-job!` below).

`<thread-pool-shut-down>` [Condition type]

{`control.thread-pool`} A condition indicating that a thread pool is already shut down by `terminate-all!` and no longer accepting new jobs. Inherits `<error>`. The following slot is provided.

`pool` [Instance Variable of `<thread-pool-shut-down>`]

The thread pool object that caused the condition.

`make-thread-pool` *size :key* (*max-backlog 0*) [Function]

{`control.thread-pool`} Creates a new thread pool of size *size* (the number of worker threads). Optionally you can give a nonnegative integer to the maximum backlog; 0 means unlimited.

thread-pool-results *pool* [Function]

{control.thread-pool} When you put a job to a thread pool, you can specify whether you need to check its result or not. If you say you need a result, the terminated job is queued to a *result queue*, an <mt-queue> object, in the pool. This procedure returns the pool's result queue. See Section 12.12 [Queue], page 656, for the details of <mt-queue>.

thread-pool-shut-down? *pool* [Function]

{control.thread-pool} Returns #t if the thread pool is shut down and no longer accepting new jobs, or #f otherwise.

add-job! *pool thunk :optional* (*need-result #f*) (*timeout #f*) [Function]

{control.thread-pool} Add a *thunk* to be executed in the thread pool *pool*. Returns a job record (see Section 12.5 [A common job descriptor for control modules], page 645).

The returned job record is not waitable; if you need to track its result, you have to give a true value to *need-result* argument. Then when the job is terminated (either normally or abnormally) the job is queued to the result-queue of the pool, and you can check the queue. If you don't pass a true value to *need-result*, the job won't be queued to result-queue even it is terminated.

The returned job is timestamped. You can examine acknowledged time, start time and finish time of the job (if the job hasn't been started and/or finished, the corresponding timestamp fields are #f.) It's sometimes handy to find out how long the job was waiting in the queue and how long it took to run.

If the pool has positive max-backlog value, and it already has that many jobs to be waiting, then add-job! blocks until some jobs are start being executed. You can give a real number in seconds, or a <time> object as an absolute point of time, to the *timeout* argument to set the time limit of blocking. If timeout is reached, add-job! returns #f without creating any job. Omitting *timeout* or giving #f to it sets no timeout.

(Note: This behavior is different from 0.9.1, in which add-job! didn't take the timeout argument and always behaved as if zero timeout value was given. To achieve the same behavior, you have to give 0 to the *timeout* argument explicitly.)

If the thread pool is shut down, this procedure raises <thread-pool-shut-down> condition.

wait-all *pool :optional* (*timeout #f*) (*check-interval #e5e8*) [Function]

{control.thread-pool} Wait for the job queue to be empty and all worker threads to finish. It is done by polling the pool's status in every *check-interval* nanoseconds. Returns #t if all jobs are finished.

You can give a real number in seconds, or a <time> object as an absolute point of time, in *timeout* optional argument. When timeout is reached, wait-all returns #f.

terminate-all! *pool :key* (*force-timeout #f*) (*cancel-queued-jobs #f*) [Function]

{control.thread-pool} Wait for all the queued jobs to be finished, then ask all threads to terminate. After calling this procedure, the pool no longer accepts new jobs. Calling add-job! on this module would raise a <thread-pool-shut-down> condition. This is intended to be called when shutting down the application.

By default, this procedure first waits for all queued jobs to be handled, then tries to terminate threads gracefully.

Giving a true value to the *cancel-queued-jobs* argument immediately cancels queued but not started jobs; the status of such jobs is set killed. It does not cancels already started jobs, though.

If you want to cancel already started jobs, you can give a timeout value (either <time> object to specify absolute point of time, or a real number indicating relative time in seconds) to the *force-timeout* argument. Once timeout is reached, it forcefully terminates the threads and the jobs handled at that time are also killed.

Forcing termination of threads is an extreme measure; the terminated thread may not have a chance to clean up properly. So it is usually better to give some time for the thread to finish the executing jobs.

12.7 `crypt.bcrypt` - Password hashing

`crypt.bcrypt` [Module]

This module implements a password hashing algorithm using blowfish, and compatible to OpenBSD's bcrypt algorithm (version 2a, 2b).

Don't use version "2a" for new code. It's vulnerable. Use version "2b".

The typical usage of this module is simple enough. To get a new password hash value (e.g. for a new user), pass the password string to `bcrypt-hashpw` as the only argument:

```
(bcrypt-hashpw password)
  ⇒ hashed-string
```

The routine automatically adds a salt value. The returned hash string can be stored in the user database. To check if the given password matches the stored one, pass the hashed string as the second argument of `bcrypt-hashpw` to check the password.

```
(bcrypt-hashpw password hashed-string)
  ⇒ hashed-string
```

If the given password is correct, the returned value should exactly matches *hash-string*.

`bcrypt-hashpw` *password :optional setting* [Function]

{`crypt.bcrypt`} Calculates a hash value of *password*, using the salt value and parameters included in *setting*. If *setting* is omitted, a suitable default settings and random salt value is chosen automatically.

The returned hash value contains the salt value and parameters, and can be used as *setting*. So, to check the password against existing hash value, just pass the hash value to *setting*; if the password is correct, the returned hash value should match the one you passed in.

The bcrypt algorithm supports up to 72 octets for the password.

To tweak parameters when you calculate a new hash value, use `bcrypt-gensalt` below to get the initial *setting* value.

`bcrypt-gensalt` *:key prefix count entropy-source* [Function]

{`crypt.bcrypt`} Returns a string that contains given parameters and suitable to pass to the *setting* argument of `bcrypt-hashpw`.

The *prefix* argument specifies the version/scheme of password hashing. Currently `$2a$` and `$2b$` are supported, which means the blowfish algorithm compatible to bcrypt. But `$2a$` is vulnerable. Use `$2b$` for new code. If you omit *prefix*, use `$2b$` for default value.

The *count* argument specifies the amount of iterations; the larger the value is, the more time is required to calculate the hash value. Note that for the password hashing, taking more time is actually a good thing, for it works against the dictionary attack. For normal password checking you need to run the hash routine only once per login, so it doesn't matter if the calculation takes a fraction of second. The bcrypt algorithm iterates `(expt 2 count)` times.

The *entropy-source* argument is a `u8vector` to feed a random bytes. For bcrypt algorithm it must be at least 16 octet long.

12.8 data.cache - Cache

`data.cache` [Module]

A cache works similarly as a dictionary, associating keys to values, but its entries may disappear according to the policy of the cache algorithm. This module defines a common protocol for cache datatypes, and also provides several typical cache implementations.

Examples

Let's start from simple examples to get the idea.

Suppose you want to read given files and you want to cache the frequently read ones. The following code defines a cached version of `file->string`:

```
(use data.cache)
(use file.util)

(define file->string/cached
  (let1 file-cache (make-lru-cache 10 :comparator string-comparator)
    (^[path] (cache-through! file-cache path file->string))))
```

The procedure closes a variable `file-cache`, which is an LRU (least recently used) cache that associates string pathnames to the file contents. The actual logic is in `cache-through!`, which first consults the cache if it has an entry for the `path`. If the cache has the entry, its value (the file content) is returned. If not, it calls `file->string` with the *path* to fetch the file content, register it to the cache, and return it. The capacity of cache is set to 10 (the first argument of `make-lru-cache`), so when the 11th file is read, the least recently used file will be purged from the cache.

The effect of cache isn't very visible in the above example. You can insert some print stubs to see the cache is actually in action, as the following example. Try read various files using `file->string/cached`.

```
(define file->string/cached
  (let1 file-cache (make-lru-cache 10 :comparator string-comparator)
    (^[path]
      (print #"file->string/cached called on ~path")
      (cache-through! file-cache path
                      (^[path]
                        (print #"cache miss.  fetching ~path")
                        (file->string path))))))
```

Caveat: A cache itself isn't MT-safe. If you are using it in multithreaded programs, you have to wrap it with an atom (see Section 9.33.3 [Synchronization primitives], page 462):

```
(use data.cache)
(use file.util)
(use gauche.threads)

(define file->string/cached
  (let1 file-cache (atom (make-lru-cache 10 :comparator string-comparator))
    (^[path]
      (atomic file-cache (cut cache-through! <> path file->string)))))
```

Common properties of caches

A cache of any kind has a comparator and a storage. The comparator is used to compare keys; in the example above, we use `string-comparator` to compare string pathnames (see Section 6.2.4 [Basic comparators], page 103, for more about comparators).

The storage is a dictionary that maps keys to internal structures of the cache. By default, a hashtable is created automatically using the given comparator (or, if a comparator is omitted, using `default-comparator`). The comparator must have hash function.

Alternatively, you can give a pre-filled dictionary (copied from another instance of the same kind of cache) to start cache with some data already in it. Note that what the cache keeps in the dictionary totally depends on the cache algorithm, so you can't just pass a random dictionary; it has to be created by the same kind of cache. If you pass in the storage, the comparator is taken from it.

Thus, the cache constructors uniformly take keyword arguments *comparator* and *storage*; you can specify either one, or omit both to use the defaults.

Predefined caches

For the *storage* and *comparator* keyword arguments, see above.

`make-fifo-cache` *capacity :key storage comparator* [Function]

{`data.cache`} Creates and returns a FIFO (first-in, first-out) cache that can hold up to *capacity* entries. If the number of entries exceeds *capacity*, the oldest entry is removed.

`make-lru-cache` *capacity :key storage comparator* [Function]

{`data.cache`} Creates and returns an LRU (least recently used) cache that can hold up to *capacity* entries. If the number of entries exceeds *capacity*, the least recently used entry is removed.

`make-ttl-cache` *timeout :key storage comparator timestamper* [Function]

{`data.cache`} Creates and returns a TTL (time to live) cache with the timeout value *timeout*. Each entry is timestamped when it's inserted, and it is removed when the current time passes *timeout* unit from the timestamp. The actual entry removal is done when the cache is accessed.

By default, the Unix system time (seconds from Epoch) is used as a timestamp, and *timeout* is in seconds. It may not be fine-grained enough if you add multiple entries in shorter intervals than seconds. You can customize it by giving a thunk to *timestamper*; the thunk is called to obtain a timestamp, which can be any monotonically increasing real number (it doesn't need to be associated with physical time). If you give *timestamper*, the unit of *timeout* value should be the same as whatever *timestamper* returns.

`make-ttlr-cache` *timeout :key storage comparator timestamper* [Function]

{`data.cache`} A variation of TTL cache, but the entry's timestamp is updated (refreshed) whenever the entry is read. Hence we call it TTL with refresh (TTLR). But you can also think it as a variation of LRU cache with timeout.

The unit of timeout, and the role of *timestamper* argument, are the same as `make-ttl-cache`.

Common operations of caches

The following APIs are for the users of a cache.

`cache-lookup!` *cache key :optional default* [Function]

{`data.cache`} Look for an entry with *key* in *cache*, and returns its value if it exists. If there's no entry, the procedure returns *default* if it is provided, or throws an error otherwise.

Some types of cache algorithms update *cache* by this operation, hence the bang is in the name.

`cache-through!` *cache key value-fn* [Function]

{`data.cache`} Look for an entry with *key* in *cache*, and returns its value if it exists. If there's no entry, a procedure *value-fn* is called with *key* as the argument, and its return value is inserted into *cache* and also returned.

`cache-write!` *cache key value* [Generic function]

{`data.cache`} This inserts association of *key* and *value* into *cache*. If there's already an entry with *key*, it is overwritten. Otherwise a new entry is created.

The same effect can be achieved by calling `cache-evict!` then `cache-through!`, but cache algorithms may provide efficient way through this method.

`cache-evict!` *cache key* [Generic function]

{`data.cache`} Removes an entry with *key* from *cache*, if it exists.

`cache-clear!` *cache* [Generic function]

{`data.cache`} Removes all entries from *cache*.

Implementing a cache algorithm

Each cache algorithm must define a class inheriting `<cache>`, and implement the following two essential methods. The higher-level API calls them.

`cache-check!` *cache key* [Generic function]

{`data.cache`} Looks for an entry with *key* in *cache*. If it exists, returns a pair of *key* and the associated value. Otherwise, returns `#f`. It may update the cache, for example, the timestamp of the entry for being read.

`cache-register!` *cache key value* [Generic function]

{`data.cache`} Add an entry with *key* and associated *value* into *cache*. This is called after *key* is confirmed not being in *cache*.

Additionally, the implementation should consider the following points.

- The initialize method must call `next-method` first, which sets up the `comparator` and `storage` slots. You should check if `storage` has pre-filled entries, and if so, set up other internal structures appropriately.
- The default methods of `cache-evict!` and `cache-clear!` only takes care of the storage of the cache. You should implement them if your auxiliary structure needs to be taken care of.
- The default method of `cache-write!` is just `cache-evict!` followed by `cache-register!`. You may provide alternative method if you can do it more efficiently, which is often the case.

There are several procedures that help implementing cache subclasses:

`cache-comparator` *cache* [Function]

`cache-storage` *cache* [Function]

{`data.cache`} Returns the comparator and the storage of the cache, respectively.

Typical caches may be constructed with a storage (dictionary) and a queue, where the storage maps keys to `(<n> . <value>)`, and queues holds `(<key> . <n>)`, `<n>` being a number (timestamp, counter, etc.) Here are some common operations work on this queue-and-dictionary scheme:

`cache-populate-queue!` *queue storage* [Function]

{`data.cache`} You can call this in the `initialize` method to set up the queue. This procedure walks storage to construct `(<key> . <n>)` pairs, sorts it in increasing order of `<n>`, and pushes them into the queue.

`cache-compact-queue!` *queue storage* [Function]

{`data.cache`} The queue may contain multiple pairs with the same key. Sometimes the queue gets to have too many duplicated entries (e.g. the same entry is read repeatedly). This scans the queue and removes duplicated entries but the up-to-date one. After this operation, the length of the queue and the number of entries in the storage should match.

`cache-renumber-entries!` *queue storage* [Function]

{`data.cache`} This procedure renumbers `<n>`s in the queue and the storage starting from 0, without changing their order, and returns the maximum `<n>`. The duplicated entries in the queue is removed as in `cache-compact-queue!`.

When you're using monotonically increasing counter for `<n>` and you don't want `<n>` to get too big (i.e. bignums), you can call this procedure occasionally to keep `<n>`'s in reasonable range.

12.9 `data.heap` - Heap

`data.heap` [Module]

A heap is a data container that allows efficient retrieval of the minimum or maximum entry. Unlike a `<tree-map>` (see Section 6.16 [Treemaps], page 182), which always keeps all entries in order, a heap only cares the minimum or the maximum of the current set; the other entries are only partially ordered, and reordered when the minimu/maximum entry is removed. Hence it is more efficient than a treemap if all you need is minimum/maximum value. Besides binary heaps can store entries in packed, memory-efficient way.

`<binary-heap>` [Class]

{`data.heap`} An implementation of a binary heap. Internally it uses min-max heap, so that you can find both minimum and maximum value in O(1). Pushing a new value and popping the minimum/maximum value are both O(log n).

It also stores its values in a flat vector, a lot more compact than a general tree structure that needs a few pointers per node. By default it uses a sparse vector for the backing storage, allowing virtually unlimited capacity (see Section 12.15.1 [Sparse vectors], page 667). But you can use an ordinal vector or a uniform vector as a backing storage instead.

A binary heap isn't MT-safe structure; you must put it in atom or use mutexes if multiple threads can access to it (see Section 9.33.3 [Synchronization primitives], page 462).

`make-binary-heap` *:key comparator storage key* [Function]

{`data.heap`} Creates and returns a new binary heap.

The *comparator* keyword argument specifies how to compare the entries. It must have comparison procedure or ordering predicate. The default is `default-comparator`. See Section 6.2.4 [Basic comparators], page 103, for the details of comparators.

The *storage* keyword argument gives alternative backing storage. It must be either a vector, a uniform vector, or an instance of a sparse vector (see Section 12.15.1 [Sparse vectors], page 667). The default is an instance of `<sparse-vector>`. If you pass a vector or a uniform vector, it determines the maximum number of elements the heap can hold. The heap won't be extend the storage once it gets full.

The *key* keyword argument must be a procedure; it is applied on each entry before comparison. Using key procedure allows you to store auxiliary data other than the actual value to be compared. The following example shows the entries are compared by their `car`'s:

```
(define *heap* (make-binary-heap :key car))
(binary-heap-push! *heap* (cons 1 'a))
(binary-heap-push! *heap* (cons 3 'b))
```

```
(binary-heap-push! *heap* (cons 1 'c))

(binary-heap-find-min *heap*) ⇒ (1 . c)
(binary-heap-find-max *heap*) ⇒ (3 . b)
```

`build-binary-heap` *storage :key comparator key num-entries* [Function]

{`data.heap`} Create a heap from the data in *storage*, and returns it. (Sometimes this operation is called *heapify.*) This allows you to create a heap without allocating a new storage. The *comparator* and *key* arguments are the same as `make-binary-heap`.

Storage must be either a vector, a uniform vector, or an instance of a sparse vector. The storage is modified to satisfy the heap property, and will be used as the backing storage of the created heap. Since the storage will be owned by the heap, you shouldn't modify the storage later.

The storage supposed to have keys from index 0 below *num-entries*. If *num-entries* is omitted or `#f`, entire vector or uniform vector, or up to `sparse-vector-num-entries` on the sparse vector, is heapified.

`binary-heap-copy` *heap* [Function]

{`data.heap`} Copy the heap. The backing storage is also copied.

`binary-heap-clear!` *heap* [Function]

{`data.heap`} Empty the heap.

`binary-heap-num-entries` *heap* [Function]

{`data.heap`} Returns the current number of entries in the heap.

`binary-heap-empty?` *heap* [Function]

{`data.heap`} Returns `#t` if the heap is empty, `#f` otherwise.

`binary-heap-push!` *heap item* [Function]

{`data.heap`} Insert *item* into the *heap*. This is O(log n) operation. If the heap is already full, an error is raised.

`binary-heap-find-min` *heap :optional fallback* [Function]
`binary-heap-find-max` *heap :optional fallback* [Function]

{`data.heap`} Returns the minimum and maximum entry of the heap, respectively. The heap will be unmodified. This is O(1) operation.

If the heap is empty, *fallback* is returned when it is provided, or an error is signaled.

`binary-heap-pop-min!` *heap* [Function]
`binary-heap-pop-max!` *heap* [Function]

{`data.heap`} Removes the minimum and maximum entry of the heap and returns it, respectively. O(log n) operation. If the heap is empty, an error is signaled.

The following procedures are not heap operations, but provided for the convenience.

`binary-heap-swap-min!` *heap item* [Function]
`binary-heap-swap-max!` *heap item* [Function]

{`data.heap`} These are operationally equivalent to the followings, respectively:

```
(begin0 (binary-heap-pop-min! heap)
  (binary-heap-push! heap item))

(begin0 (binary-heap-pop-max! heap)
  (binary-heap-push! heap item))
```

However, those procedures are slightly efficient, using heap property maintaining procedure only once per function call.

`binary-heap-find` *heap pred* [Function]

{`data.heap`} Returns an item in the heap that satisfies *pred*. If there are more than one item that satisfy *pred*, any one of them can be returned. If no item satisfy *pred*, `#f` is returned. This is O(n) operation.

`binary-heap-remove!` *heap pred* [Function]

{`data.heap`} Remove all items in the heap that satisfy *pred*. This is O(n) operation.

`binary-heap-delete!` *heap item* [Function]

{`data.heap`} Delete all items in the heap that are equal to *item*, in terms of the heap's comparator and key procedure. This is O(n) operation.

Note that the key procedure is applied to *item* as well before comparison.

12.10 `data.ideque` - Immutable deques

`data.ideque` [Module]

This module provides a functional double-ended queue (deque, pronounced as "deck").

Almost all procedures in this module are now a part of R7RS large. See Section 10.3.8 [R7RS immutable deques], page 540, for description of the following procedures:

`ideque`	`ideque-unfold`	`ideque-unfold-right`
`ideque-tabulate`	`ideque?`	`ideque-empty?`
`ideque-add-front`	`ideque-add-back`	
`ideque-remove-front`	`ideque-remove-back`	
`ideque-front`	`ideque-back`	
`ideque-reverse`	`ideque=`	`ideque-ref`
`ideque-take`	`ideque-drop`	
`ideque-take-right`	`ideque-drop-right`	
`ideque-split-at`	`ideque-append`	`ideque-zip`
`ideque-map`	`ideque-for-each`	`ideque-for-each-right`
`ideque-fold`	`ideque-fold-right`	`ideque-append-map`
`ideque-filter`	`ideque-remove`	
`ideque-find`	`ideque-find-right`	
`ideque-take-while`	`ideque-take-while-right`	
`ideque-drop-while`	`ideque-drop-while-right`	
`ideque-span`	`ideque-break`	
`ideque-any`	`ideque-every`	
`ideque->list`	`list->ideque`	
`ideque->generator`	`generator->ideque`	

`make-ideque` *n :optional init* [Function]

{`data.ideque`} Creates an ideque of length *n* with all the elements being *init*. If *init* is omitted, `#f` is used.

This is provided just for the symmetry with other container data structures; it's not in srfi-134, and the portable code can use `ideque-tabulate`.

12.11 `data.imap` - Immutable map

`data.imap` [Module]

This module provides a immutable data structure with O(log n) access and update operations (here, update means to return a new structure with requested changes). The current implementation is based on the functional red-black tree.

Although lists and alists are useful for stack-like immutable operations, where you can add and remove items to the head of existing data without modifying them, they require O(n) access time and sometimes you need better one. The `<imap>` object provides O(log n) access, in exchange of O(log n) insertion and deletion.

`<imap-meta>` [Class]

{`data.imap`} Metaclass of `<imap>`.

`<imap>` [Class]

{`data.imap`} Immutable map class. An instance of `<imap-meta>`.

Inherits `<ordered-dictionary>`, conforms dictionary protocol except mutating operators (see Section 9.9 [Dictionary framework], page 366). As a sequence, you can access key-value pairs in increasing order of keys.

`make-imap` [Function]
`make-imap` *comparator* [Function]
`make-imap` *key=? key<?* [Function]

{`data.imap`} Creates a new empty immutable map. Without arguments, `default-comparator` is used to compare keys. To give a specific comparator, use the second form; the *comparator* argument should have comparison procedure. For the details of comparators, see Section 6.2.4 [Basic comparators], page 103. The third form creates a key comparator from a equality predicate *key=?* and less-than predicate `key<?`, both must accept two keys. This interface is consistent with `tree-map` (see Section 6.16 [Treemaps], page 182).

`alist->imap` *alist* [Function]
`alist->imap` *alist comparator* [Function]
`alist->imap` *alist key=? key<?* [Function]

{`data.imap`} Creates a new empty immutable map, populates it with key-value association list *alist*, and returns it. This may be a bit more efficient than creating an empty map with `make-imap` and populates it with `imap-put` one by one.

The *comparator* argument specifies how to compare the keys. It must have comparison procedure. If omitted, `default-comparator` is used. See Section 6.2.4 [Basic comparators], page 103, for the details.

The third form creates a key comparator from a equality predicate *key=?* and less-than predicate `key<=?`, both must accept two keys.

```
(define m (alist->imap '((a . 1) (b . 2))))

(imap-get m 'a) ⇒ 1
(imap-get m 'b) ⇒ 2
```

`tree-map->imap` *tree-map* [Function]

{`data.imap`} Returns a new immutable map with the same content (and the same comparator) as *tree-map*.

`imap?` *obj* [Function]

{`data.imap`} Returns `#t` if *obj* is an immutable map, `#f` otherwise.

`imap-empty?` *immap* [Function]

{`data.imap`} Returns `#t` if an immutable map *immap* is empty, `#f` otherwise.

`imap-exists?` *immap key* [Function]

{`data.imap`} Returns `#t` if *key* exists in an immutable map *immap*.

`imap-get` *immap key :optional default* [Function]

{`data.imap`} Returns the value associated with *key* in an immutable map *immap*. If *immap* doesn't have *key*, *default* is returned when provided, otherwise an error is signalled.

`imap-put` *immap key val* [Function]

{`data.imap`} Returns a new immutable map where association of *key* to *val* is added to (or replaced in) an immutable map *immap*. This operation is O(log n).

```
(define m1 (alist->imap '((a . 1) (b . 2))))

(define m2 (imap-put m1 'a 3))

(imap-get m2 'a)  ⇒ 3
(imap-get m1 'a)  ⇒ 1  ; not affected
```

`imap-delete` *immap key* [Function]

{`data.imap`} Returns a new immutable map where *key* is removed from *immap*. If *immap* doesn't have *key*, returned map has the same content as *immap*.

```
(define m1 (alist->imap '((a . 1) (b . 2))))

(define m2 (imap-delete m1 'a))

(imap-get m2 'a #f)  ⇒ #f
(imap-get m1 'a)     ⇒ 1  ; not affected
```

`imap-min` *immap* [Function]

`imap-max` *immap* [Function]

{`data.imap`} Returns a pair of key and value with the minimum or maximum key in *immap*, respectively. If *immap* is empty, `#f` is returned.

12.12 `data.queue` - Queue

`data.queue` [Module]

Provides a queue (FIFO). You can create a simple queue, which is lightweight but not thread-safe, or an MTqueue, a thread-safe queue. Basic queue operations work on both type of queues. When an mtqueue is passed to the procedures listed in this section, each operation is done in atomic way, unless otherwise noted.

There are also a set of procedures for mtqueues that can be used for thread synchronization; for example, you can let the consumer thread block if an mtqueue is empty, and/or the producer thread block if the number of items in the mtqueue reaches a specified limit. Using these procedures allows the program to use an mtqueue as a *channel*.

The simple queue API is a superset of SLIB's queue implementation, which supports not only `enqueue!` (add item to the end of the sequence) and `dequeue!` (take item from the front of the sequence), but also `queue-push!` (add item to the front of the sequence), so that it can be used as a stack as well.

If you also want to take item from the end of the sequence in O(1), you need a deque (double-ended queue). See Section 12.14 [Ring buffer], page 665, which works as an efficient (both speed and space) dequeue on top of vectors. Or you can use immutable deques provided by `data.ideque` (see Section 12.10 [Immutable deques], page 654).

See also SRFI-117 (Section 11.26 [Queues based on lists], page 631), which defines a portable API for list-based queue.

`<queue>` [Class]

{`data.queue`} A class of simple queue.

`length` [Instance Variable of `<queue>`]

A read-only slot that returns the number of items in the queue.

`<mtqueue>` [Class]

{`data.queue`} A class of mtqueue. Inherits `<queue>`.

`max-length` [Instance Variable of `<mtqueue>`]

The upper bound of the number of items in the queue.

If this slot is zero, the queue cannot hold any items, but works as a synchronization device. A writer will block until a reader appears to take the item; a reader will block until a writer appears to give the item.

`make-queue` [Function]

{`data.queue`} Creates and returns an empty simple queue.

`make-mtqueue` *:key max-length* [Function]

{`data.queue`} Creates and returns an empty mtqueue. When an integer is given to the keyword argument *max-length*, it is used to initialize the `max-length` slot.

`queue?` *obj* [Function]

{`data.queue`} Returns `#t` if *obj* is a queue (either a simple queue or an mtqueue).

`mtqueue?` *obj* [Function]

{`data.queue`} Returns `#t` if *obj* is an mtqueue.

`queue-empty?` *queue* [Function]

{`data.queue`} Returns `#t` if *obj* is an empty queue.

`queue-length` *queue* [Function]

{`data.queue`} Returns the number of the items in the queue.

`mtqueue-max-length` *mtqueue* [Function]

{`data.queue`} Returns the maximum number of items the mtqueue can hold. If the queue doesn't have a limit, `#f` is returned.

`mtqueue-room` *mtqueue* [Function]

{`data.queue`} Returns the number of elements the mtqueue can accept at this moment before it hits its maximum length. For example, if the queue already has the maximum number of elements, 0 is returned. If the queue doesn't have the limit, `+inf.0` is returned.

Note that even if this returns a non-zero finite value, subsequent `enqueue!` may throw an error because of the queue being full. It's because another thread may put an item to the queue between this procedure call and `enqueue!`. To avoid this situation, use `enqueue/wait!` to insert item to mtqueue with finite max-length.

`mtqueue-num-waiting-readers` *mtqueue* [Function]

{`data.queue`} Returns the number of threads waiting on the mtqueue to read at this moment. The return value is always a nonnegative exact integer.

Note that the value might change between this procedure's returning the value and your checking it, if some other thread inserts an element into the queue. To use the value reliably, you need another mutex to restrict putting items in the queue.

```
(define q (make-mtqueue))
```

```
(thread-start! (make-thread (^[] (dequeue/wait! q))))

(mtqueue-num-waiting-readers q) ⇒ 1

(enqueue! q 'a)

(mtqueue-num-waiting-readers q) ⇒ 0
```

`copy-queue` *queue* [Function]
{`data.queue`} Returns a copy of the queue.

`enqueue!` *queue obj :optional more-objs . . .* [Function]
{`data.queue`} Add *obj* to the end of *queue*. You may give more than one object, and each of them are enqueued in order.

If *queue* is an mtqueue, all the objects are enqueued atomically; no other objects from other threads can be inserted between the objects given to a single `enqueue!` call. Besides, if the value of its `max-length` slot has a positive finite value, and adding *obj*s makes the number of elements in *queue* exceeds `max-length`, an error is signaled and *queue* won't be modified. (If `max-length` is zero, this procedure always fail. Use `enqueue/wait!` below.)

`queue-push!` *queue obj :optional more-objs . . .* [Function]
{`data.queue`} Add *obj* in front of *queue*. You may give more than one object, and each of them are pushed in order.

Like `enqueue!`, when *queue* is an mtqueue, all objects are added atomically, and the value of `max-length` slot is checked. See `enqueue!` above for the details.

`enqueue-unique!` *queue eq-proc obj :optional more-objs . . .* [Function]
`queue-push-unique!` *queue eq-proc obj :optional more-objs . . .* [Function]
{`data.queue`} Like `enqueue!` and `queue-push!`, respectively, except that these don't modify *queue* if it already contains *obj* (elements are compared by two-argument procedure *eq-proc*).

When *queue* is an mtqueue, all objects are added atomically, and the value of `max-length` slot is checked. See `enqueue!` above for the details.

`dequeue!` *queue :optional fallback* [Function]
`queue-pop!` *queue :optional fallback* [Function]
{`data.queue`} Take one object from the front of the queue *queue* and returns it. Both function works the same, but `queue-pop!` may be used to emphasize it works with `queue-push!`.

If *queue* is empty, *fallback* is returned if given, otherwise an error is signaled.

If *queue* is an mtqueue and its `max-length` is zero, the queue is always empty. Use `dequeue/wait!` to use such a queue as an synchronization device.

`dequeue-all!` *queue* [Function]
{`data.queue`} Returns the whole content of the queue by a list, with emptying *queue*. If *queue* is already empty, returns an empty list. See also `queue->list` below.

`queue-front` *queue :optional fallback* [Function]
`queue-rear` *queue :optional fallback* [Function]
{`data.queue`} Peek the head or the tail of the queue and returns the object, respectively. The queue itself is not modified. If *queue* is empty, *fallback* is returned if it is given, otherwise an error is signaled.

`list->queue` *list :optional class :rest initargs* [Function]
{`data.queue`} Returns a new queue whose content is the elements in *list*, in the given order.

By default the created queue is a simple queue, but you can create mtqueue or instances of other subclasses of `<queue>` by giving the class to the optional *class* arguments. The optional *initargs* arguments are passed to the constructor of *class*.

`queue->list` *queue* [Function]

{`data.queue`} Returns a list whose content is the items in the queue in order. Unlike `dequeue-all!`, the content of *queue* remains intact.

In Gauche, `queue->list` copies the content of the queue to a freshly allocated list, while `dequeue-all!` doesn't copy but directly returns the queue's internal list. There are some Scheme systems that has `queue->list` but doesn't guarantee the content is copied, so if you're planning to share the code among these implementations, it's better not to rely on the fact that `queue->list` copies the content.

`queue-internal-list` *queue* [Function]

{`data.queue`} Like `queue->list`, returns a list whose content is the items in the queue in order, but the returned list *may* share the internal storage of *queue*. The returned list can be modified by subsequent operations of *queue*, and any modification on the list can make *queue* inconsistent.

Because of this danger, we don't allow `<mtqueue>` to be passed to this procedure; it would signal an error if you do so.

If you just want to extract the accumulated result in *queue* without copying, consider `dequeue-all!`, which is safe because it atomically resets the queue. Use this procedure only when you absolutely need to access the contents of the queue without taking them out.

`find-in-queue` *pred queue* [Function]

{`data.queue`} Returns the first item in *queue* that satisfies a predicate *pred*. The order of arguments follows `find` (see Section 6.6.6 [Other list procedures], page 134).

`any-in-queue` *pred queue* [Function]

{`data.queue`} Like `any` in SRFI-1, apply *pred* on each item in *queue* until it evaluates true, and returns that true value (doesn't necessarily be `#t`). If no items in the queue satisfies *pred*, `#f` is returned.

`every-in-queue` *pred queue* [Function]

{`data.queue`} Like `every` in SRFI-1, apply *pred* on each item in *queue*. If *pred* returns `#f`, stops iteration and returns `#f` immediately. Otherwise, returns the result of the application of *pred* on the last item of the queue. If the queue is empty, `#t` is returned.

`remove-from-queue!` *pred queue* [Function]

{`data.queue`} Removes all items in the queue that satisfies *pred*. Returns `#t` if any item is removed. Otherwise returns `#f`. The order of arguments follows `remove` in `scheme.list` (see Section 10.3.1 [R7RS lists], page 512).

Note on portability: Scheme48 has `delete-from-queue!`, which takes object to remove rather than predicate, and also takes arguments in reversed order (i.e. queue comes first). Avoid conflicting with that I intentionally left out `delete-from-queue!`; it's easy to write one in either Scheme48 compatible way or consistent to SRFI-1 argument order.

`enqueue/wait!` *mtqueue obj :optional timeout timeout-val* [Function]
`queue-push/wait!` *mtqueue obj :optional timeout timeout-val* [Function]
`dequeue/wait!` *mtqueue :optional timeout timeout-val* [Function]
`queue-pop/wait!` *mtqueue :optional timeout timeout-val* [Function]

{`data.queue`} These synchronizing variants work on an mtqueue and make the caller thread block when the mtqueue has reached its maximum length (for `enqueue/wait!` and queue-push/wait!), or the mtqueue is empty (for dequeue/wait! and queue-pop/wait!). The blocked caller thread is unblocked either when the blocking condition is resolved, or the timeout condition is met.

The optional *timeout* argument specifies the timeout condition. If it is #f, those procedures wait indefinitely. If it is a real number, they wait at least the given number of seconds. If it is a <time> object (see Section 6.25.9 [Time], page 269), they wait until the absolute point of time the argument specifies.

In case the call is blocked then timed out, the value of *timeout-val* is returned, which defaults to #f.

When enqueue/wait! and queue-push/wait! succeeds without hitting timeout, they return #t.

12.13 data.random - Random data generators

data.random [Module]

This module defines a set of generators and generator makers that yield random data of specific type and distribution.

A naming convention: Procedures that takes parameters and returns a generator is suffixed by $ (e.g. integer$). Procedures that are generators themselves are not (e.g. fixnums). Procedures that are combinators, that is, the ones that take one or more generators and returns a generator, generally ends with a preposition (e.g. list-of).

Global state

All the generators in this module shares a global random state. The random seed is initialized by a fixed value when the module is loaded. You can get and set the random seed by the following procedure.

random-data-seed [Function]
(setter random-data-seed) *seed-value* [Function]

{data.random} Calling random-data-seed (without arguments) returns the random seed value used to initialize the current random state.

It can be used with generic setter, to reinitialize the random state with *seed-value*.

Random seed value must be an exact integer. Its lower 32bits are used.

```
; reinitialize the random state with a new random seed.
(set! (random-data-seed) 1)

(random-data-seed) ⇒ 1
```

Note: This procedure doesn't have parameter interface (alter the global value by giving the new value as an argument), since it doesn't work like a parameter (see Section 9.22 [Parameters], page 411). You can get the random seed value, but you can't get the current random state itself—if you restore the random seed value again, the internal state is reset, instead of restoring the state at the time you called random-data-seed.

If you want to use different random state temporarily, and ensure to restore original state afterwards, use with-random-data-seed below.

with-random-data-seed *seed thunk* [Function]

{data.random} Saves the current global random state, initializes the random state with *seed*, then executes *thunk*. If *thunk* returns or the control exits out of *thunk*, the state at the time with-random-data-seed was called is restored.

Since the default random seed value is fixed, you can get deterministic output when you call the random data generators below without altering the random seed explicitly.

Generators of primitive data types

Those generators generate uniformly distributed data.

In the following examples, we use `generator->list` to show some concrete data from the generators. It is provided in `gauche.generator` module. See Section 9.11 [Generators], page 372, for more utilities work on generators.

`integers$` *size :optional* (*start 0*) [Function]
`integers-between$` *lower-bound upper-bound* [Function]

{`data.random`} Create exact integer generators. The first one, `integers$`, creates a generator that generates integers from start (inclusive) below start+size (exclusive) uniformly. The second one, `integers-between$`, creates a generator that generates integers between *lower-bound* and *upper-bound* (both inclusive) uniformly.

```
;; A dice roller
(define dice (integers$ 6 1))

;; Roll the dice 10 times
(generator->list dice 10)
 ⇒ (6 6 2 4 2 5 5 1 2 2)
```

`fixnums` [Function]
`int8s` [Function]
`uint8s` [Function]
`int16s` [Function]
`uint16s` [Function]
`int32s` [Function]
`uint32s` [Function]
`int64s` [Function]
`uint64s` [Function]

{`data.random`} Uniform integer generators. Generate integers in fixnum range, and 8/16/32/64bit signed and unsigned integers, respectively.

```
(generator->list int8s 10)
 ⇒ (20 -101 50 -99 -111 -28 -19 -61 39 110)
```

`booleans` [Function]

{`data.random`} Generates boolean values (`#f` and `#t`) in equal probability.

```
(generator->list booleans 10)
 ⇒ (#f #f #t #f #f #t #f #f #f #f)
```

`chars$` *:optional char-set* [Function]

{`data.random`} Creates a generator that generates characters in *char-set* uniformly. The default *char-set* is `#[A-Za-z0-9]`.

```
(define alphanumeric-chars (chars$))

(generator->list alphanumeric-chars 10)
 ⇒ (#\f #\m #\3 #\S #\z #\m #\x #\S #\l #\y)
```

`reals$` *:optional size start* [Function]
`reals-between$` *lower-bound upper-bound* [Function]

{`data.random`} Create a generator that generates real numbers uniformly with given range. The first procedure, `reals$`, returns reals between start and start+size, inclusively. The default of size is 1.0 and start is 0.0. The second procedure, `reals-between$`, returns reals between lower-bound and upper-bound, inclusively.

```
(define uniform-100 (reals$ 100))
```

```
(generator->list uniform-100 10)
 ⇒ (81.67965004942268 81.84927577572596 53.02443813660833)
```

Note that a generator from `reals$` can generate the upper-bound value start+size, as opposed to `integers$`. If you need to exclude the bound value, just discard the bound value; `gfilter` may come handy.

```
(define generate-from-0-below-1
  (gfilter (^r (not (= r 1.0))) (reals$ 1.0 0.0)))
```

`samples$` *collection* [Function]

{`data.random`} Creates a generator that returns randomly chosen item in *collection* at a time.

Do not confuse this with `samples-from` below, which is to combine multiple generators for sampling.

```
(define coin-toss (samples$ '(head tail)))

(generator->list coin-toss 5)
 ⇒ (head tail tail head tail)
```

`regular-string$` *regexp* [Function]

{`data.random`} Creates an infinite generator that generates random strings each of which matches the given *regexp*. The *regexp* shouldn't include conditional patterns and lookahead/behind assertions.

Note: It is hard to define how the distribution of the generated strings should look like. For now, we build an NFA from regexp and put the same probability when there are multiple choices, but that may not be really useful for typical use cases (e.g. generate test data). Please assume the current implementation strategy a provisional one.

Nonuniform distributions

`reals-normal$` *:optional mean deviation* [Function]

{`data.random`} Creates a generator that yields real numbers from normal distribution with *mean* and *deviation*. The default of *mean* is 0.0 and *deviation* is 1.0.

`reals-exponential$` *mean* [Function]

{`data.random`} Creates a generator that yields real numbers from exponential distribution with *mean*.

`integers-geometric$` *p* [Function]

{`data.random`} Creates a generator that yields integers from geometric distribution with success probability *p* (0 <= p <= 1). The mean is `1/p` and variance is `(1-p)/p^2`.

`integers-poisson$` *L* [Function]

{`data.random`} Creates a generator that yields integers from poisson distribution with mean *L*, variance *L*.

Aggregate data generators

`samples-from` *generators* [Function]

{`data.random`} Takes a finite sequence of generators (sequence in the sense of `gauche.sequence`), and returns a generator. Every time the resulting generator is called, it picks one of the input generators in equal probability, then calls it to get a value.

```
(define g (samples-from (list uint8s (chars$ #[a-z]))))
```

```
(generator->list g 10)
 ⇒ (207 107 #\m #\f 199 #\o #\b 57 #\j #\e)
```

NB: To create a generator that samples from a fixed collection of items, use `samples$` described above.

`weighted-samples-from` *weight*&**gens** [Function]

{`data.random`} The argument is a list of pairs of a nonnegative real number and a generator. The real number determines the weight, or the relative probability that the generator is chosen. The sum of weight doesn't need to be 1.0.

The following example chooses the uint8 generator four times frequently than the character generator.

```
(define g (weighted-samples-from
           `((4.0 . ,uint8s)
             (1.0 . ,(chars$)))))

(generator->list g 10)
 ⇒ (195 97 #\j #\W #\5 72 49 143 19 164)
```

`pairs-of` *car-gen cdr-gen* [Function]

{`data.random`} Returns a generator that yields pairs, whose car is generated from *car-gen* and whose cdr is generated from *cdr-gen*.

```
(define g (pairs-of int8s booleans))

(generator->list g 10)
 ⇒ ((113 . #t) (101 . #f) (12 . #t) (68 . #f) (-55 . #f))
```

`tuples-of` *gen* ... [Function]

{`data.random`} Returns a generator that yields lists, whose i-th element is generated from the i-th argument.

```
(define g (tuples-of int8s booleans (char$)))

(generator->list g 3)
 ⇒ ((-43 #f #\8) (53 #f #\1) (-114 #f #\i))
```

`permutations-of` *seq* [Function]

{`data.random`} Returns a generator that yields a random permutations of *seq*.

The type of *seq* should be a sequence with a builder (see Section 9.29 [Sequence framework], page 441). The type of generated objects will be the same as *seq*.

```
(generator->list (permutations-of '(1 2 3)) 3)
 ⇒ ((1 2 3) (2 3 1) (3 2 1))

(generator->list (permutations-of "abc") 3)
 ⇒ ("cba" "cba" "cab")
```

`combinations-of` *size seq* [Function]

{`data.random`} Returns a generator that yields a sequence of *size* elements randomly picked from *seq*.

The type of *seq* should be a sequence with a builder (see Section 9.29 [Sequence framework], page 441). The type of generated objects will be the same as *seq*.

```
(generator->list (combinations-of 2 '(a b c)) 5)
```

```
 ⇒ ((a c) (a b) (a c) (b a) (a c))

(generator->list (combinations-of 2 '#(a b c)) 5)
 ⇒ (#(a c) #(b c) #(c b) #(b a) #(b c))
```

The following procedures takes optional *sizer* argument, which can be either a nonnegative integer or a generator of nonnegative integers. The value of the sizer determines the length of the result data.

Unlike most of Gauche procedures, *sizer* argument comes before the last argument when it is not omitted. We couldn't resist the temptation to write something like `(lists-of 3 booleans)`.

If *sizer* is omitted, the default value is taken from the parameter `default-sizer`. The default of `default-sizer` is `(integers-poisson$ 4)`.

`lists-of` *item-gen* [Function]
`lists-of` *sizer item-gen* [Function]
`vectors-of` *item-gen* [Function]
`vectors-of` *sizer item-gen* [Function]
`strings-of` [Function]
`strings-of` *item-gen* [Function]
`strings-of` *sizer item-gen* [Function]

{`data.random`} Creates a generator that generates lists, vectors or strings of values from *item-gen*, respectively. The size of each datum is determined by *sizer*.

You can also omit *item-gen* for `strings-of`. In that case, a generator created by `(chars$)` is used.

```
(generator->list (lists-of 3 uint8s) 4)
 ⇒ ((254 46 0) (77 158 46) (1 134 156) (74 5 110))

;; using the default sizer
(generator->list (lists-of uint8s) 4)
 ⇒ ((93 249) (131 97) (98 206 144 247 241) (126 156 31))

;; using a generator for the sizer
(generator->list (strings-of (integers$ 8) (chars$)) 5)
 ⇒ ("dTJYVhu" "F" "PXkC" "w" "")
```

`sequences-of` *class item-gen* [Function]
`sequences-of` *class sizer item-gen* [Function]

{`data.random`} Creates a generator that yields sequences of class *class*, whose items are generated by *item-gen*. The size of each sequence is determined by *sizer*, or the value of `default-sizer` if omitted; the sizer can be a nonnegative integer, or a generator that yields nonnegative integers.

The class *class* must be a subclass of `<sequence>` and implement the builder interface.

```
(generator->list (sequences-of <u8vector> 4 uint8s) 3)
 ⇒ (#u8(95 203 243 46) #u8(187 199 153 152) #u8(39 114 39 25))
```

`default-sizer` [Parameter]

{`data.random`} The sizer used by `lists-of`, `vectors-of` and `strings-of` when *sizer* argument is omitted.

The value must be either an nonnegative integer, or a generator of nonnegative integers.

12.14 `data.ring-buffer` - Ring buffer

`data.ring-buffer` [Module]

A ring buffer is an array with two fill pointers; in a typical usage, a producer adds new data to one end while a consumer removes data from the other end; if fill pointer reaches at the end of the array, it wraps around to the beginning, hence the name.

The ring buffer of this module allows adding and removing elements from both ends, hence functionally it is a double-ended queue, or deque. It also allows O(1) indexed access to the contents, and customized handling for the case when the buffer gets full.

You can use an ordinary vector or a uniform vector as the backing storage of a ring buffer.

`make-ring-buffer` *:optional initial-storage :key overflow-handler* [Function]

{`data.ring-buffer`} Creates a ring buffer. By default, a fresh vector is allocated for the backing storage. You can pass a vector or a uvector to *initial-storage* to be used instead. The passed storage must be mutable, and will be modified by the ring buffer; the caller shouldn't modify it, nor make assumption about its content.

The *overflow-handler* keyword argument specifies what to do when a new element is about to be added to the full buffer. It must be a procedure, or a symbol `error` or `overwrite`.

If it is a procedure, it will be called with a ring buffer and a backing storage (vector or uvector) when it is filled. The procedure must either (1) allocate and return a larger vector/uvector of the same type of the passed backing storage, (2) return a symbol `error`, or (3) return a symbol `overwrite`. If it returns a vector/uvector, it will be used as the new backing storage. The returned vector doesn't need to be initialized; the ring buffer routine takes care of it. If it returns `error`, an error ("buffer is full") is thrown. If it returns `overwrite`, the new element overwrites the existing element (as if one element from the other end is popped and discarded.)

Passing a symbol `error` or `overwrite` to *overflow-handler* is a shorthand of passing a procedure that unconditionally returns *error* or `overwrite`, respectively.

The default behavior on overflow is to double the size of backing storage. You can use `make-overflow-doubler` below to create the customized overflow handler easily.

`make-overflow-doubler` *:key max-increase max-capacity* [Function]

{`data.ring-buffer`} Returns a procedure suitable to be passed to the *overflow-handler* keyword argument of `make-ring-buffer`.

The returned procedure takes a ring buffer and its backing storage, and behaves as follows.

- If the size of current backing storage is equal to or greater than *max-capacity*, returns `error`.
- Otherwise, if the size of current backing storage is equal to or greater than *max-increase*, allocates a vector/uvector of the same type of the current backing storage, with the size `(+ max-increase size-of-current-storage)`.
- Otherwise, allocates a vector/uvector of the same type of the current backing storage with the size `(* 2 size-of-current-storage)`.

The default value of *max-increase* and *max-capacity* is `+inf.0`.

`ring-buffer-empty?` *rb* [Function]

{`data.ring-buffer`} Returns `#t` if the ring buffer *rb* is empty, `#f` if not.

`ring-buffer-full?` *rb* [Function]

{`data.ring-buffer`} Returns `#t` if the ring buffer *rb* is full, `#f` if not.

`ring-buffer-num-entries` *rb* [Function]

{`data.ring-buffer`} Returns the number of current elements in the ring buffer *rb*.

`ring-buffer-capacity` *rb* [Function]

{`data.ring-buffer`} Returns the size of the current backing storage of the ring buffer *rb*.

`ring-buffer-front` *rb* [Function]
`ring-buffer-back` *rb* [Function]

{`data.ring-buffer`} Returns the element in the front or back of the ring buffer *rb*, respectively. If the buffer is empty, an error is signaled.

`ring-buffer-add-front!` *rb elt* [Function]
`ring-buffer-add-back!` *rb elt* [Function]

{`data.ring-buffer`} Add an element to the front or back of the ring buffer *rb*, respectively. If *rb* is full, the behavior is determined by the buffer's overflow handler, as described in `make-ring-buffer`.

`ring-buffer-remove-front!` *rb* [Function]
`ring-buffer-remove-back!` *rb* [Function]

{`data.ring-buffer`} Remove an element from the front or back of the ring buffer *rb*, and returns the removed element, respectively. If the buffer is empty, an error is signaled.

`ring-buffer-ref` *rb index :optional fallback* [Function]

{`data.ring-buffer`} Returns *index*-th element in the ring buffer *rb*. The elements are counted from the front; thus, if a new element is added to the front, the indexes of existing elements will shift.

If the index out of bounds of the existing content, *fallback* will be returned; if *fallback* is not provided, an error is signaled.

`ring-buffer-set!` *rb index value* [Function]

{`data.ring-buffer`} Sets *index*-th element of the ring buffer *rb* to *value*. The elements are counted from the front; thus, if a new element is added to the front, the indexes of existing elements will shift.

An error is signaled if the index is out of bounds.

12.15 `data.sparse` - Sparse data containers

`data.sparse` [Module]

This module provides a *sparse vector* and *sparse matrix*, a space efficient data container indexed by nonnegative integer(s), and a *sparse table*, a hash table using a sparse vector as a backing storage.

A sparse vector associates a nonnegative integer index to a value. It has *vector* in its name since it is indexed by an integer, but it isn't like a flat array on contiguous memory; it's more like an associative array. (Internally, the current implementation uses compact trie structure.) It is guaranteed that you can store a value with index at least up to `2^32-1`; the actual maximum bits of indexes can be queried by `sparse-vector-max-index-bits`. (We have a plan to remove the maximum bits limitation in future).

Unlike ordinary vectors, you don't need to specify the size of a sparse vector when you create one. You can just set a value to any index in the supported range.

```
(define v (make-sparse-vector))

(sparse-vector-set! v 0 'a)
(sparse-vector-ref v 0) ⇒ a

(sparse-vector-set! v 100000000 'b)
```

```
(sparse-vector-ref v 100000000) ⇒ b

;; set! also work
(set! (sparse-vector-ref v 100) 'c)
(sparse-vector-ref v 100) ⇒ c
```

If you try to access an element that hasn't been set, an error is signaled by default. You can set a default value for each vector, or give a fallback value to `sparse-vector-ref`, to suppress the error.

```
(sparse-vector-ref v 1)        ⇒ error
(sparse-vector-ref v 1 'noval) ⇒ noval

(let1 w (make-sparse-vector #f :default 'x)
  (sparse-vector-ref w 1))     ⇒ x
```

A sparse matrix is like a sparse vector, except it can be indexed by a pair of integers.

A sparse table works just like a hash table, but it uses a sparse vector to store the values using hashed number of the keys.

The main reason of these sparse data containers are for memory efficiency. If you want to store values in a vector but knows you'll use only some entries sparsely, obviously it is waste to allocate a large vector and to leave many entries unused. But it is worse than that; Gauche's GC doesn't like a large contiguous region of memory. Using lots of large vectors adds GC overhead quickly. It becomes especially visible when you store large number of entries (like >100,000) into hash tables, since Gauche's builtin hash tables use a flat vector as a backing storage. You'll see the heap size grows quickly and GC runs more frequently and longer. On the other hand, sparse table works pretty stable with large number of entries.

Sparse data containers does have overhead on access speed. They are a bit slower than the ordinary hash tables, and much slower than ordinary vectors. We should note, however, as the number of entries grow, access time on ordinary hash tables grows quicker than sparse tables and eventually two become comparable.

It depends on your application which you should use, and if you're not sure, you need to benchmark. As a rule of thumb, if you use more than several hashtables each of which contains more than a few tens of thousands of entries, sparse tables may work better. If you see GC Warnings telling "repeated allocation of large blocks", you should definitely consider sparse tables.

12.15.1 Sparse vectors

`<sparse-vector-base>` [Class]

{`data.sparse`} An abstract base class of sparse vectors. Inherits `<dictionary>` and `<collection>`. Note that sparse vectors are *not* `<sequence>`; even they can be indexable by integers, they don't have means of *ordered* access.

Sparse vector may be a general vector that can contain any Scheme objects (like `<vector>`), or a specialized vector that can contain only certain types of numbers (like `<s8vector>` etc.).

All of these sparse vectors can be accessed by the same API.

Sparse vectors also implements the Collection API (see Section 9.5 [Collection framework], page 344) and the Dictionary API (see Section 9.9 [Dictionary framework], page 366).

`<sparse-vector>` [Class]
`<sparse-@vector>` [Class]

{`data.sparse`} The actual sparse vector classes. Inherits `<sparse-vector-base>`. An instance of `<sparse-vector>` can contain any Scheme objects.

`@` is either one of `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f16`, `f32`, or `f64`. The range of values an instance of those classes can hold is the same as the corresponding `<@vector>` class in `gauche.uvector` (see Section 9.36 [Uniform vectors], page 476). That is, `<sparse-u8vector>` can have exact integer values between 0 and 255.

`make-sparse-vector` *:optional type :key default* [Function]
{`data.sparse`} Creates an empty sparse vector. The *type* argument can be `#f` (default), one of subclasses of `<sparse-vector-base>`, or a symbol of either one of `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f16`, `f32`, or `f64`.

If *type* is omitted or `#f`, a `<sparse-vector>` is created. If it is a class, an instance of the class is created (It is an error to pass a class that is not a subclass of `<sparse-vector-base>`.) If it is a symbol, an instance of corresponding `<sparse-@vector>` is created.

You can specify the default value of the vector by *default* keyword argument. If given, the vector behaves as if it is filled with the default value (but the vector iterator only picks the values explicitly set).

Note that you have to give the optional argument as well to specify the keyword argument.

```
(define v (make-sparse-vector 'u8 :default 128))

(sparse-vector-ref v 0) ⇒ 128
```

`sparse-vector-max-index-bits` [Function]
{`data.sparse`} Returns maximum number of bits of allowed integer. If this returns 32, the index up to `(expt 2 32)` is supported. It is guaranteed that this is at least 32.

In the following entries, the argument *sv* denotes an instance of sparse vector; an error is signaled if other object is passed.

`sparse-vector-copy` *sv* [Function]
{`data.sparse`} Returns a copy of a sparse vector *sv*.

`sparse-vector-ref` *sv k :optional fallback* [Function]
{`data.sparse`} Returns *k*-th element of a sparse vector *sv*, where *k* must an exact integer.

If the sparse vector doesn't have a value for *k*, it behaves as follows:

- If *fallback* is given, it is returned.
- Otherwise, if the vector has the default value, it is returned.
- Otherwise, an error is signaled.

`sparse-vector-set!` *sv k value* [Function]
{`data.sparse`} Sets *value* for *k*-th element of a sparse vector *sv*. *K* must be a nonnegative exact integer, and below the maximum allowed index.

If *sv* is a numeric sparse vector, *value* must also be within the allowed range, or an error is signaled.

`sparse-vector-num-entries` *sv* [Function]
{`data.sparse`} Returns the number of entries in *sv*.

`sparse-vector-exists?` *sv k* [Function]
{`data.sparse`} Returns `#t` if *sv* has an entry for index *k*, `#f` otherwise.

`sparse-vector-delete!` *sv k* [Function]
{`data.sparse`} Deletes the *k*-th entry of *sv*. If *sv* had the entry , returns `#t`. If *sv* didn't have the entry, returns `#f`.

`sparse-vector-clear!` *sv* [Function]
{data.sparse} Empties a sparse vector.

`sparse-vector-inc!` *sv k delta :optional* (*fallback 0*) [Function]
{data.sparse} This is a shortcut of the following. It is especially efficient for numeric sparse vectors.

```
(sparse-vector-set! sv k (+ (sparse-vector-ref sv k fallback) delta))
```

If the result of addition exceeds the allowed value range of *sv*, an error is signaled. In future we'll allow an option to clamp the result value within the range.

`sparse-vector-update!` *sv k proc :optional fallback* [Function]
`sparse-vector-push!` *sv k val* [Function]
`sparse-vector-pop!` *sv k :optional fallback* [Function]
{data.sparse} Convenience routines to fetch-and-update an entry of a sparse vector. Works just like `hash-table-update!`, `hash-table-push!` and `hash-table-pop!`; (see Section 6.15 [Hashtables], page 177).

The following procedures traverses a sparse vector. Note that elements are not visited in the order of index; it's just like hash table traversers.

At this moment, if you want to walk a sparse vector with increasing/decreasing index order, you have to get a list of keys by `sparse-vector-keys`, sort it, then use it to retrieve values. We may add an option in future to `make-sparse-vector` so that those walk operation will be more convenient.

`sparse-vector-fold` *sv proc seed* [Function]
{data.sparse} For each entry in *sv*, calls *proc* as `(proc k_n v_n seed_n)`, where *k_n* is an index and *v_n* is a value for it, and *seed_n* is the returned value of the previous call to *proc* if *n* `>=` 1, and *seed* if $n = 0$. Returns the value of the last call of *proc*.

`sparse-vector-for-each` *sv proc* [Function]
`sparse-vector-map` *sv proc* [Function]
{data.sparse} Calls `proc` with index and value, e.g. `(proc k value)`, for each element of *sv*.

The results of *proc* are discarded by `sparse-vector-for-each`, and gathered to a list and returned by `sparse-vector-map`.

`sparse-vector-keys` *sv* [Function]
`sparse-vector-values` *sv* [Function]
{data.sparse} Returns a list of all keys and all values in *sv*, respectively.

12.15.2 Sparse matrixes

A sparse matrix is like a sparse vector, except it can be indexed by two nonnegative integers.

Note: This implementation of sparse matrixes aims at a reasonable space efficiency for sparse matrixes without knowing its structure beforehand (imagine, for example, a 2D map with some scattered landmarks). If what you want is a sparse matrix implementation for efficient numeric calculations, with certain particular structures, probably the access speed of this module isn't suitable.

Currently, each index can have half of bits of `sparse-vector-max-index-bits`. We'll remove this limitation in future.

`<sparse-matrix-base>` [Class]
{data.sparse} An abstract base class of sparse matrixes. Inherits `<collection>`.

Like sparse vectors, a sparse matrix can be of type that can store any Scheme objects, or that can store only certain types of numbers.

All of these sparse matrix subtypes can be accessed by the same API.

`<sparse-matrix>` [Class]
`<sparse-@matrix>` [Class]

{`data.sparse`} The actual sparse matrix classes. Inherits `<sparse-matrix-base>`. An instance of `<sparse-matrix>` can contain any Scheme objects.

`@` is either one of `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f16`, `f32`, or `f64`. The range of values an instance of those classes can hold is the same as the corresponding `<@vector>` class in `gauche.uvector` (see Section 9.36 [Uniform vectors], page 476). That is, `<sparse-u8matrix>` can have exact integer values between 0 and 255.

`make-sparse-matrix` *:optional type :key default* [Function]

{`data.sparse`} Creates an empty sparse matrix. The *type* argument can be `#f` (default), one of subclasses of `<sparse-matrix-base>`, or a symbol of either one of `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f16`, `f32`, or `f64`.

If *type* is omitted or `#f`, a `<sparse-matrix>` is created. If it is a class, an instance of the class is created (It is an error to pass a class that is not a subclass of `<sparse-matrix-base>`.) If it is a symbol, an instance of corresponding `<sparse-@matrix>` is created.

You can specify the default value of the matrix by *default* keyword argument. If given, the vector behaves as if it is filled with the default value (but the matrix iterator only picks the values explicitly set).

Note that you have to give the optional argument as well to specify the keyword argument.

`sparse-matrix-num-entries` *mat* [Function]

{`data.sparse`} Returns the number of entries explicitly set in a sparse matrix *mat*.

`sparse-matrix-ref` *mat x y :optional fallback* [Function]

{`data.sparse`} Returns an element indexed by (x, y) in a sparse matrix *mat*. If the indexed element isn't set, *fallback* is returned if provided; otherwise, if the matrix has the default value, it is returned; otherwise, an error is raised.

`sparse-matrix-set!` *mat x y value* [Function]

{`data.sparse`} Set *value* to the sparse matrix *mat* at the location (x, y).

`sparse-matrix-exists?` *mat x y* [Function]

{`data.sparse`} Returns `#t` iff the sparse matrix *mat* has a value at (x, y).

`sparse-matrix-clear!` *mat* [Function]

{`data.sparse`} Empties the sparse matrix *mat*.

`sparse-matrix-delete!` *mat x y* [Function]

{`data.sparse`} Remove the value at (x, y) from the sparse matrix *mat*.

`sparse-matrix-copy` *mat* [Function]

{`data.sparse`} Returns a fresh copy of *mat*.

`sparse-matrix-update!` *mat x y proc :optional fallback* [Function]

{`data.sparse`} Call *proc* with the value at (x, y) of the sparse matrix, and sets the result of *proc* as the new value of the location.

The optional *fallback* argument works just like `sparse-matrix-ref`; if provided, it is passed to *proc* in case the matrix doesn't have a value at (x, y). If *fallback* isn't provided and the matrix doesn't have a value at the location, the default value of the matrix is used if it has one. Otherwise, an error is signalled.

`sparse-matrix-inc!` *mat x y delta :optional fallback* [Function]
{`data.sparse`}

```
(sparse-matrix-update! mat x y (cut + <> delta) fallback)
```

`sparse-matrix-push!` *mat x y val* [Function]
{`data.sparse`}

```
(sparse-matrix-update! mat x y (cut cons val <>) '())
```

`sparse-matrix-pop!` *mat x y* [Function]
{`data.sparse`}

```
(rlet1 r #f
  (sparse-matrix-update! mat x y (^p (set! r (car p)) (cdr p))))
```

`sparse-matrix-fold` *mat proc seed* [Function]
{`data.sparse`} Loop over values in the sparse matrix *mat*. The procedure *proc* is called with four arguments, *x*, *y*, *val* and *seed*, for each index (*x*, *y*) which has the value *val*. The initial value of *seed* is the one given to `sparse-matrix-fold`, and the result of *proc* is passed as the next seed value. The last result of *proc* is returned from `sparse-matrix-fold`.

The procedure *proc* is only called on the entries that's actually has a value, and the order of which the procedure is called is undefined.

`sparse-matrix-map` *mat proc* [Function]
{`data.sparse`}

```
(sparse-matrix-fold sv (^[x y v s] (cons (proc x y v) s)) '()))
```

`sparse-matrix-for-each` *mat proc* [Function]
{`data.sparse`}

```
(sparse-matrix-fold sv (^[x y v _] (proc x y v)) #f))
```

`sparse-matrix-keys` *mat* [Function]
{`data.sparse`}

```
(sparse-matrix-fold sv (^[x y _ s] (cons (list x y) s)) '())
```

`sparse-matrix-values` *mat* [Function]
{`data.sparse`}

```
(sparse-matrix-fold sv (^[x y v s] (cons v s)) '())
```

12.15.3 Sparse tables

`<sparse-table>` [Class]
{`data.sparse`} A class for sparse table. Inherits `<dictionary>` and `<collection>`.

Operationally sparse tables are the same as hash tables, but the former consumes less memory in trade of slight slower access. (Roughly x1.5 to x2 access time when the table is small. As the table gets larger the difference becomes smaller.)

`make-sparse-table` *comparator* [Function]
{`data.sparse`} Creates and returns an empty sparse table. The *comparator* argument specifies how to compare and hash keys; it must be either a comparator (see Section 6.2.4 [Basic comparators], page 103), or one of the symbols `eq?`, `eqv?`, `equal?` and `string=?`, like hash tables (see Section 6.15 [Hashtables], page 177). If it is a symbol, `eq-comparator`, `eqv-comparator`, `equal-comparator` or `string-comparator` are used, respectively.

`sparse-table-comparator` *st* [Function]
{`data.sparse`} Returns the comparator used in the sparse table *st*.

`sparse-table-copy` *st* [Function]
{`data.sparse`} Returns a copy of a sparse table *st*.

`sparse-table-num-entries` *st* [Function]
{`data.sparse`} Returns the number of entries in a sparse table *st*.

`sparse-table-ref` *st key :optional fallback* [Function]
{`data.sparse`} Retrieves a value associated to the *key* in *st*. If no entry with *key* exists, *fallback* is returned when it is provided, or an error is signaled otherwise.

`sparse-table-set!` *st key value* [Function]
{`data.sparse`} Sets *value* with *key* in *st*.

`sparse-table-exists?` *st key* [Function]
{`data.sparse`} Returns `#t` if an entry with *key* exists in *st*, `#f` otherwise.

`sparse-table-delete!` *st key* [Function]
{`data.sparse`} Deletes an entry with *key* in *st* if it exists. Returns `#t` if an entry is actually deleted, or `#f` if there hasn't been an entry with *key*.

`sparse-table-clear!` *st* [Function]
{`data.sparse`} Empties *st*.

`sparse-table-update!` *st key proc :optional fallback* [Function]
`sparse-table-push!` *st key val* [Function]
`sparse-table-pop!` *st key :optional fallback* [Function]
{`data.sparse`}

`sparse-table-fold` *st proc seed* [Function]
`sparse-table-for-each` *st proc* [Function]
`sparse-table-map` *st proc* [Function]
{`data.sparse`}

`sparse-table-keys` *st* [Function]
`sparse-table-values` *st* [Function]
{`data.sparse`}

12.16 `data.trie` - Trie

`data.trie` [Module]
This module provides *Trie*, a dictionary-like data structure that maps keys to values, where a key is an arbitrary sequence. Internally it stores the data as a tree where each node corresponds to each element in the key sequence. Key lookup is O(n) where n is the length of the key, and not affected much by the number of total entries. Also it is easy to find a set of values whose keys have a common prefix.

The following example may give you the idea.

```
(define t (make-trie))   ;; create a trie

(trie-put! t "pho" 3)    ;; populate the trie
(trie-put! t "phone" 5)
(trie-put! t "phrase" 6)

(trie-get t "phone")  ⇒ 5  ;; lookup
```

```
(trie-common-prefix t "pho")       ;; common prefix search
  ⇒ (("phone" . 5) ("pho" . 3))
(trie-common-prefix-keys t "ph")
  ⇒ ("phone" "pho" "phrase")
```

Tries are frequently used with string keys, but you are not limited to do so; any sequence (see Section 9.29 [Sequence framework], page 441) can be a key. If the types of keys differ, they are treated as different keys:

```
(trie-put! t '(#\p #\h #\o) 8)  ;; different key from "pho"
```

Trie inherits `<collection>` and implements collection framework including the builder. So you can apply generic collection operations on a trie (see Section 9.5 [Collection framework], page 344). When iterated, each element of a trie appears as a pair of a key and a value.

`<trie>` [Class]

{`data.trie`} A class for Trie. No slots are intended for public. Use the following procedures to operate on tries.

This class also implements the dictionary interface (see Section 9.9.1 [Generic functions for dictionaries], page 367).

`make-trie` *:optional tab-make tab-get tab-put! tab-fold tab-empty?* [Function]

{`data.trie`} Creates and returns an empty trie. The optional arguments are procedures to customize how the nodes of the internal tree are managed.

Each node can have a table to store its child nodes, indexed by an element of the key sequence (e.g. if the trie uses strings as keys, a node's table is indexed by characters).

`tab-make` A procedure with no arguments. When called, creates and returns an empty table for a node.

`tab-get` *tab elt*
: Returns a child node indexed by *elt*, or returns `#f` if the table doesn't have a child for *elt*.

`tab-put!` *tab elt child-node*
: If *child-node* isn't `#f`, stores a *child-node* with index *elt*. If *child-node* is `#f`, removes the entry with index *elt*. In both cases, this procedure should return the updated table.

`tab-fold` *tab proc seed*
: Calls *proc* for every index and node in *tab*, while passing a seed value, whose initial value is *seed*. That is, *proc* has a type of `(index, node, seed) -> seed`. Should return the last result of *proc*.

`tab-empty?` *tab*
: Returns `#t` if *tab* is empty, `#f` otherwise. You can omit or pass `#f` to this procedure; then we use `tab-fold` to check if *tab* is empty, which can be expensive.

The default assumes `eqv?`-hashtables, i.e. the following procedures are used.

```
tab-make: (lambda () (make-hash-table 'eqv?))

tab-get:  (lambda (tab k) (hash-table-get tab k #f))

tab-put!: (lambda (tab k v)
            (if v
              (hash-table-put! tab k v)
              (hash-table-delete! tab k))
```

```
               tab)

tab-fold: hash-table-fold

tab-empty?: (lambda (tab) (zero? (hash-table-num-entries tab)))
```

The following example creates a trie using assoc list to manage children, while comparing string keys with case-insensitive way:

```
(make-trie list
           (cut assoc-ref <> <> #f char-ci=?)
           (lambda (t k v)
             (if v
               (assoc-set! t k v char-ci=?)
               (alist-delete! k t char-ci=?)))
           (lambda (t f s) (fold f s t))
           null?)
```

It is important that `tab-put!` must return an updated table—by that, you can replace the table structure on the fly. For example, you may design a table which uses assoc list when the number of children are small, and then switches to a vector (indexed by character code) once the number of children grows over a certain threshold.

`trie` *params kv ...* [Function]

{`data.trie`} Construct a trie with the initial contents *kv ...*, where each *kv* is a pair of a key and a value. *Params* are a list of arguments which will be given to `make-trie` to create the trie. The following example creates a trie with two entries and the default table procedures.

```
(trie '() '("foo" . a) '("bar" . b))
```

`trie-with-keys` *params key ...* [Function]

{`data.trie`} A convenient version of `trie` when you only concern the keys. Each value is the same as its key. The following example creates a trie with two entries and the default table procedures.

```
(trie-with-keys '() "foo" "bar")
```

`trie?` *obj* [Function]

{`data.trie`} Returns `#t` if *obj* is a trie, or `#f` otherwise.

`trie-num-entries` *trie* [Function]

{`data.trie`} Returns the number of entries in *trie*.

`trie-exists?` *trie key* [Function]

{`data.trie`} Returns `#t` if *trie* contains an entry with *key*, or returns `#f` otherwise.

```
(let1 t (trie '() '("foo" . ok))
  (list (trie-exists? t "foo")
        (trie-exists? t "fo")
        (trie-exists? t "bar")))
  ⇒ '(#t #f #f)
```

`trie-partial-key?` *trie seq* [Function]

{`data.trie`} Returns `#t` if there's at least one key in *trie* that is not equal to *seq* but *seq* matches its prefix. Note that *seq* may or may not a key of *trie*; see the example below.

```
(define t (trie '() '("foo" . ok) '("fo" . ok)))
```

```
(trie-partial-key? t "f")    ⇒ #t
(trie-partial-key? t "fo")   ⇒ #t
(trie-partial-key? t "foo")  ⇒ #f
(trie-partial-key? t "bar")  ⇒ #f
```

`trie-get` *trie key :optional fallback* [Function]

{`data.trie`} Returns the value associated with *key* in *trie*, if such an entry exists. When there's no entry for *key*, if *fallback* is given, it is returned; otherwise, an error is signaled.

`trie-put!` *trie key value* [Function]

{`data.trie`} Puts *value* associated to *key* into *trie*.

`trie-update!` *trie key proc :optional fallback* [Function]

{`data.trie`} Works like the following code, except that the lookup of entry in *trie* is done only once.

```
(let ((val (trie-get trie key fallback)))
  (trie-put! trie key (proc val)))
```

`trie-delete!` *trie key* [Function]

{`data.trie`} Removes an entry associated with *key* from *trie*. If there's no such entry, this procedure does nothing.

`trie->list` *trie* [Function]

{`data.trie`} Makes each entry in *trie* to a pair `(key . value)` and returns a list of pairs of all entries. The order of entries are undefined.

`trie-keys` *trie* [Function]

`trie-values` *trie* [Function]

{`data.trie`} Returns a list of all keys and values in *trie*, respectively. The order of keys/values are undefined.

`trie->hash-table` *trie ht-type* [Function]

{`data.trie`} Creates a hash table with type *ht-type* (see Section 6.15 [Hashtables], page 177, about hash table types), and populates it with every key and value pair in *trie*.

`trie-longest-match` *trie seq :optional fallback* [Function]

{`data.trie`} Returns a pair of the key and its value, where the key is the longest prefix of *seq*. If no such key is found, *fallback* is returned if it is provided, or an error is thrown.

Do not confuse this with `trie-common-prefix-*` procedures below; In this procedure, the key is the prefix of the given argument. In `trie-common-prefix-*` procedures, the given argument is the prefix of the keys.

```
(let1 t (make-trie)
  (trie-put! t "a"  'a)
  (trie-put! t "ab" 'ab)

  (trie-longest-match t "abc")  ⇒ ("ab" . ab)
  (trie-longest-match t "acd")  ⇒ ("a"  . a)
  (trie-longest-match t "ab")   ⇒ ("ab" . ab)
  (trie-longest-match t "zy")   ⇒ error
  )
```

`trie-common-prefix` *trie prefix* [Function]

`trie-common-prefix-keys` *trie prefix* [Function] `trie-common-prefix-values` *trie prefix* [Function]

{`data.trie`} Gathers all entries whose keys begin with *prefix*; `trie-common-prefix` returns those entries in a list of pairs `(key . value)`; `trie-common-prefix-keys` returns a list of keys; and `trie-common-prefix-values` returns a list of values. The order of entries in a returned list is undefined. If *trie* contains no entry whose key has *prefix*, an empty list is returned.

Note that prefix matching doesn't consider the type of sequence; if *trie* has entries for `"foo"` and `(#\f #\o #\o)`, `(trie-common-prefix trie "foo")` will return both entries.

`trie-common-prefix-fold` *trie prefix proc seed* [Function]

{`data.trie`} For each entry whose key begins with *prefix*, calls *proc* with three arguments, the entry's key, its value, and the current seed value. *Seed* is used for the first seed value, and the value *proc* returns is used for the seed value of the next call of *proc*. The last returned value from *proc* is returned from `trie-common-prefix-fold`. The order of entries on which *proc* is called is undefined. If *trie* contains no entry whose key has *prefix*, *proc* is never called and *seed* is returned.

`trie-common-prefix-map` *trie prefix proc* [Function]
`trie-common-prefix-for-each` *trie prefix proc* [Function]

{`data.trie`} These are to *trie-common-prefix-fold* as `map` and `for-each` are to `fold`; `trie-common-prefix-map` calls *proc* with key and value for matching entries and gathers its result to a list; `trie-common-prefix-for-each` also applies *proc*, but discards its results.

`trie-fold` *trie proc seed* [Function]
`trie-map` *trie proc* [Function]
`trie-for-each` *trie proc* [Function]

{`data.trie`} These procedures are like their common-prefix versions, but traverse entire *trie* instead.

12.17 `dbi` - Database independent access layer

`dbi` [Module]

This module provides the unified interface to access various relational database systems (RDBMS). The operations specific to individual database systems are packaged in database driver (DBD) modules, which is usually loaded implicitly by DBI layer.

The module is strongly influenced by Perl's DBI/DBD architecture. If you have used Perl DBI, it would be easy to use this module.

It's better to look at the example. This is a simple outline of accessing a database by `dbi` module:

```
(use dbi)
(use gauche.collection) ; to make 'map' work on the query result

(guard (e ((<dbi-error> e)
           ;; handle error
           ))
  (let* ((conn   (dbi-connect "dbi:mysql:test;host=dbhost"))
         (query  (dbi-prepare conn
                   "SELECT id, name FROM users WHERE department = ?"))
         (result (dbi-execute query "R&D"))
         (getter (relation-accessor result)))
    (map (lambda (row)
```

```
                (list (getter row "id")
                      (getter row "name")))
              result)))
```

There's nothing specific to the underlying database system except the argument `"dbi:mysql:test;host=dbhost"` passed to `dbi-connect`, from which `dbi` module figures out that it is an access to `mysql` database, loads `dbd.mysql` module, and let it handle the mysql-specific stuff. If you want to use whatever database system, you can just pass `"dbi:`*whatever*`:`*parameter*`"` to `dbi-connect` instead, and everything stays the same as far as you have `dbd.whatever` installed in your system.

A query to the database can be created by `dbi-prepare`. You can issue the query by `dbi-execute`. This two-phase approach allows you to create a prepared query, which is a kind of parameterized SQL statement. In the above example the query takes one parameter, denoted as `'?'` in the SQL. The actual value is given in `dbi-execute`. When you issue similar queries a lot, creating a prepared query and execute it with different parameters may give you performance gain. Also the parameter is automatically quoted.

When the query is a `SELECT` statement, its result is returned as a collection that implements the relation protocol. See Section 9.5 [Collection framework], page 344, and Section 12.70 [Relation framework], page 807, for the details.

The outermost `guard` is to catch errors. The `dbi` related errors are supposed to inherit `<dbi-error>` condition. There are a few specific errors defined in `dbi` module. A specific `dbd` layer may define more specific errors.

In the next section we describe user-level API, that is, the procedures you need to concern when you're using `dbi`. The following section is for the driver API, which you need to use to write a specific `dbd` driver to make it work with `dbi` framework.

12.17.1 DBI user API

DBI Conditions

There are several predefined conditions `dbi` API may throw. See Section 6.20 [Exceptions], page 204, for the details of conditions.

`<dbi-error>` [Condition Type]

{`dbi`} The base class of `dbi`-related conditions. Inherits `<error>`.

`<dbi-nonexistent-driver-error>` [Condition Type]

{`dbi`} This condition is thrown by `dbi-connect` when it cannot find the specified driver. Inherits `<dbi-error>`.

`driver-name` [Instance Variable of `<dbi-nonexistent-driver-error>`]

Holds the requested driver name as a string.

`<dbi-unsupported-error>` [Condition Type]

{`dbi`} This condition is thrown when the called method isn't supported by the underlying driver. Inherits `<dbi-error>`.

`<dbi-parameter-error>` [Condition Type]

{`dbi`} This condition is thrown when the number of parameters given to the prepared query doesn't match the ones in the prepared statement.

Besides these errors, if a driver relies on `dbi` to parse the prepared SQL statement, `<sql-parse-error>` may be thrown if an invalid SQL statement is passed to `dbi-prepare`. (see Section 12.58 [SQL parsing and construction], page 790).

Connecting to the database

`dbi-connect` *dsn :key username password* [Function]

{`dbi`} Connect to a database using a data source specified by *dsn* (data source name). *Dsn* is a string with the following syntax:

```
dbi:driver:options
```

Driver part names a specific driver. You need to have the corresponding driver module, `dbd.driver`, installed in your system. For example, if *dsn* begins with `"dbi:mysql:"`, `dbi-connect` tries to load `dbd.mysql`.

Interpretation of the *options* part is up to the driver. Usually it is in the form of `key1=value1;key2=value2;...`, but some driver may interpret it differently. For example, `mysql` driver allows you to specify a database name at the beginning of *options*. You have to check out the document of each driver for the exact specification of *options*.

The keyword arguments gives extra information required for connection. The *username* and *password* are commonly supported arguments. The driver may recognize more keyword arguments.

If a connection to the database is successfully established, a connection object (an instance of a subclass of `<dbi-connection>`) is returned. Otherwise, an error is signaled.

`<dbi-connection>` [Class]

{`dbi`} The base class of a connection to a database system. Each driver defines a subclass of this to keep information about database-specific connections.

`dbi-open?` (*c* <*dbi-connection*>) [Method]

{`dbi`} Queries whether a connection to the database is still open (active).

`dbi-close` (*c* <*dbi-connection*>) [Method]

{`dbi`} Closes a connection to the database. This causes releasing resources related to this connection. Once closed, *c* cannot be used for any dbi operations (except passing to `dbi-open?`). Calling `dbi-close` on an already closed connection has no effect.

Although a driver usually closes a connection when `<dbi-connection>` object is garbage-collected, it is not a good idea to rely on that, since the timing of GC is unpredictable. The user program must make sure that it calls `dbi-close` at a proper moment.

`dbi-list-drivers` [Function]

{`dbi`} Returns a list of module names of known drivers.

`<dbi-driver>` [Class]

{`dbi`} The base class of a driver. You usually don't need to see this as far as you're using the high-level `dbi` API.

`dbi-make-driver` *driver-name* [Function]

{`dbi`} This is a low-level function called from `dbi-connect` method, and usually a user doesn't need to call it.

Loads a driver module specified by *driver-name*, and instantiate the driver class and returns it.

Preparing and issuing queries

`dbi-prepare` *conn sql :key pass-through . . .* [Method]

{`dbi`} From a string representation of SQL statement *sql*, creates and returns a query object (an instance of `<dbi-query>` or its subclass) for the database connection `conn`

Sql may contain parameter slots, denoted by `?`.

```
(dbi-prepare conn "insert into tab (col1, col2) values (?, ?)")

(dbi-prepare conn "select * from tab where col1 = ?")
```

They will be filled when you actually issue the query by `dbi-execute`. There are some advantages of using parameter slots: (1) The necessary quoting is done automatically. You don't need to concern about security holes caused by improper quoting, for example. (2) Some drivers support a feature to send the template SQL statement to the server at the preparation stage, and send only the parameter values at the execution stage. It would be more efficient if you issue similar queries lots of time.

If the backend doesn't support prepared statements (SQL templates having `?` parameters), the driver may use `text.sql` module to parse *sql*. It may raise `<sql-parse-error>` condition if the given SQL is not well formed.

You may pass a true value to the keyword argument *pass-through* to suppress interpretation of SQL and pass *sql* as-is to the back end database system. It is useful if the back-end supports extension of SQL which `text.sql` doesn't understand.

If the driver lets prepared statement handled in back-end, without using `text.sql`, the `pass-through` argument may be ignored. The driver may also take other keyword arguments. Check out the documentation of individual drivers.

Note: Case folding of SQL statement is implementation dependent. Some DBMS may treat table names and column names in case insensitive way, while others do in case sensitive way. To write a portable SQL statement, make them quoted identifiers, that is, always surround names by double quotes.

`<dbi-query>` [Class]

{`dbi`} Holds information about prepared query, created by `dbi-prepare`. The following slots are defined.

`connection` [Instance Variable of `<dbi-query>`]

Contains the `<dbi-connection>` object.

`prepared` [Instance Variable of `<dbi-query>`]

If the driver prepares query by itself, this slot may contain a prepared statement. It is up to each driver how to use this slot, so the client shouldn't rely on its value.

`dbi-open?` (*q* <*dbi-query*>) [Method]

{`dbi`} Returns `#t` iff the query can still be passed to `dbi-execute`.

`dbi-close` (*q* <*dbi-query*>) [Method]

{`dbi`} Destroy the query and free resources associated to the query. After this operation, `dbi-open?` returns `#f` for *q*, and the query can't be used in any other way. Although the resource may be freed when *q* is garbage-collected, it is strongly recommended that the application closes queries explicitly.

`dbi-execute` (*q* <*dbi-query*>) *parameter . . .* [Method]

{`dbi`} Executes a query created by `dbi-prepare`. You should pass the same number of *parameter*s as the query expects.

If the issued query is `select` statement, `dbi-execute` returns an object represents a *relation*. A relation encapsulates the values in rows and columns, as well as meta information like column names. See "Retrieving query results" below for how to access the result.

If the query is other types, such as `create`, `insert` or `delete`, the return value of the query closure is unspecified.

`dbi-do` *conn sql :optional options parameter-value . . .* [Method]

{`dbi`} This is a convenience procedure when you create a query and immediately execute it. It is equivalent to the following expression, although the driver may overload this method to avoid creating intermediate query object to avoid the overhead.

```
(dbi-execute (apply dbi-prepare conn sql options)
             parameter-value ...)
```

`dbi-escape-sql` *conn str* [Method]

{`dbi`} Returns a string where special characters in *str* are escaped.

The official SQL standard only specify a single quote (`'`) as such character. However, it doesn't specify non-printable characters, and the database system may use other escaping characters. So it is necessary to use this method rather than doing escaping by your own.

```
;; assumes c is a valid DBI connection
(dbi-escape-sql c "don't know")
  ⇒ "don''t know"
```

Retrieving query results

If the query is a `select` statement, it returns an object of both `<collection>` and `<relation>`. It is a collection of rows (that is, it implements `<collection>` API), so you can use `map`, `for-each` or other generic functions to access rows. You can also use the relation API to retrieve column names and accessors from it. See Section 12.70 [Relation framework], page 807, for the relation API, and Section 9.5 [Collection framework], page 344, for the collection API.

The actual class of the object returned from a query depends on the driver, but you may use the following method on it.

`dbi-open?` *result* [Method]

{`dbi`} Check whether the result of a query is still active. The result may become inactive when it is explicitly closed by `dbi-close` and/or the connection to the database is closed.

`dbi-close` *result* [Method]

{`dbi`} Close the result of the query. This may cause releasing resources related to the result. You can no longer use *result* once it is closed, except passing it to `dbi-open?`.

Although a driver usually releases resources when the result is garbage-collected, the application shouldn't rely on that and is recommended call `dbi-close` explicitly when it is done with the result.

12.17.2 Writing drivers for DBI

Writing a driver for a specific database system means implementing a module `dbd.`*`foo`*, where *foo* is the name of the driver.

The module have to implement several classes and methods, as explained below.

DBI classes to implement

You have to define the following classes.

- Subclass `<dbi-driver>`. The class name *must* be `<`*`foo`*`-driver>`, where *foo* is the name of the driver. Usually this class produces a singleton instance, and is only used to dispatch `dbi-make-connection` method below.
- Subclass `<dbi-connection>`. An instance of this class is created by `dbi-make-connection`. It needs to keep the information about the actual connections.
- Subclass `<relation>` and `<collection>` to represent query results suitable for the driver. (In most cases, the order of the result of SELECT statement is significant, since it may be

sorted by ORDER BY clause. Thus it is more appropriate to inherit `<sequence>`, rather than `<collection>`).

- Optionally, subclass `<dbi-query>` to keep driver-specific information of prepared queries.

DBI methods to implement

The driver need to implement the following methods.

`dbi-make-connection` (*d <foo-driver>*) (*options <string>*) (*options-alist <list>*) *:key username password . . .* [Method]

{`dbi`} This method is called from `dbi-connect`, and responsible to connect to the database and to create a connection object. It must return a connection object, or raise an `<dbi-error>` if it cannot establish a connection.

Options is the option part of the data source name (DSN) given to `dbi-connect`. *options-alist* is an assoc list of the result of parsing *options*. Both are provided so that the driver may interpret *options* string in nontrivial way.

For example, given `"dbi:foo:myaddressbook;host=dbhost;port=8998"` as DSN, foo's `dbi-make-connection` will receive `"myaddressbook;host=dbhost;port=8998"` as *options*, and `(("myaddressbook" . #t) ("host" . "dbhost") ("port" . "8998"))` as *options-alist*.

After *options-alist*, whatever keyword arguments given to `dbi-connect` are passed. DBI protocol currently specifies only *username* and *password*. The driver may define other keyword arguments. It is recommended to name the driver-specific keyword arguments prefixed by the driver name, e.g. for `dbd.foo`, it may take a `:foo-whatever` keyword argument.

It is up to the driver writer to define what options are available and the syntax of the options. The basic idea is that the DSN identifies the source of the data; it's role is like URL in WWW. So, it may include the hostname and port number of the database, and/or the name of the database, etc. However, it shouldn't include information related to authentication, such as username and password. That's why those are passed via keyword arguments.

`dbi-prepare` (*c <foo-connection>*) (*sql <string>*) *:key pass-through . . .* [Method]

{`dbi`} This method should create and return a prepared query object, which is an instance of `<dbi-query>` or its subclass. The query specified by *sql* is issued to the database system when the prepared query object is passed to `dbi-execute`.

The method must set *c* to the `connection` slot of the returned query object.

Sql is an SQL statement. It may contain placeholders represented by `'?'`. The query closure should take the same number of arguments as of the placeholders. It is up to the driver whether it parses *sql* internally and construct a complete SQL statement when the query closure is called, or it passes *sql* to the back-end server to prepare the statement and let the query closure just send parameters.

If the driver parses SQL statement internally, it should recognize a keyword argument `pass-through`. If a true value is given, the driver must treat `sql` opaque and pass it as is when the query closure is called.

The driver may define other keyword arguments. It is recommended to name the driver-specific keyword arguments prefixed by the driver name, e.g. for `dbd.foo`, it may take a `:foo-whatever` keyword argument.

`dbi-execute-using-connection` (*c <foo-connection>*) (*q <dbi-query>*) (*params <list>*) [Method]

{`dbi`} This method is called from `dbi-execute`. It must issue the query kept in *q*. If the query is parameterized, the actual parameters given to *dbi-execute* are passed to *params* argument.

If *q* is a `select`-type query, this method must return an appropriate relation object.

`dbi-escape-sql` (*c* <*foo-connection*>) *str* [Method]

{`dbi`} If the default escape method isn't enough, the driver may overload this method to implement a specific escaping. For example, MySQL treats backslash characters specially as well as single quotes, so it has its `dbi-escape-sql` method.

`dbi-open?` (*c* <*foo-connection*>) [Method]
`dbi-open?` (*q* <*foo-query*>) [Method]
`dbi-open?` (*r* <*foo-result*>) [Method]
`dbi-close` (*c* <*foo-connection*>) [Method]
`dbi-close` (*q* <*foo-query*>) [Method]
`dbi-close` (*r* <*foo-result*>) [Method]

{`dbi`} Queries open/close status of a connection and a result, and closes a connection and a result. The close methods should cause releasing resources used by connection/result. The driver has to allow `dbi-close` to be called on a connection or a result which has already been closed.

`dbi-do` (*c* <*foo-connection*>) (*sql* <*string*>) *:optional options parameter-value . . .* [Method]

{`dbi`} The default method uses `dbi-prepare` and `dbi-execute` to implement the function. It just works, but the driver may overload this method in order to skip creating intermediate query object for efficiency.

DBI utility functions

The following functions are low-level utilities which you may use to implement the above methods.

`dbi-parse-dsn` *data-source-name* [Function]

{`dbi`} Parse the data source name (DSN) string given to `dbi-connect`, and returns tree values: (1) The driver name in a string. (2) 'options' part of DSN as a string. (3) parsed options in an assoc list. This may raise `<dbi-error>` if the given string doesn't conform DSN syntax.

You don't need to use this to write a typical driver, for the parsing is done before `dbi-make-connection` is called. This method may be useful if you're writing a kind of meta-driver, such as a proxy.

`dbi-prepare-sql` *connection sql* [Function]

{`dbi`} Parses an SQL statement *sql* which may contain placeholders, and returns a closure, which generates a complete SQL statement when called with actual values for the parameters. If the back-end doesn't support prepared statements, you may use this function to prepare queries in the driver.

Connection is a DBI connection to the database. It is required to escape values within SQL properly (see `dbi-escape-sql` above).

```
;; assume c contains a valid dbi connection
((dbi-prepare-sql c "select * from table where id=?") "foo'bar")
 => "select * from table where id='foo''bar'"
```

12.18 `dbm` - Generic DBM interface

`dbm` [Module]

DBM-like libraries provides an easy way to store values to a file, indexed by keys. You can think it as a persistent associative memory.

This modules defines `<dbm>` abstract class, which has a common interface to use various DBM-type database packages. As far as you operate on the already opened database, importing `dbm` module is enough.

To create or open a database, you need a concrete implementation of the database. With the default build-time configuration, the following implementations are included in Gauche. Bindings to various other dbm-like libraries are available as extension packages. Each module defines its own low-level accessing functions as well as the common interface. Note that your system may not have one or more of those DBM libraries; Gauche defines only what the system provides.

`dbm.fsdbm`
: file-system dbm (see Section 12.19 [File-system dbm], page 687).

`dbm.gdbm`
: GDBM library (see Section 12.20 [GDBM interface], page 688).

`dbm.ndbm`
: NDBM library (see Section 12.21 [NDBM interface], page 690).

`dbm.odbm`
: DBM library (see Section 12.22 [Original DBM interface], page 690).

The following code shows a typical usage of the database.

```
(use dbm)           ; dbm abstract interface
(use dbm.gdbm)      ; dbm concrete interface

; open the database
(define *db* (dbm-open <gdbm> :path "mydb" :rw-mode :write))

; put the value to the database
(dbm-put! *db* "key1" "value1")

; get the value from the database
(define val (dbm-get *db* "key1"))

; iterate over the database
(dbm-for-each *db* (lambda (key val) (foo key val)))

; close the database
(dbm-close *db*)
```

The `<dbm>` abstract class implements collection and dictionary framework. (See Section 9.5 [Collection framework], page 344, and Section 9.9 [Dictionary framework], page 366, respectively).

12.18.1 Opening and closing a dbm database

`<dbm>` [Class]

{`dbm`} An abstract class for dbm-style database. Inherits `<dictionary>` (see Section 9.9 [Dictionary framework], page 366). Defines the common database operations. This class has the following instance slots. They must be set before the database is actually opened by `dbm-open`.

The concrete class may add more slots for finer control on the database, such as locking.

`path` [Instance Variable of `<dbm>`]

Pathname of the dbm database. Some dbm implementation may append suffixes to this.

`rw-mode` [Instance Variable of `<dbm>`]

Specifies read/write mode. Can be either one of the following keywords:

`:read` The database will be opened in read-only mode. The database file must exist when `dbm-open` is called.

`:write` The database will be opened in Read-write mode. If the database file does not exist, `dbm-open` creates one.

`:create` The database will be created and opened in Read-write mode. If the database file exists, `dbm-open` truncates it.

`file-mode` [Instance Variable of `<dbm>`]

Specifies the file permissions (as `sys-chmod`) to create the database. The default value is `#o664`.

`key-convert` [Instance Variable of `<dbm>`]

`value-convert` [Instance Variable of `<dbm>`]

By default, you can use only strings for both key and values. With this option, however, you can specify how to convert other Scheme values to/from string to be stored in the database. The possible values are the followings:

`#f` The default value. Keys (values) are not converted. They must be a string.

`#t` Keys (values) are converted to its string representation, using `write`, to store in the database, and converted back to Scheme values, using `read`, to retrieve from the database. The data must have an external representation that can be read back. (But it is not checked when the data is written; you'll get an error when you read the data). The key comparison is done in the string level, so the external representation of the same key must match.

a list of two procedures

Both procedure must take a single argument. The first procedure must receive a Scheme object and returns a string. It is used to convert the keys (values) to store in the database. The second procedure must receive a string and returns a Scheme object. It is used to convert the stored data in the database to a Scheme object. The key comparison is done in the string level, so the external representation of the same key must match.

`<dbm-meta>` [Metaclass]

{`dbm`} A metaclass of `<dbm>` and its subclasses.

`dbm-open` (*dbm* `<dbm>`) [Method]

{`dbm`} Opens a dbm database. *dbm* must be an instance of one of the concrete classes that derived from the `<dbm>` class, and its slots must be set appropriately. On success, it returns the *dbm* itself. On failure, it signals an error.

`dbm-open` (*dbm-class* `<dbm-meta>`) *options* . . . [Method]

{`dbm`} A convenient method that creates dbm instance and opens it. It is defined as follows.

```
(define-method dbm-open ((class <class>) . initargs)
  (dbm-open (apply make class initargs)))
```

Database file is closed when it is garbage collected. However, to ensure the modification is properly synchronized, you should close the database explicitly.

`dbm-close` (*dbm* `<dbm>`) [Method]

{`dbm`} Closes a database *dbm*. Once the database is closed, any operation to access the database content raises an error.

dbm-closed? (*dbm* <dbm>) [Method]

{dbm} Returns true if a database *dbm* is already closed, false otherwise.

dbm-type->class *dbmtype* [Function]

{dbm} Sometimes you don't know which type of dbm implementation you need to use in your application beforehand, but rather you need to determine the type according to the information given at run-time. This procedure fulfills the need.

The *dbmtype* argument is a symbol that names the type of dbm implementation; for example, `gdbm` for `dbm.gdbm`, and `fsdbm` for `dbm.fsdbm`. We assume that the dbm implementation of type *foo* is provided as a module `dbm.`*foo*, and its class is named as `<`*foo*`>`.

This procedure first checks if the required module has been loaded, and if not, it tries to load it. If the module loads successfully, it returns the class object of the named dbm implementation. If it can't load the module, or can't find the dbm class, this procedure returns #f.

```
(use dbm)

(dbm-type->class 'gdbm)
  ⇒ #<class <gdbm>>

(dbm-type->class 'nosuchdbm)
  ⇒ #f
```

12.18.2 Accessing a dbm database

Once a database is opened, you can use the following methods to access individual key/value pairs.

dbm-put! (*dbm* <dbm>) *key value* [Method]

{dbm} Put a *value* with *key*.

dbm-get (*dbm* <dbm>) *key :optional default* [Method]

{dbm} Get a value associated with *key*. If no value exists for *key* and *default* is specified, it is returned. If no value exists for *key* and *default* is not specified, an error is signaled.

dbm-exists? (*dbm* <dbm>) *key* [Method]

{dbm} Return true if a value exists for *key*, false otherwise.

dbm-delete! (*dbm* <dbm>) *key* [Method]

{dbm} Delete a value associated with *key*.

12.18.3 Iterating on a dbm database

To walk over the entire database, following methods are provided.

dbm-fold (*dbm* <dbm>) *procedure knil* [Method]

{dbm} The basic iterator. For each key/value pair, *procedure* is called as `(`*procedure key value r*`)`, where *r* is *knil* for the fist call of *procedure*, and the return value of the previous call for subsequent calls. Returns the result of the last call of *procedure*. If no data is in the database, *knil* is returned.

The following method returns the sum of all the integer values.

```
(dbm-fold dbm (lambda (k v r) (if (integer? v) (+ v r) r)) 0)
```

dbm-for-each (*dbm* <dbm>) *procedure* [Method]

{dbm} For each key/value pair in the database *dbm*, *procedure* is called. Two arguments are passed to *procedure*—a key and a value. The result of *procedure* is discarded.

`dbm-map` (*dbm* <dbm>) *procedure* [Method]

{`dbm`} For each key/value pair in the database *dbm*, *procedure* is called. Two arguments are passed to *procedure*—a key and a value. The result of *procedure* is accumulated to a list which is returned as a result of `dbm-map`.

12.18.4 Managing dbm database instance

Each dbm implementation has its own way to store the database. Legacy dbm uses two files, whose names are generated by adding `.dir` and `.pag` to the value of *path* slot. `Fsdbm` creates a directory under *path*. If dbm database is backed up by some database server, *path* may be used only as a key to the database in the server. The following methods hide such variations and provides a convenient way to manage a database itself. You have to pass a class that implements a concrete dbm database to their first argument.

`dbm-db-exists?` *class name* [Generic Function]

{`dbm`} Returns `#t` if a database of class *class* specified by *name* exists.

```
;; Returns #t if testdb.dir and testdb.pag exist
(dbm-db-exists? <odbm> "testdb")
```

`dbm-db-remove` *class name* [Generic Function]

{`dbm`} Removes an entire database of class *class* specified by *name*.

`dbm-db-copy` *class from to* [Generic Function]

{`dbm`} Copy a database of class *class* specified by *from* to *to*. The integrity of *from* is guaranteed if the *class*'s dbm implementation supports locking (i.e. you won't get a corrupted database even if some other process is trying to write to *from* during copy). If the destination database *to* exists, its content is destroyed. If this function is interrupted, whether *to* is left in incomplete state or not depends on the dbm implementation. The implementation usually tries its best to provide transactional behavior, that is, to recover original *to* when the copy fails. However, for the robust operations the caller have to check the state of *to* if `dbm-db-copy` fails.

```
(dbm-db-copy <gdbm> "testdb.dbm" "backup.dbm")
```

`dbm-db-move` *class from to* [Generic Function]

{`dbm`} Moves or renames a database of class *class* specified by *from* to *to*. Like `dbm-db-copy`, the database integrity is guaranteed as far as *class*'s dbm implementation supports locking. If the destination database *to* exists, its content is destroyed.

12.18.5 Dumping and restoring dbm database

Most dbm implementations use some kind of binary format, and some of them are architecture dependent. That makes it difficult to pass around dbm databases between different machines. A safe way is to write out the content of a dbm database into some portable format on the source machine, and rebuild another dbm database from it on the destination machine.

The operation is so common that Gauche provides convenience scripts that does the job. They are installed into the standard Gauche library directory, so it can be invoked by `gosh <scriptname>`.

To write out the content of a dbm database named by *dbm-name*, you can use `dbm/dump` script:

```
$ gosh dbm/dump [-o outfile][-t type] dbm-name
```

The *outfile* argument names the output file. If omitted, the output is written out to stdout. The *type* argument specifies the implementation type of the dbm database; e.g. `gdbm` or `fsdbm`. The program calls `dbm-type->class` (see Section 12.18.1 [Opening and closing a dbm database], page 683) on the *type* argument to load the necessary dbm implementation.

The dumped format is simply a series of S-expressions, each of which is a dotted pair of string key and string value. Character encodings are assumed to be the same as `gosh`'s native character encoding.

The dumped output may contain S-expressions other than dotted pair of strings to include meta information. For now, programs that deals with dumped output should just ignore S-expressions other than dotted pairs.

To read back the dumped dbm format, you can use `dbm/restore` script:

```
$ gosh dbm/restore [-i infile][-t type] dbm-name
```

The *infile* argument names the dumped file to be read. If omitted, it reads from stdin. The *type* argument specifies the dbm type, as in `dbm/dump` script. The *dbm-name* argument names the dbm database; if the database already exists, its content is cleared, so be careful.

12.18.6 Writing a dbm implementation

When you write an extension module that behaves like a persistent hashtable, it is a good idea to adapt it to the dbm interface, so that the application can use the module in a generic way.

The minimum procedures to conform the dbm interface are as follow:

- Define a metaclass `<foo-meta>`. It doesn't need to inherit anything except `<class>`.
- Define a dbm class `<foo>` that inherits `<dbm>` and whose metaclass is `<foo-meta>`.
- Define methods for `dbm-open`, `dbm-close`, `dbm-put!`, `dbm-get`, `dbm-exists`, `dbm-delete!`, `dbm-fold`, `dbm-closed?`, specialized for `<foo>`. (The case of `dbm-open` for `<foo-meta>` is handled automatically, so you don't need to define it unless you want something special). Also note that the specialized `dbm-open` must call `next-method` in it to set up dbm base class internals.
- Define methods for `dbm-db-exists?` and `dbm-db-remove` on `<foo-meta>`.

Besides above, you may define the following methods.

- Methods for `dbm-for-each` and `dbm-map`. If you don't define them, a generic implementation by `dbm-fold` is used. There may be an implementation specific way which is more efficient.
- Methods for `dbm-db-copy` and `dbm-db-move`. If you don't define them, a fallback method opens the specified databases and copies elements one by one, and removes the original if the method is `dbm-db-move`. Note that the fallback method is not only inefficient, but also it may not copy any implementation-specific meta information. It is highly recommended for the dbm implementation to provide these methods as well.

It is generally recommended to name the implementation module as `dbm.foo`, and the class of the implementation as `<foo>`. With this convention it is easier to write an application that dynamically loads and uses dbm implementation specified at runtime.

12.19 `dbm.fsdbm` - File-system dbm

`dbm.fsdbm` [Module]

Implements fsdbm. Extends `dbm`.

`<fsdbm>` [Class]

{`dbm.fsdbm`} `Fsdbm` is a dbm implementation that directly uses the filesystem. Basically, it uses file names for keys, and file content for values. Unlike other dbm implementations, this doesn't depend on external libraries—it is pure Scheme implementation—so it is always available, while other dbm implementations may not. Obviously, it is not suitable for the database that has lots of entries, or has entries deleted and added very frequently. The advantage is when the number of entries are relatively small, and the values are relatively large while keys are small. The database name given to `<fsdbm>` instance is used as a directory name that stores the data. The data files are stored in subdirectories under *path* of `fsdbm` instance, hashed by the key. Non-alphanumeric characters in the key is encoded like `_3a` for ':', for example. If a key is too long to be a file name, it is chopped to chunks, and each chunk but the last one is used as a directory name. Note that a long key name may still cause a problem, for example, some of old 'tar' command can't deal with pathnames (not each pathname components, but the entire pathname) longer than 256 characters.

Fsdbm implements all of the dbm protocol (see Section 12.18 [Generic DBM interface], page 682). It doesn't have any fsdbm-specific procedures.

12.20 `dbm.gdbm` - GDBM interface

`dbm.gdbm` [Module]
Provides interface to the gdbm library. Extends `dbm`.

`<gdbm>` [Class]
{`dbm.gdbm`} Inherits `<dbm>`. Provides an implementation for GDBM library. This module is only installed when your system already has GDBM (1.8.0 is preferred, but works with older 1.7.x with some limitations).

`sync` [Instance Variable of `<gdbm>`]

`nolock` [Instance Variable of `<gdbm>`]

`bsize` [Instance Variable of `<gdbm>`]

Besides the unified DBM interface (see Section 12.18 [Generic DBM interface], page 682), this module provides the following low-level functions that provides direct access to the gdbm API. See gdbm manual for details of these APIs.

`gdbm-open` *path :optional size rwmode fmode error-callback* [Function]
{`dbm.gdbm`}

`GDBM_READER` [Variable]
{`dbm.gdbm`}

`GDBM_WRITER` [Variable]
{`dbm.gdbm`}

`GDBM_WRCREAT` [Variable]
{`dbm.gdbm`}

`GDBM_NEWDB` [Variable]
{`dbm.gdbm`}

`GDBM_FAST` [Variable]
{`dbm.gdbm`}

`GDBM_SYNC` [Variable]
{`dbm.gdbm`}

`GDBM_NOLOCK` [Variable]
{`dbm.gdbm`}

`gdbm-close` *gdbm-object* [Function]
{`dbm.gdbm`}

`gdbm-closed?` *gdbm-object* [Function]
{dbm.gdbm}

`gdbm-store` *key value :optional flag* [Function]
{dbm.gdbm}

`GDBM_INSERT` [Variable]
{dbm.gdbm}

`GDBM_REPLACE` [Variable]
{dbm.gdbm}

`gdbm-fetch` *gdbm-object key* [Function]
{dbm.gdbm}

`gdbm-delete` *gdbm-object key* [Function]
{dbm.gdbm}

`gdbm-firstkey` *gdbm-object* [Function]
{dbm.gdbm}

`gdbm-nextkey` *gdbm-object key* [Function]
{dbm.gdbm}

`gdbm-reorganize` *gdbm-object* [Function]
{dbm.gdbm}

`gdbm-sync` *gdbm-object* [Function]
{dbm.gdbm}

`gdbm-exists?` *gdbm-object key* [Function]
{dbm.gdbm}

`gdbm-strerror` *errno* [Function]
{dbm.gdbm}

`gdbm-setopt` *gdbm-object option value* [Function]
{dbm.gdbm}

`GDBM_CACHESIZE` [Variable]
{dbm.gdbm}

`GDBM_FASTMODE` [Variable]
{dbm.gdbm}

`GDBM_SYNCMODE` [Variable]
{dbm.gdbm}

`GDBM_CENTFREE` [Variable]
{dbm.gdbm}

`GDBM_COALESCEBLKS` [Variable]
{dbm.gdbm}

`gdbm-version` [Function]
{dbm.gdbm}

`gdbm-errno` [Function]
{dbm.gdbm}

12.21 dbm.ndbm - NDBM interface

`dbm.ndbm` [Module]
Provides interface to the 'new' dbm library, a.k.a. ndbm. Extends `dbm`.

`<ndbm>` [Class]
{`dbm.ndbm`} Inherits `<dbm>`. Provides an implementation for NDBM library. This module is only installed when your system already has NDBM.

Besides the unified DBM interface (see Section 12.18 [Generic DBM interface], page 682), this module provides the following low-level functions that provides direct access to the ndbm API. See ndbm manual for details of these APIs.

`ndbm-open` *path flags mode* [Function]
{`dbm.ndbm`}

`ndbm-close` *ndbm-object* [Function]
{`dbm.ndbm`}

`ndbm-closed?` *ndbm-object* [Function]
{`dbm.ndbm`}

`ndbm-store` *ndbm-object key content :optional flag* [Function]
{`dbm.ndbm`}

`ndbm-fetch` *ndbm-object key* [Function]
{`dbm.ndbm`}

`ndbm-delete` *ndbm-object key* [Function]
{`dbm.ndbm`}

`ndbm-firstkey` *ndbm-object* [Function]
{`dbm.ndbm`}

`ndbm-nextkey` *ndbm-object* [Function]
{`dbm.ndbm`}

`ndbm-error` *ndbm-object* [Function]
{`dbm.ndbm`}

`ndbm-clear-error` *ndbm-object* [Function]
{`dbm.ndbm`}

12.22 dbm.odbm - Original DBM interface

`dbm.odbm` [Module]
Provides interface to the legacy dbm library. Extends `dbm`.

`<odbm>` [Class]
{`dbm.odbm`} Inherits `<dbm>`. Provides an implementation for legacy DBM library. This module is only installed when your system already has DBM.

The biggest limitation of the legacy DBM is that you can only open one database at a time. You can create a multiple `<odbm>` instances, but you can open at most one of it at a time, or you'll get an error.

Besides the unified DBM interface (see Section 12.18 [Generic DBM interface], page 682), this module provides the following low-level functions that provides direct access to the dbm API. See dbm manual for details of these APIs.

`odbm-init` *path* [Function]
{`dbm.odbm`}

`odbm-close` [Function]
{`dbm.odbm`}

`odbm-store` *key value* [Function]
{`dbm.odbm`}

`odbm-fetch` *key* [Function]
{`dbm.odbm`}

`odbm-delete` *key* [Function]
{`dbm.odbm`}

`odbm-firstkey` [Function]
{`dbm.odbm`}

`odbm-nextkey` *key* [Function]
{`dbm.odbm`}

12.23 `file.filter` - Filtering file content

`file.filter` [Module]
This module provides utilities for a common pattern in filter-type commands, that is, to take an input, to process the content, and to write the result. The common occurring pattern is:

- Input may be a specified file, or an input port (the current input port by default).
- Output may be a specified file, or an output port (the current output port by default).
- Output may be a temporary file, which will be renamed upon completion of the processing.
- Output file may be removed when an error occurs in the processing.

`file-filter` *proc :key input output temporary-file keep-output? rename-hook* [Function]
{`file.filter`} Calls *proc* with two arguments, an input port and an output port. Returns the result(s) of *proc*. The input port and output port are chosen depending on the keyword arguments.

`input` The argument must be either an input port or a string that specifies a file name. If it's an input port, it is passed to *proc* as is. If it's a string, the named file is opened for input and the resulting port is passed to *proc*, and the port is closed when *proc* returns. If this argument is omitted, the current input port is passed.

`output` The argument must be either an output port or a string that specifies a file name. If it's an output port, it is passed to *proc* as is. If it's a string, the named file is opened for output (unless *temporary-file* is given, in that case a temporary file is opened instead), and the resulting port is passed to *proc*. This port is closed when *proc* returns. If this argument is omitted, the current output port is passed.

`temporary-file`
: The value must be a boolean or a string. If a non-false value is given, and output is a file, then a fresh temporary file is created and opened for output and passed to *proc*. When *proc* returns normally, the file is renamed to the name given to *output* keyword argument.

  If `#t` is given, a temporary file name is generated based on the name of the output file. If a string file name is given to this argument, the name is used for `sys-mkstemp`.

  If the given file name begins with characters except `"/"`, `"./"` or `"../"`, the directory of the file name given to *output* argument is attached before it.

  The default value is `#f` (do not use a temporary file).

`keep-output?`
: If a true value is given, the output is not deleted even when *proc* signals an error. By default, the output (or the temporary file when *temporary-file* is given) will be deleted on error.

`leave-unchanged`
: When a temporary file is used, and a true value is given to this argument, the existing output file is left intact when the generated output in the temporary file exactly matches the original content of the output file. It is useful if touching output file may trigger some actions (e.g. by `make`) and you want to avoid invoking unnecessary actions. The default value is `#f` (always replace the output).

`file-filter-fold` *proc seed :key reader input output temporary-file keep-output? rename-hook* [Function]

{`file.filter`} A convenience wrapper of `file-filter`. Call *proc* for each item read from input by *reader* (`read-line` by default). The argument *proc* receives is the item, the seed value and the output port; *proc* can emit the output, as well as returning some value that is passed along as the seed value. Other keyword arguments are passed to `file-filter`.

For example, the following code reads each line from `file.txt` and displays lines matching `#/regexp/` with line numbers.

```
(file-filer-fold
  (^[line nc out]
    (when (#/regexp/ line) (format out "~3d: ~a\n" nc line))
    (+ nc 1))
  1 :input "file.txt")
```

`file-filter-map` *proc :key reader input output temporary-file keep-output? rename-hook* [Function]

`file-filter-for-each` *proc :key reader input output temporary-file keep-output? rename-hook* [Function]

{`file.filter`} Utilities similar to `file-filter-fold`, like `map` and `for-each` to `fold`.

The procedure *proc* is called with two arguments, an item read from the input and an output port. The results of *proc* are collected as a list and returned by `file-filter-map`, and discarded by `file-filter-for-each`.

The meaning of keyword arguments are the same as `file-filter-fold`.

12.24 `file.util` - Filesystem utilities

`file.util` [Module]

Provides convenient utility functions handling files and directories. Those functions are built on top of the primitive system procedures described in Section 6.25.4 [Filesystems], page 250.

Many procedures in this module takes a keyword argument *follow-link?*, which specifies the behavior when the procedure sees a symbolic link. If true value is given to *follow-link?* (which is the default), the procedure operates on the file referenced by the link; if false is given, it operates on the link itself.

Note on the naming convention: Some Scheme implementations "create" new directories and files, while the others "make" them. Some implementations "delete" them, while the others "remove" them. It seems that both conventions are equally popular. So Gauche provides *both*.

12.24.1 Directory utilities

`current-directory` *:optional new-directory* [Function]

{`file.util`} When called with no argument, this returns the pathname of the current working directory. When called with a string argument *new-directory*, this sets the current working directory of the process to it. If the process can't change directory to *new-directory*, an error is signaled.

This function is in ChezScheme, MzScheme and some other Scheme implementations.

`home-directory` *:optional user* [Function]

{`file.util`} Returns the home directory of the given *user*, which may be a string user name or an integer user id. If *user* is omitted, the current user is assumed. If the given user cannot be found, or the home directory of the user cannot be determined, `#f` is returned.

On Windows native platforms, this function is only supported to query the current user's directory.

`directory-list` *path :key children? add-path? filter filter-add-path?* [Function]

{`file.util`} Returns a list of entries in the directory *path*. The result is sorted by dictionary order.

By default, only the basename (the last component) of the entries returned. If *add-path?* is given and true, *path* is appended to each entry. If *children?* is given and true, `"."` and `".."` are excluded from the result.

If *filter* is given, it must be a predicate that takes one argument. It is called on every element of the directory entry, and only the entries on which *filter* returns true are included in the result. The argument passed to *filter* is a basename of the directory entry by default, but when *filter-add-path?* is true, *path* is appended to the entry.

If *path* is not a directory, an error is signaled.

```
(directory-list "test")
 ⇒ ("." ".." "test.scm" "test.scm~")

(directory-list "test" :add-path? #t)
 ⇒ ("test/." "test/.." "test/test.scm" "test/test.scm~")

(directory-list "test" :children? #t)
 ⇒ ("test.scm" "test.scm~")

(directory-list "test" :children? #t :add-path? #t
   :filter (lambda (e) (not (string-suffix? "~" e))))
 ⇒ ("test/test.scm")
```

`directory-list2` *path :key children? add-path? filter follow-link?* [Function]

{`file.util`} Like `directory-list`, but returns two values; the first one is a list of subdirectories, and the second one is a list of the rest. The keyword arguments *children?*, `add-path?` and *filter* are the same as `directory-list`.

Giving false value to *follow-link?* makes `directory-list2` not follow the symbolic links; if the *path* contains a symlink to a directory, it will be included in the first list if *follow-link?* is omitted or true, while it will be in the second list if *follow-link?* is false.

`directory-fold` *path proc seed :key lister follow-link?* [Function]

{`file.util`} A fundamental directory traverser. Conceptually it works as follows, in recursive way.

- If *path* is not a directory, calls `(proc path seed)` and returns the result.
- If *path* is a directory, calls `(lister path seed)`. The procedure *lister* is expected to return two values: a list of pathnames, and the next seed value. Then `directory-fold` is called on each returned pathname, passing the returned seed value to the *seed* argument of the next call of `directory-fold`. Returns the result of the last seed value.

The default procedure of *lister* is just a call to `directory-list`, as follows.

```
(lambda (path seed)
  (values (directory-list path :add-path? #t :children? #t)
          seed))
```

Note that *lister* shouldn't return the given path itself (`"."`) nor the parent directory (`".."`), or the recursion wouldn't terminate. Also note *lister* is expected to return a path accessible from the current directory, i.e. if *path* is `"/usr/lib/foo"` and it contains `"libfoo.a"` and `"libfoo.so"`, *lister* should return `'("/usr/lib/foo/libfoo.a" "/usr/lib/foo/libfoo.so")`.

The keyword argument *follow-link?* is used to determine whether *lister* should be called on a symbolic link pointing to a directory. When *follow-link?* is true (default), *lister* is called with the symbolic link if it points to a directory. When *follow-link?* is false, *proc* is not called.

The following example returns a list of pathnames of the emacs backup files (whose name ends with `"~"`) under the given path.

```
(use srfi-13) ;; for string-suffix?
(directory-fold path
                (lambda (entry result)
                  (if (string-suffix? "~" entry)
                      (cons entry result)
                      result))
                '())
```

The following example lists all the files and directories under the given pathname. Note the use of *lister* argument to include the directory path itself in the result.

```
(directory-fold path cons '()
  :lister (lambda (path seed)
            (values (directory-list path :add-path? #t :children? #t)
                    (cons path seed))))
```

`make-directory*` *name :optional perm* [Function]
`create-directory*` *name :optional perm* [Function]

{`file.util`} Creates a directory *name*. If the intermediate path to the directory doesn't exist, they are also created (like `mkdir -p` command on Unix). If the directory *name* already exist, these procedure does nothing. *Perm* specifies the integer flag for permission bits of the directory.

`remove-directory*` *name* [Function]
`delete-directory*` *name* [Function]

{`file.util`} Deletes directory *name* and its content recursively (like `rm -r` command on Unix). Symbolic links are not followed.

copy-directory* *src dst :key if-exists backup-suffix safe keep-timestamp keep-mode follow-link?* [Function]

{file.util} If *src* is a regular file, copies its content to *dst*, just like `copy-file` does. If *src* is a directory, recursively descends it and copy the file tree to *dst*. Basically it mimics the behavior of `cp -r` command.

If there's any symbolic links under *src*, the link itself is copied instead of the file pointed to by it, unless a true value is given to the *follow-link?* keyword argument, i.e. the default value of *follow-link?* is `#f`. (Note that this is opposite to the `copy-file`, in which *follow-link?* is true by default.)

The meanings of the other keyword arguments are the same as `copy-file`. See the entry of `copy-file` for the details.

create-directory-tree *dir spec* [Function]

{file.util} Creates a directory tree under *dir* according to *spec*. This procedure is useful to set up certain directory hierarchy at once.

The *spec* argument is an S-expression with the following structure:

```
<spec> : <name>                              ; empty file
       | (<name> <option> ...)               ; empty file
       | (<name> <option> ... <string>)      ; file with content
       | (<name> <option> ... <procedure>)   ; file with generated content
       | (<name> <option> ... (<spec> ...)) ; directory

<name> : string or symbol

<option> ... : keyword-value alternating list
```

With the first and second form of *spec*, an empty file is created with the given name. With the third form of *spec*, the string becomes the content of the file.

With the fourth form of *spec*, the procedure is called with the pathname as an argument, and output to the current output port within the procedure is written to the created file. The pathname is relative to the *dir* argument. At the time the procedure is called, its parent directory is already created.

The last form of *spec* creates a named directory, then creates its children recursively according to the specs.

With *option*s you can control attributes of created files/directories. Currently the following options are recognized.

`:mode` *mode*
: Takes integer as permission mode bits.

`:owner` *uid*
`:group` *gid*
: Takes integer uid/gid of the owner/group of the file/directory. Calling process may need special privilege to change the owner and/or group.

`:symlink` *path*
: This is only valid for file spec, and it causes `create-directory-tree` to create a named symbolic link whose content is *path*.

check-directory-tree *dir spec* [Function]

{file.util} Checks if a directory hierarchy according to *spec* exists under *dir*. Returns `#t` if it exists, or `#f` otherwise.

The format of *spec* is the same as `create-directory-tree` described above.

If *spec* contains options, the attributes of existing files/directories are also checked if they match the given options.

12.24.2 Pathname utilities

`build-path` *base-path component . . .* [Function]

{`file.util`} Appends pathname components *component* to the *base-path*. *Component* can be a symbol `up` or `same`; in Unix, they are synonym to `".."` and `"."`. This API is taken from MzScheme.

`absolute-path?` *path* [Function]
`relative-path?` *path* [Function]

{`file.util`} Returns `#t` if *path* is absolute or relative, respectively.

`expand-path` *path* [Function]

{`file.util`} Expands tilda-notation of *path* if it contains one. Otherwise, *path* is returned. This function does not check if *path* exists and/or readable.

`resolve-path` *path* [Function]

{`file.util`} Expands *path* like `expand-path`, then resolve symbolic links for every components of the path. If *path* does not exist, or contains dangling link, or contains unreadable directory, an error is signaled.

`simplify-path` *path* [Function]

{`file.util`} Remove 'up' (`".."`) components and 'same' (`"."`) components from *path* as much as possible. This function does not access the filesystem.

`decompose-path` *path* [Function]

{`file.util`} Returns three values; the directory part of *path*, the basename without extension of *path*, and the extension of *path*. If the pathname doesn't have an extension, the third value is `#f`. If the pathname ends with a directory separator, the second and third values are `#f`. (Note: This treatment of the trailing directory separator differs from `sys-dirname`/`sys-basename`; those follow popular shell's convention, which ignores trailing slashes.)

```
(decompose-path "/foo/bar/baz.scm")
  ⇒ "/foo/bar", "baz", "scm"
(decompose-path "/foo/bar/baz")
  ⇒ "/foo/bar", "baz", #f

(decompose-path "baz.scm")
  ⇒ ".", "baz", "scm"
(decompose-path "/baz.scm")
  ⇒ "/", "baz", "scm"

;; Boundary cases
(decompose-path "/foo/bar/baz.")
  ⇒ "/foo/bar", "baz", ""
(decompose-path "/foo/bar/.baz")
  ⇒ "/foo/bar", ".baz", #f
(decompose-path "/foo/bar.baz/")
  ⇒ "/foo/bar.baz", #f, #f
```

path-extension *path* [Function]
path-sans-extension *path* [Function]

{file.util} Returns an extension of *path*, and a pathname of *path* without extension, respectively. If *path* doesn't have an extension, #f and *path* is returned respectively.

```
(path-extension "/foo/bar.c")       ⇒ "c"
(path-sans-extension "/foo/bar.c")  ⇒ "/foo/bar"

(path-extension "/foo/bar")         ⇒ #f
(path-sans-extension "/foo/bar")    ⇒ "/foo/bar"
```

path-swap-extension *path newext* [Function]

{file.util} Returns a pathname in which the extension of *path* is replaced by *newext*. If *path* doesn't have an extension, "." and *newext* is appended to *path*.

If *newext* is #f, it returns *path* without extension.

```
(path-swap-extension "/foo/bar.c" "o")  ⇒ "/foo/bar.o"
(path-swap-extension "/foo/bar.c" "")   ⇒ "/foo/bar."
(path-swap-extension "/foo/bar.c" #f)   ⇒ "/foo/bar"

(path-swap-extension "/foo/bar" "o")  ⇒ "/foo/bar.o"
(path-swap-extension "/foo/bar" "")   ⇒ "/foo/bar."
(path-swap-extension "/foo/bar" #f)   ⇒ "/foo/bar"
```

find-file-in-paths *name :key paths pred extensions* [Function]

{file.util} Looks for a file that has name *name* in the given list of pathnames *paths* and that satisfies a predicate *pred*. If found, the absolute pathname of the file is returned. Otherwise, #f is returned.

If *name* is an absolute path, only the existence of *name* and whether it satisfies *pred* are checked.

The default value of *paths* is taken from the environment variable PATH, and the default value of *pred* is file-is-executable? (see Section 12.24.3 [File attribute utilities], page 698). That is, find-file-in-paths searches the named executable file in the command search paths by default.

```
(find-file-in-paths "ls")
  ⇒ "/bin/ls"

;; example of searching user preference file of my application
(find-file-in-paths "userpref"
  :paths `(,(expand-path "~/.myapp")
           "/usr/local/share/myapp"
           "/usr/share/myapp")
  :pred  file-is-readable?)
```

The *extensions* keyword argument may list alternative extensions added to *name*. For example, the following example searches not only notepad, but also notepad.exe and notepad.com, in the PATH. If an alternate name is found, the returned pathname contains the extension.

```
(find-file-in-paths "notepad" :extensions '("exe" "com"))
```

For each path, the name and the alternative names are checked in order. That is, if there are /bin/b.com and /usr/bin/b.exe and *paths* is ("/bin" "/usr/bin"), you'll get /bin/b.com when you search b with extensions ("exe" "com").

`null-device` [Function]

{`file.util`} Returns a name of the *null* device. On unix platforms (including cygwin) it returns `"/dev/null"`, and on Windows native platforms (including mingw) it returns `"NUL"`.

`console-device` [Function]

{`file.util`} Returns a name of the console device. On unix platforms (including cygwin) it returns `"/dev/tty"`, and on Windows native platforms (including mingw) it returns `"CON"`.

This function does not guarantee the device is actually available to the calling process.

12.24.3 File attribute utilities

`file-type` *path :key follow-link?* [Function]
`file-perm` *path :key follow-link?* [Function]
`file-mode` *path :key follow-link?* [Function]
`file-ino` *path :key follow-link?* [Function]
`file-dev` *path :key follow-link?* [Function]
`file-rdev` *path :key follow-link?* [Function]
`file-nlink` *path :key follow-link?* [Function]
`file-uid` *path :key follow-link?* [Function]
`file-gid` *path :key follow-link?* [Function]
`file-size` *path :key follow-link?* [Function]
`file-atime` *path :key follow-link?* [Function]
`file-mtime` *path :key follow-link?* [Function]
`file-ctime` *path :key follow-link?* [Function]

{`file.util`} These functions return the attribute of file/directory specified by *path*. The attribute name corresponds to the slot name of `<sys-stat>` class (see Section 6.25.4.4 [File stats], page 255). If the named path doesn't exist, `#f` is returned.

If *path* is a symbolic link, these functions queries the attributes of the file pointed by the link, unless an optional argument *follow-link?* is given and false.

MzScheme and Chicken have `file-size`. Chicken also has `file-modification-time`, which is `file-mtime`.

`file-is-readable?` *path* [Function]
`file-is-writable?` *path* [Function]
`file-is-executable?` *path* [Function]

{`file.util`} Returns `#t` if *path* exists and readable/writable/executable by the current effective user, respectively. This API is taken from STk.

`file-is-symlink?` *path* [Function]

{`file.util`} Returns `#t` if *path* exists and a symbolic link. See also `file-is-regular?` and `file-is-directory?` in Section 6.25.4.4 [File stats], page 255.

`file-eq?` *path1 path2* [Function]
`file-eqv?` *path1 path2* [Function]
`file-equal?` *path1 path2* [Function]

{`file.util`} Compares two files specified by *path1* and *path2*. `file-eq?` and `file-eqv?` checks if *path1* and *path2* refers to the identical file, that is, whether they are on the same device and have the identical inode number. The only difference is when the last component of *path1* and/or *path2* is a symbolic link, `file-eq?` doesn't resolve the link (so compares the links themselves) while *file-eqv?* resolves the link and compares the files referred by the link(s).

`file-equal?` compares *path1* and *path2* considering their content, that is, when two are not the identical file in the sense of `file-eqv?`, `file-equal?` compares their content and returns `#t` if all the bytes match.

The behavior of `file-equal?` is undefined when *path1* and *path2* are both directories. Later, it may be extended to scan the directory contents.

`file-mtime=?` *f1 f2* [Generic Function]
`file-mtime<?` *f1 f2* [Generic Function]
`file-mtime<=?` *f1 f2* [Generic Function]
`file-mtime>?` *f1 f2* [Generic Function]
`file-mtime>=?` *f1 f2* [Generic Function]

{`file.util`} Compares file modification time stamps. There are a bunch of methods defined, so each argument can be either one of the followings.

- String pathname. The mtime of the specified path is used.
- `<sys-stat>` object (see Section 6.25.4.4 [File stats], page 255). The mtime is taken from the stat structure.
- `<time>` object. The time is used as the mtime.
- Number. It is considered as the number of seconds since Unix Epoch, and used as mtime.

```
;; compare "foo.c" is newer than "foo.o"
(file-mtime>? "foo.c" "foo.o")

;; see if "foo.log" is updated within last 24 hours
(file-mtime>? "foo.c" (- (sys-time) 86400))
```

`file-ctime=?` *f1 f2* [Generic Function]
`file-atime=?` *f1 f2* [Generic Function]

{`file.util`} Same as `file-mtime=?`, except these checks file's change time and access time, respectively. All the variants of `<`, `<=`, `>`, `>=` are also defined.

12.24.4 File operations

`touch-file` *path :key (time #f) (type #f) (create #t)* [Function]
`touch-files` *paths :key (time #f) (type #f) (create #t)* [Function]

{`file.util`} Updates timestamp of *path*, or each path in the list *paths*, to the current time. If the specified path doesn't exist, a new file with size zero is created, unless the keyword argument *create* is `#f`.

If the keyword argument *time* is given and not `#f`, it must be a nonnegative real number. It is used as the timestamp value instead of the current time.

The keyword argument *type* can be `#f` (default), a symbol `atime` or `mtime`. If it is a symbol, only the access time or modification time is updated.

Note: `touch-files` processes one file at a time, so the timestamp of each file may not be exactly the same.

These procedures are built on top of the system call `sys-utime` (see Section 6.25.4.4 [File stats], page 255).

`copy-file` *src dst :key if-exists backup-suffix safe keep-timestamp keep-mode follow-link?* [Function]

{`file.util`} Copies file from *src* to *dst*. The source file *src* must exist. The behavior when the destination *dst* exists varies by the keyword argument *if-exists*;

`:error` (Default) Signals an error when *dst* exists.

`:supersede`
Replaces *dst* to the copy of *src*.

`:backup` Keeps *dst* by renaming it.

`:append` Append the *src*'s content to the end of *dst*.

`#f` Doesn't copy and returns `#f` when *dst* exists.

`Copy-file` returns `#t` after completion.

If *src* is a symbolic link, `copy-file` follows the symlink and copies the actual content by default. An error is raised if *src* is a dangling symlink.

Giving `#f` to the keyword argument *follow-link?* makes `copy-file` to copy the link itself. It is possible that *src* is a dangling symlink in this case.

If *if-exists* is `:backup`, the keyword argument *backup-suffix* specifies the suffix attached to the *dst* to be renamed. The default value is `".orig"`.

By default, `copy-file` starts copying to *dst* directly. However, if the keyword argument *safe* is a true value, it copies the file to a temporary file in the same directory of *dst*, then renames it to *dst* when copy is completed. (When *safe* is true and *if-exists* is `:append`, we first copy the content of *dst* to a temporary file if *dst* exists, appends the content of *src*, then renames the result to *dst*). If copy is interrupted for some reason, the filesystem is "rolled back" properly.

If the keyword argument *keep-timestamp* is true, `copy-file` sets the destination's timestamp to the same as the source's timestamp after copying.

If the keyword argument *keep-mode* is true, the destination file's permission bits are set to the same as the source file's. If it is false (default), the destination file's permission remains the same if the destination already exists and the *safe* argument is false, otherwise it becomes `#o666` masked by umask settings.

`move-file` *src dst :key if-exists backup-suffix* [Function]

{`file.util`} Moves file *src* to *dst*. The source *src* must exist. The behavior when *dst* exists varies by the keyword argument *if-exists*, as follows.

`:error` (Default) Signals an error when *dst* exists.

`:supersede`
Replaces *dst* by `src`.

`:backup` Keeps *dst* by renaming it.

`#f` Doesn't move and returns `#f` when *dst* exists.

`Move-file` returns `#t` after completion.

If *if-exists* is `:backup`, the keyword argument *backup-suffix* specifies the suffix attached to the *dst* to be renamed. The default value is `".orig"`.

The file *src* and *dst* can be on the different filesystem. In such a case, `move-file` first copies *src* to the temporary file on the same directory as *dst*, then renames it to *dst*, then removes *src*.

`remove-file` *filename* [Function]
`delete-file` *filename* [Function]

[R7RS file] {`file.util`} Removes the named file. An error is signalled if *filename* does not exist, is a directory, or cannot be deleted with other reasons such as permissions. R7RS defines `delete-file`.

Compare with `sys-unlink` (see Section 6.25.4.2 [Directory manipulation], page 252), which doesn't raise an error when the named file doesn't exist.

`remove-files` *paths* [Function]
`delete-files` *paths* [Function]

{`file.util`} Removes each path in a list *paths*. If the path is a file, it is `unlink`ed. If it is a directory, its contents are recursively removed by `remove-directory*`. If the path doesn't exist, it is simply ignored.

`delete-files` is just an alias of `remove-files`.

`file->string` *filename options . . .* [Function]
`file->list` *reader filename options . . .* [Function]
`file->string-list` *filename options . . .* [Function]
`file->sexp-list` *filename options . . .* [Function]

{`file.util`} Convenience procedures to read from a file *filename*. They first open the named file, then call `port->string`, `port->list`, `port->string-list` and `port->sexp-list` on the opened file, respectively. (see Section 6.22.7.4 [Input utility functions], page 231). The file is closed if all the content is read or an error is signaled during reading.

Those procedures take the same keyword arguments as `call-with-input-file`. When the named file doesn't exist, the behavior depends on *:if-does-not-exist* keyword argument—an error is signaled if it is `:error`, and `#f` is returned if the argument is `#f`.

`string->file` *filename string options . . .* [Function]
`list->file` *writer filename lis options . . .* [Function]
`string-list->file` *filename lis options . . .* [Function]
`sexp-list->file` *filenme lis options . . .* [Function]

{`file.util`} Opposite of `file->string` etc. They are convenient to quickly write out things into a file.

NB: The name `string->file` etc. might suggest they would take the object to be written as the first argument. We decided to put *filename* first, since in the situations where these procedures are used, it is more likely that one want to write literal data, which would be bigger than the filename itself.

The options part is passed to `call-with-output-file` as is. For example, the following code appends the text when `foo.txt` already exists:

```
(string->file "foo.txt" "New text to append\n"
              :if-exists :append)
```

The `list->file` takes *writer* argument, which is a procedure that receives two arguments, an element from the list *lis*, and an output port. It should write out the element to the port in a suitable way. The `string-list->file` and `sexp-list->file` are specialized versions of `list->file`, where `string-list->file` uses `(^[s p] (display s p) (newline p))` as *writer*, and `sexp-list->file` uses `(^[s p] (write s p) (newline p))` as *writer*.

12.24.5 Temporary files and directories

`temporary-directory` [Parameter]

{`file.util`} A parameter that keeps the name of the directory that can be used to create a temporary files. The default value is the one returned from `sys-tmpdir` (see Section 6.25.4.3 [Pathnames], page 253). The difference of `sys-tmpdir` is that, since this is a parameter, it can be overridden by application during execution. Libraries are recommended to use this instead of `sys-tmpdir` for greater flexibility.

`call-with-temporary-file` *proc :key directory prefix* [Function]

{`file.util`} Creates a temporary file with unique name and opens it for output, then calls *proc* with the output port and the temporary file's name. The temporary file is removed after either *proc* returns or raises an uncaught error. Returns the value(s) *proc* returns.

The temporary file is created in the directory *directory*, with the name *prefix* followed by several random alphanumeric characters. When omitted, the value of `(temporary-directory)` is used for *directory*, and `"gtemp"` for *prefix*.

The name passed to *proc* consists of *directory* and the file's name. So whether the name is absolute or relative pathname depends on the value of *directory*.

```
(call-with-temporary-file (^[_ name] name)
  ⇒ Something like "/tmp/gtemp4dSpMh"
```

You can keep the output file by renaming it in *proc*. But if doing so, make sure to specify *directory* so that the temporary file is created in the same directory as the final output; rename may not work across filesystems.

Internally, it calls `sys-mkstemp` to create a unique file. See Section 6.25.4.2 [Directory manipulation], page 252, for the details.

`call-with-temporary-directory` *proc :key directory prefix* [Function]

{`file.util`} Creates a temporary directory with unique name, then calls *proc* with the name. The temporary directory and its contents are removed after either *proc* returns or raises an uncaught error. Returns the value(s) *proc* returns.

The temporary directory is created in the directory *directory*, with the name *prefix* followed by several random alphanumeric characters. When omitted, the value of `(temporary-directory)` is used for *directory*, and `"gtemp"` for *prefix*.

The name passed to *proc* consists of *directory* and the directory name. So whether the name is absolute or relative pathname depends on the value of *directory*.

Internally, it calls `sys-mkdtemp` to create a unique file. See Section 6.25.4.2 [Directory manipulation], page 252, for the details.

12.24.6 Lock files

Exclusivity of creating files or directories is often used for inter-process locking. The following procedure provides a packaged interface for it.

`with-lock-file` *lock-name thunk :key type retry-interval retry-limit secondary-lock-name retry2-interval retry2-limit perms abandon-timeout* [Function]

{`file.util`} Exclusively creates a file or a directory (*lock file*) with *lock-name*, then executes *thunk*. After *thunk* returns, or an error is thrown in it, the lock file is removed. When *thunk* returns normally, its return values become the return values of `with-lock-file`.

If the lock file already exists, `with-lock-file` waits and retries getting the lock until timeout reaches. It can be configured by the keyword arguments.

There's a chance that `with-lock-file` leaves the lock file when it gets a serious error situation and doesn't have the opportunity to clean up. You can allow `with-lock-file` to *steal* the lock if its timestamp is too old; say, if you know that the applications usually locks just for seconds, and you find the lock file is 10 minutes old, then it's likely that the previous process was terminated abruptly and couldn't clean it up. You can also configure this behavior by the keyword arguments.

Internally, *two* lock files are used to implement this stealing behavior safely. The creation and removal of the primary lock file (named by *lock-name* argument) are guarded by the secondary lock file (named by *secondary-lock-file* argument, defaulted by `.2` suffix attached to *lock-name*). The secondary lock prevents more than one process steals the same primary lock file simultaneously.

The secondary lock is acquired for a very short period so there's much less chance to be left behind by abnormal terminations. If it happens, however, we just give up; we don't steal the secondary lock.

If `with-lock-file` couldn't get a lock before timeout, a `<lock-file-failure>` condition is thrown.

Here's a list of keyword arguments.

type

It can be either one of the symbols `file` or `directory`.

If it is `file`, we use a lock file, relying on the `O_EXCL` exclusive creation flag of `open(2)`. This is the default value. It works for most platforms; however, some NFS implementation may not implement the exclusive semantics properly.

If it is `directory`, we use a lock directory, relying on the atomicity of `mkdir(2)`. It should work for any platforms, but it may be slower than `file`.

retry-interval
retry-limit

Accepts a nonnegative real number that specifies either the interval to attempt to acquire the primary lock, or the maximum time we should keep retrying, respectively, in seconds. The default value is 1 second interval and 10 second limit. To prevent retrying, give 0 to *retry-limit*.

secondary-lock-name

The name of the secondary lock file (or directory). If omitted, *lock-name* with a suffix `.2` attached is used. Note: The secondary lock name must be agreed on all programs that locks the same (primary) lock file. I recommend to leave this to the default unless there's a good reason to do otherwise.

retry2-interval
retry2-limit

Like *retry-interval* and *retry-limit*, but these specify interval and timeout for the secondary lock file. The possibility of secondary lock file collision is usually pretty low, so you would hardly need to tweak these. The default values are 1 second interval and 10 second limit.

perms

Specify the permission bitmask of the lock file or directory, in a nonnegative exact integer. The default is `#o644` for a lock file and `#o755` for a lock directory.

Note that to control who can acquire/release/steal the lock, what matters is the permission of the directory in which the lock file/directory, not the permission of the lock file/directory itself.

abandon-timeout

Specifies the period in seconds in a nonnegative real number. If the primary lock file is older than that, `with-lock-file` steals the lock. To prevent stealing, give `#f` to this argument. The default value is 600 seconds.

`<lock-file-failure>` [Condition type]

{`file.util`} A condition indicating that `with-lock-file` couldn't obtain the lock. Inherits `<error>`.

`lock-file-name` [Instance Variable of `<lock-file-failure>`]

The primary lock file name.

Gauche also provides OS-supported file locking feature, `fcntl` lock, via `gauche.fcntl` module. Whether you want to use `fcntl` lock or `with-lock-file` will depend on your application.

These are the advantages of the `fcntl` lock:

- The lock is removed when the process dies without explicitly unlocking it.

- You can directly lock the file you're touching.
- You can lock a part of a file.
- You can have shared (read) and exclusive (write) locks.

In common situations, probably the most handy property is the first one; you don't need to worry about leaving lock behind unexpected process termination.

However, there are a couple of shortcomings in `fcntl` locks.

- It is not guaranteed to work across different platforms, and/or NFS-mounted filesystems.
- The lock is per-process, per-file, and non-recursive. If you have a lock in a file, then calls a library that also locks the file, the lock always succeeds. Worse, if the library unlocks the file, the lock is completely removed, while the caller doesn't know about it. It also means that, in order to prevent multiple threads in a process from accessing the same file, you have to use mutex along the fcntl lock.

Especially because of the second point, it is very difficult to use `fcntl` lock unless you have total control over and knowledge of the entire application. It is ok to use the `fcntl` lock by the application code to lock the application-specific file. Library developers have difficulty, however, to make sure any potential user of the library won't try to lock the same file as the library tries to lock (usually it's impossible).

12.25 `math.const` - Mathematic constants

`math.const` [Module]

This module defines several commonly-used mathematic constants.

`pi` [Constant]
`pi/2` [Constant]
`pi/4` [Constant]
`pi/180` [Constant]
`1/pi` [Constant]
`180/pi` [Constant]

{`math.const`} Bound to pi, pi/2, pi/4, pi/180, 1/pi and 180/pi, respectively.

`e` [Constant]

{`math.const`} Napier's constant.

12.26 `math.mt-random` - Mersenne Twister Random number generator

`math.mt-random` [Module]

Provides a pseudo random number generator (RNG) based on "Mersenne Twister" algorithm developed by Makoto Matsumoto and Takuji Nishimura. It is fast, and has huge period of 2^19937-1. See [MT], page 831, for details about the algorithm.

For typical use cases of random number generators, we recommend to use `srfi-27` which is implemented on top of this module and provides portable API. You should use this module directly only when you need functions that aren't available through `srfi-27`.

`<mersenne-twister>` [Class]

{`math.mt-random`} A class to encapsulate the state of Mersenne Twister RNG. Each instance of this class has its own state, and can be used as an independent source of random bits if initialized by individual seed.

The random seed value can be given at the instantiation time by `:seed` initialization argument, or by using `mt-random-set-seed!` described below.

```
(define m (make <mersenne-twister> :seed (sys-time)))

(mt-random-real m) ⇒ 0.10284287848537865
(mt-random-real m) ⇒ 0.463227748348805
(mt-random-real m) ⇒ 0.8628500643709712
...
```

`mt-random-set-seed!` *mt seed* [Function]

{`math.mt-random`} Sets random seed value *seed* to the Mersenne Twister RNG *mt*. *Seed* can be an arbitrary positive exact integer, or arbitrary length of u32vector (see Section 11.2 [Homogeneous vectors], page 590). If it is an integer, the lower 32bits are used for initialization. If it is a u32vector, up to 624 elements are used for initialization.

`mt-random-get-state` *mt* [Function]
`mt-random-set-state!` *mt state* [Function]

{`math.mt-random`} Retrieves and reinstalls the state of Mersenne Twister RNG *mt*. The state is represented by a u32vector of 625 elements. The state can be stored elsewhere, and then restored to an instance of `<mersenne-twister>` to continue to generate the pseudo random sequence.

`mt-random-real` *mt* [Function]
`mt-random-real0` *mt* [Function]

{`math.mt-random`} Returns a random real number between 0.0 and 1.0. 1.0 is not included in the range. `Mt-random-real` doesn't include 0.0 either, while `mt-random-real0` does. Excluding 0.0 is from the draft SRFI-27.

`mt-random-integer` *mt range* [Function]

{`math.mt-random`} Returns a random exact positive integer between 0 and *range*-1. *Range* can be any positive exact integer.

`mt-random-fill-u32vector!` *mt u32vector* [Function]
`mt-random-fill-f32vector!` *mt f32vector* [Function]
`mt-random-fill-f64vector!` *mt f64vector* [Function]

{`math.mt-random`} Fills the given uniform vector by the random numbers. For `mt-random-fill-u32vector!`, the elements are filled by exact positive integers between 0 and 2^32-1. For `mt-random-fill-f32vector!` and `mt-random-fill-f64vector!`, it is filled by an inexact real number between 0.0 and 1.0, exclusive.

If you need a bunch of random numbers at once, these are much faster than getting one by one.

12.27 `math.prime` - Prime numbers

`math.prime` [Module]

This module provides utilities related to prime numbers.

Sequence of prime numbers

`*primes*` [Variable]

{`math.prime`} An infinite lazy sequence of primes.

```
;; show 10 prime numbers from 100-th one.
(take (drop *primes* 100) 10)
 ⇒ (547 557 563 569 571 577 587 593 599 601)
```

`reset-primes` [Function]

{`math.prime`} Once you take a very large prime out of `*primes*`, all primes before that has been calculated remains in memory, since the head of sequence is held in `*primes*`. Sometimes you know you need no more prime numbers and you wish those calculated ones to be garbage-collected. Calling `reset-primes` rebinds `*primes*` to unrealized lazy sequence, allowing the previously realized primes to be GCed.

`primes` [Function]

{`math.prime`} Returns a fresh lazy sequence of primes. It is useful when you need certain primes in a short period of time—if you don't keep a reference to the head of the returned sequence, it will be garbage collected after you've done with the primes. (Note that calculation of a prime number needs the sequence of primes from the beginning, so even if your code only keep a reference in the middle of the sequence, the entire sequence will be kept in the thunk within the lazy sequence—you have to release all references in order to make the sequence GCed.)

On the other hand, each sequence returned by `primes` are realized individually, duplicating calculation.

The rule of thumb is—if you use primes repeatedly throughout the program, just use `*primes*` and you'll save calculation. If you need primes one-shot, call `primes` and abandon it and you'll save space.

Testing primality

`small-prime?` *n* [Function]

{`math.prime`} For relatively small positive integers (below `*small-prime-bound*`, to be specific), this procedure determines if the input is prime or not, quickly and deterministically. If *n* is on or above the bound, this procedure returns `#f`.

This can be used to quickly filter out known primes; it never returns `#t` on composite numbers (while it may return `#f` on large prime numbers). Miller-Rabin test below can tell if the input is composite for sure, but it may return `#t` on some composite numbers.

`*small-prime-bound*` [Variable]

{`math.prime`} For all positive integers below this value (slightly above 3.4e14 in the current implementation), `small-prime?` can determines whether it is a prime or not.

`miller-rabin-prime?` *n :key num-tests random-integer* [Function]

{`math.prime`} Check if an exact integer *n* is a prime number, using probabilistic Miller-Rabin algorithm (*n* must be greater than 1). If this procedure returns `#f`, *n* is a composite number. If this procedure returns `#t`, *n* is *likely* a prime, but there's a small probability that it is a false positive.

Note that if *n* is smaller than a certain number (`*small-prime-bound*`), the algorithm is deterministic; if it returns `#t`, *n* is certainly a prime.

If *n* is greater than or equal to `*small-prime-bound*`, we use a probabilistic test. We choosing random base integer to perform Miller-Rabin test up to 7 times by default. You can change the number of tests by the keyword argument *num-tests*. The error probability (to return `#t` for a composite number) is at most `(expt 4 (- num-tests))`.

For a probabilistic test, `miller-rabin-prime?` uses its own fixed random seed by default. We chose fixed seed so that the behavior can be reproducible. To change the random sequence, you can provide your own random integer generator to the *random-integer* keyword argument. It must be a procedure that takes a positive integer *k* and returns a random integer from 0 to *k-1*, including.

`bpsw-prime?` *n* [Function]

{`math.prime`} Check if an exact integer *n* is a prime number, using Baillie-PSW primality test (`http://www.trnicely.net/misc/bpsw.html`). It is deterministic, and returns the definitive answer below 2^64 (around 1.8e19). For larger integers this can return `#t` on a composite number, although such number hasn't been found yet. This never returns `#f` on a prime number.

This is slower than Miller-Rabin but fast enough for casual use, so it is handy when you want a definitive answer below the above range.

Factorization

`naive-factorize` *n :optional divisor-limit* [Function]

{`math.prime`} Factorize a positive integer *n* by trying to divide it with all primes up to `(sqrt n)`. Returns a list of prime factors, smaller ones first.

```
(naive-factorize 142857)
  ⇒ (3 3 3 11 13 37)
```

Although this is pretty naive method, this works well as far as any of *n*'s factors are up to the order of around `1e7`. For example, the following example runs in about 0.4sec on 2.4GHz Core2 machine. (The first time will take about 1.3sec to realize lazy prime sequences.)

```
(naive-factorize 3644357367494986671013))
  ⇒ (10670053 10670053 32010157)
```

Of course, if *n* includes any factors above that order, the performance becomes abysmal. So it is better to use this procedure below 1e14 or so.

Alternatively, you can give *divisor-limit* argument that specifies the upper bound of the prime number to be tried. If it is given, `naive-factorize` leaves a factor *f* as is if it can't be divided by any primes less than or equal to *divisor-limit*. So, the last element of the returned list may be composite number. This is handy to exclude trivial factors before applying more sophisticated factorizing algorithms.

```
(naive-factorize 825877877739 1000)
  ⇒ (3 43 6402154091)

;; whereas
(naive-factorize 825877877739)
  ⇒ (3 43 4591 1394501)
```

The procedure also memoizes the results on smaller *n* to make things faster.

`mc-factorize` *n* [Function]

{`math.prime`} Factorize a positive integer *n* using the algorithm described in R. P. Brent, An improved Monte Carlo factorization algorithm, BIT 20 (1980), 176-184. `http://maths-people.anu.edu.au/~brent/pub/pub051.html`.

This one is capable to handle much larger range than `naive-factorize`, somewhere around 1e20 or so.

Since this method is probabilistic, the execution time may vary on the same *n*. But it will always return the definitive results as far as every prime factor of *n* is smaller than 2^64.

At this moment, if *n* contains a prime factor greater than 2^64, this routine would keep trying factorizing it forever. Practical applications should have some means to interrupt the function and give it up after some time bounds. This will be addressed once we have deterministic primality test.

Miscellaneous

`jacobi` *a n* [Function]

{`math.prime`} Calculates Jacobi symbol `(a/n)` (`http://en.wikipedia.org/wiki/Jacobi_symbol`).

`totient` *n* [Function]

{`math.prime`} Euler's totient function of nonnegative integer *n*.

The current implementation relies on `mc-factorize` above, so it may take very long if *n* contains large prime factors.

12.28 `os.windows` - Windows support

`os.windows` [Module]

This module is only available on Windows-native Gauche, and provides Windows-specific procedures. You can check `gauche.os.windows` feature with `cond-expand` macro (see Section 4.12 [Feature conditional], page 68) to conditionalize windows-specific code.

```
(cond-expand
  [gauche.os.windows
   (use os.windows)
   ... Windows-specific code ...]
  [else
   ... Unix code ...])
```

Currently there aren't enough procedures provided here, but eventually we want to support simple scripting on Windows.

Unless otherwise noted, when Windows API returns an error value, a `<system-error>` condition is thrown.

12.28.1 Windows dialogs

Currenly we only have MessageBox API.

`sys-message-box` *window message :optional caption flags* [Function]

{`os.windows`} Calls Windows MessageBox API. The *window* argument should be a handle for a window, or `#f`; at the moment we don't provide any API that retrieves window handles, so you should always pass `#f` here. The *message* argument takes a string for the content of the message box. Optional *caption* argument takes a string to be used in the window title.

The *flags* argument is an integer; it should be `logior` of values from one or more of the following groups. See the Windows reference manual for the details.

Buttons `MB_ABORTRETRYIGNORE`, `MB_CANCELTRYCONTINUE`, `MB_HELP`, `MB_OK` (default), `MB_OKCANCEL`, `MB_RETRYCANCEL`, `MB_YESNO`, `MB_YESNOCANCEL`

Icon Default is no icon. Possible values: `MB_ICONEXCLAMATION`, `MB_ICONWARNING`, `MB_ICONINFORMATION`, `MB_ICONASTERISK`, `MB_ICONQUESTION`, `MB_ICONSTOP`, `MB_ICONERROR`, `MB_ICONHAND`

Default button
`MB_DEFBUTTON1` (default), `MB_DEFBUTTON2`, `MB_DEFBUTTON3`, `MB_DEFBUTTON4`

Modality `MB_APPLMODAL` (default), `MB_SYSTEMMODAL`, `MB_TASKMODAL`

Other options
`MB_DEFAULT_DESKTOP_ONLY`, `MB_RIGHT`, `MB_RTLREADING`, `MB_SETFOREGROUND`, `MB_TOPMOST`, `MB_SERVICE_NOTIFICATION`

Return value is one of the following integer constants, indicating which button is pressed: `IDABORT`, `IDCANCEL`, `IDCONTINUE`, `IDIGNORE`, `IDNO`, `IDOK`, `IDRETRY`, `IDTRYAGAIN`, or `IDYES`

12.28.2 Windows console API

Most of these procedures corresponds to Windows Console API one-to-one. See the Windows reference for the detail description of what each API does.

Attaching and detaching

`sys-alloc-console` [Function]
`sys-free-console` [Function]
[Windows] {`os.windows`} Calls `AllocConsole` and `FreeConsole`, respectively.

`sys-generate-console-ctrl-event` *event pgid* [Function]
[Windows] {`os.windows`}

`CTRL_C_EVENT` [Constant]
`CTRL_BREAK_EVENT` [Constant]
[Windows] {`os.windows`}

Console codepage

`sys-get-console-cp` [Function]
`sys-get-console-output-cp` [Function]
`sys-set-console-cp` *codepage* [Function]
`sys-set-console-output-cp` *codepage* [Function]
[Windows] {`os.windows`}

`sys-get-console-cursor-info` *handle* [Function]
`sys-set-console-cursor-info` *handle size visible* [Function]
[Windows] {`os.windows`}

`sys-set-console-cursor-position` *handle x y* [Function]
[Windows] {`os.windows`}

Console mode

`sys-get-console-mode` *handle* [Function]
`sys-set-console-mode` *handle mode* [Function]
[Windows] {`os.windows`}

`ENABLE_LINE_INPUT` [Constant]
`ENABLE_ECHO_INPUT` [Constant]
`ENABLE_PROCESSED_INPUT` [Constant]
`ENABLE_WINDOW_INPUT` [Constant]
`ENABLE_MOUSE_INPUT` [Constant]
`ENABLE_PROCESSED_OUTPUT` [Constant]
`ENABLE_WRAP_AT_EOL_OUTPUT` [Constant]
[Windows] {`os.windows`}

Screen buffer

`sys-create-console-screen-buffer` *desired-access share-mode inheritable* [Function]
[Windows] {`os.windows`}

`GENERIC_READ` [Constant]
`GENERIC_WRITE` [Constant]
[Windows] {`os.windows`}

`FILE_SHARE_READ` [Constant]
`FILE_SHARE_WRITE` [Constant]
[Windows] {`os.windows`}

`sys-set-console-active-screen-buffer` *handle* [Function]
[Windows] {`os.windows`}

`sys-scroll-console-screen-buffer` *handle scroll-rectangle clip-rectangle x y fill* [Function]
[Windows] {`os.windows`}

`<win:console-screen-buffer-info>` [Class]
[Windows] {`os.windows`}

`size.x` [Instance Variable of `<win:console-screen-buffer-info>`]
`size.y` [Instance Variable of `<win:console-screen-buffer-info>`]

`cursor-position.x` [Instance Variable of `<win:console-screen-buffer-info>`]
`cursor-position.y` [Instance Variable of `<win:console-screen-buffer-info>`]

`attributes` [Instance Variable of `<win:console-screen-buffer-info>`]

`window.left` [Instance Variable of `<win:console-screen-buffer-info>`]
`window.top` [Instance Variable of `<win:console-screen-buffer-info>`]
`window.right` [Instance Variable of `<win:console-screen-buffer-info>`]
`window.bottom` [Instance Variable of `<win:console-screen-buffer-info>`]

`maximum-window-size.x` [Instance Variable of `<win:console-screen-buffer-info>`]
`maximum-window-size.y` [Instance Variable of `<win:console-screen-buffer-info>`]

`FOREGROUND_BLUE` [Constant]
`FOREGROUND_GREEN` [Constant]
`FOREGROUND_RED` [Constant]
`FOREGROUND_INTENSITY` [Constant]
`BACKGROUND_BLUE` [Constant]
`BACKGROUND_GREEN` [Constant]
`BACKGROUND_RED` [Constant]
`BACKGROUND_INTENSITY` [Constant]
[Windows] {`os.windows`}

`sys-get-console-screen-buffer-info` *handle* [Function]
[Windows] {`os.windows`}

`sys-get-largest-console-window-size` *handle* [Function]
[Windows] {`os.windows`}

`sys-set-screen-buffer-size` *handle x y* [Function]
[Windows] {`os.windows`}

Console input/output

`<win:input-record>` [Class]
[Windows] {`os.windows`}

`event-type` [Instance Variable of `<win:input-record>`]

`key.down` [Instance Variable of `<win:input-record>`]
`key.repeat-count` [Instance Variable of `<win:input-record>`]
`key.virtual-key-code` [Instance Variable of `<win:input-record>`]
`key.virtual-scan-code` [Instance Variable of `<win:input-record>`]
`key.unicode-char` [Instance Variable of `<win:input-record>`]
`key.ascii-char` [Instance Variable of `<win:input-record>`]
`key.control-key-state` [Instance Variable of `<win:input-record>`]

`mouse.x` [Instance Variable of `<win:input-record>`]
`mouse.y` [Instance Variable of `<win:input-record>`]
`mouse.button-state` [Instance Variable of `<win:input-record>`]
`mouse.control-key-state` [Instance Variable of `<win:input-record>`]
`mouse.event-flags` [Instance Variable of `<win:input-record>`]

`window-buffer-size.x` [Instance Variable of `<win:input-record>`]
`window-buffer-size.y` [Instance Variable of `<win:input-record>`]

`menu.command-id` [Instance Variable of `<win:input-record>`]

`focus.set-focus` [Instance Variable of `<win:input-record>`]

`sys-get-number-of-console-input-events` *handle* [Function]
[Windows] {`os.windows`}

`sys-get-number-of-console-mouse-buttons` [Function]
[Windows] {`os.windows`}

`sys-peek-console-input` *handle* [Function]
`sys-read-console-input` *handle* [Function]
[Windows] {`os.windows`}

`sys-read-console` *handle buf* [Function]
[Windows] {`os.windows`}

`sys-read-console-output` *handle buf w h x y region* [Function]
[Windows] {`os.windows`}

`sys-read-console-output-attribute` *handle buf x y* [Function]
[Windows] {`os.windows`}

`sys-read-console-output-character` *handle len x y* [Function]
[Windows] {`os.windows`}

`sys-set-console-text-attribute` *handle attr* [Function]
[Windows] {`os.windows`}

`sys-set-console-window-info` *handle absolute window* [Function]
[Windows] {`os.windows`}

`sys-write-console` *handle string* [Function]
[Windows] {`os.windows`}

`sys-write-console-output-character` *handle string x y* [Function]
[Windows] {`os.windows`}

`sys-fill-console-output-character` *handle char len x y* [Function]
[Windows] {`os.windows`}

`sys-fill-console-output-attribute` *handle attr len x y* [Function]
[Windows] {`os.windows`}

`sys-flush-console-input-buffer` *handle* [Function]
[Windows] {`os.windows`}

`sys-get-console-title` [Function]
[Windows] {`os.windows`}

`sys-set-console-title` *string* [Function]
[Windows] {`os.windows`}

Standard handles

`sys-get-std-handle` *which* [Function]
`sys-set-std-handle` *which handle* [Function]
[Windows] {`os.windows`}

`STD_INPUT_HANDLE` [Constant]
`STD_OUTPUT_HANDLE` [Constant]
`STD_ERROR_HANDLE` [Constant]
[Windows] {`os.windows`}

12.29 `rfc.822` - RFC822 message parsing

`rfc.822` [Module]
Defines a set of functions that parses and constructs the "Internet Message Format", a text format used to exchange e-mails. The most recent specification can be found in RFC5322. The format was originally defined in RFC 822, and people still call it "RFC822 format", hence I named this module. In the following document, I also refer to the format as "RFC822 format".

Parsing message headers

`rfc822-read-headers` *iport :key strict? reader* [Function]
{`rfc.822`} Reads RFC822 format message from an input port *iport*, until it reaches the end of the message header. The header fields are broken into a list of the following format:

```
((name body) ...)
```

Name ... are the field names, and *body* ... are the corresponding field body, both as strings. Field names are converted to lower-case characters. Field bodies are not modified, except the folded line is unfolded. The order of fields are preserved.

By default, the parser works permissively. If EOF is encountered during parsing header, it is taken as the end of the message. And if a line that doesn't consist neither continuing (folded) line nor start a new header field, it is simply ignored. You can change this behavior by giving true value to the keyword argument *strict?*; then the parser raises an error for such a malformed header.

The keyword argument *reader* takes a procedure that reads a line from *iport*. Its default is `read-line`, which should be enough for most cases.

`rfc822-header->list` *iport :key strict? reader* [Function]
{`rfc.822`} This is an old name of `rfc822-read-headers`. This is kept for the backward compatibility. The new code should use `rfc822-read-headers` instead.

`rfc822-header-ref` *header-list field-name :optional default* [Function]

{`rfc.822`} An utility procedure to get a specific field from the parsed header list, which is returned by `rfc822-read-headers`.

Field-name specifies the field name in a lowercase string. If the field with given name is in *header-list*, the procedure returns its value in a string. Otherwise, if *default* is given, it is returned, and if not, `#f` is returned.

This procedure can actually be used not only for the result of `rfc822-read-headers`, but for retrieving a value keyed by strings in a list-of-list structure: `((name value option ...) ...)`. For example, the return value of `parse-cookie-string` can be passed to `rfc-822-header-ref` (see Section 12.31 [HTTP cookie handling], page 717, for `parse-cookie-string`).

```
(rfc822-header-ref
  '(("from" "foo@example.com") ("to" "bar@example.com"))
  "from")
 ⇒ "foo@example.com"

;; If no entry matches, #f is returned by default
(rfc822-header-ref
  '(("from" "foo@example.com") ("to" "bar@example.com"))
  "reply-to")
 ⇒ #f

;; You can give the default value for no-match case
(rfc822-header-ref
  '(("from" "foo@example.com") ("to" "bar@example.com"))
  "reply-to" 'none)
 ⇒ none

;; By giving the default value, you can distinguish
;; the no-match case and there's actually an entry with value #f.
(rfc822-header-ref
  '(("from" "foo@example.com") ("reply-to" #f))
  "reply-to" 'none)
 ⇒ #f
```

Basic field parsers

Several procedures are provided to parse "structured" header fields of RFC2822 messages. These procedures deal with the body of a header field, i.e. if the header field is `"To: Wandering Schemer <schemer@example.com>"`, they parse `"Wandering Schemer <schemer@example.com>"`.

Most of procedures take an input port. Usually you first parse the entire header fields by `rfc822-read-headers`, obtain the body of the header by `rfc822-header-ref`, then open an input string port for the body and use those procedures to parse them.

The reason for this complexity is because you need different tokenization schemes depending on the type of the field. Rfc2822 also allows comments to appear between tokens for most cases, so a simple-minded regexp won't do the job, since rfc2822 comment can be nested and can't be represented by regular grammar. So, this layer of procedures are designed flexible enough to handle various syntaxes. For the standard header types, high-level parsers are also provided; see "specific field parsers" below.

`rfc822-next-token` *iport :optional tokenizer-specs* [Function]

{`rfc.822`} A basic tokenizer. First it skips whitespaces and/or comments (`CFWS`) from *iport*, if any. Then reads one token according to *tokenizer-specs*. If *iport* reaches EOF before any token is read, EOF is returned.

Tokenizer-specs is a list of tokenizer spec, which is either a char-set or a cons of a char-set and a procedure.

After skipping `CFWS`, the procedure peeks a character at the head of *iport*, and checks it against the char-sets in *tokenizer-specs* one by one. If a char-set that contains the character belongs to is found, then a token is retrieved as follows: If the tokenizer spec is just a char-set, a sequence of characters that belong to the char-set consists a token. If it is a cons, the procedure is called with *iport* to read a token.

If the head character doesn't match any char-sets, the character is taken from *iport* and returned.

The default *tokenizer-specs* is as follows:

```
(list (cons #["] rfc822-quoted-string)
      (cons *rfc822-atext-chars* rfc822-dot-atom))
```

Where `rfc822-quoted-string` and `rfc822-dot-atom` are tokenizer procedures described below, and `*rfc822-atext-chars*` is bound to a char-set of `atext` specified in rfc2822. This means `rfc822-next-token` retrieves a token either `quoted-string` or `dot-atom` specified in rfc2822 by default.

Using *tokenizer-specs*, you can customize how the header field is parsed. For example, if you want to retrieve a token that is either (1) a word constructed by alphabetic characters, or (2) a quoted string, then you can call `rfc822-next-token` by this:

```
(rfc822-next-token iport
  `(#[[:alpha:]] (#["] . ,rfc822-quoted-string)))
```

`rfc822-field->tokens` *field :optional tokenizer-specs* [Function]

{`rfc.822`} A convenience procedure. Creates an input string port for a field body *field*, and calls `rfc822-next-token` repeatedly on it until it consumes all input, then returns a list of tokens. *Tokenizer-specs* is passed to `rfc822-next-token`.

`rfc822-skip-cfws` *iport* [Function]

{`rfc.822`} A utility procedure that consumes any comments and/or whitespace characters from *iport*, and returns the head character that is neither a whitespace nor a comment. The returned character remains in *iport*.

`*rfc822-atext-chars*` [Constant]

{`rfc.822`} Bound to a char-set that is a valid constituent of `atom`.

`*rfc822-standard-tokenizers*` [Constant]

{`rfc.822`} Bound to the default *tokenizer-specs*.

`rfc822-atom` *iport* [Function]
`rfc822-dot-atom` *iport* [Function]
`rfc822-quoted-string` *iport* [Function]

{`rfc.822`} Tokenizers for `atom`, `dot-atom` and `quoted-string`, respectively. The double-quotes and escaping backslashes within `quoted-string` are removed by `rfc822-quoted-string`.

Specific field parsers

`rfc822-parse-date` *string* [Function]

{`rfc.822`} Takes RFC-822 type date string, and returns eight values:

```
year, month, day-of-month, hour, minutes, seconds, timezone,
day-of-week.
```

Timezone is an offset from UT in minutes. *Day-of-week* is a day from sunday, and may be #f if that information is not available. *Month* is an integer between 1 and 12, inclusive. If the string is not parsable, all the elements are #f.

`rfc822-date->date` *string* [Function]

{`rfc.822`} Parses RFC822 type date format and returns SRFI-19 `<date>` object (see Section 11.7.4 [SRFI-19 Date], page 602). If *string* can't be parsed, returns `#f` instead.

To construct rfc822 date string from SRFI-19 date, you can use `date->rfc822-date` below.

Message constructors

`rfc822-write-headers` *headers :key output continue check* [Function]

{`rfc.822`} This is a sort of inverse function of `rfc822-read-headers`. It receives a list of header data, in which each header data consists of `(<name> <body>)`, and writes them out in RFC822 header field format to the output port specified by the *output* keyword argument. The default output is the current output port.

By default, the procedure assumes *headers* contains all the header fields, and adds an empty line in the end of output to indicate the end of the header. You can pass a true value to the *continue* keyword argument to prevent this, enabling more headers can be added later.

I said "a sort of" above. That's because this function doesn't (and can't) do the exact inverse. Specifically, the caller is responsible for line folding and make sure each header line doesn't exceed the "hard limit" defined by RFC2822 (998 octets). This procedure cannot do the line folding on behalf of the caller, because the places where line folding is possible depend on the semantics of each header field.

It is also the caller's responsibility to make sure header field bodies don't have any characters except non-NUL US-ASCII characters. If you want to include characters outside of that range, you should convert them in the way allowed by the protocol, e.g. MIME. The `rfc.mime` module (see Section 12.39 [MIME message handling], page 730) provides a convenience procedure `mime-encode-text` for such purpose. Again, this procedure cannot do the encoding automatically, since the way the field should be encoded depends on header fields.

What this procedure can do is to check and report such violations. By default, it runs several checks and signals an error if it finds any violations of RFC2822. You can control this checking behavior by the *check* keyword argument. It can take one of the following values:

`:error` Default. Signals an error if a violation is found.

`#f, :ignore`
Doesn't perform any check. Trust the caller.

`procedure`
When `rfc822-write-headers` finds a violation, the procedure is called with three arguments; the header field name, the header field body, and the type of violation explained below. The procedure may correct the problem and return two values, the corrected header field name and body. The returned values are checked again. If the procedure returns the header field name and body unchanged, an error is signaled in the same way as `:error` is specified.

The third argument passed to the procedure given to the *check* argument is one of the following symbols. New symbols may be added in future versions for more checks.

`incomplete-string`
: Incomplete string is passed.

`bad-character`
: Header field contains characters outside of US-ASCII or NUL.

`line-too-long`
: Line length exceeds 998 octet limit.

`stray-crlf`
: The string contains CR and/or LF character that doesn't consist of proper line folding.

`date->rfc822-date` *date* [Function]

{`rfc.822`} Takes SRFI-19 `<date>` object (see Section 11.7.4 [SRFI-19 Date], page 602) and returns a string of its rfc822 date representation. This is a reverse operation of `rfc822-date->date`.

12.30 `rfc.base64` - Base64 encoding/decoding

`rfc.base64` [Module]

This module defines a few functions to encode/decode Base64 format, defined in RFC 2045 ([RFC2045], page 829), section 6.3 and RFC 4648 ([RFC4648], page 829)

`base64-encode` *:key line-width url-safe* [Function]

{`rfc.base64`} Reads byte stream from the current input port, encodes it in Base64 format and writes the result character stream to the current output port. The conversion ends when it reads EOF from the current input port.

Newline characters can be inserted to keep the maximum line width to the value given to the *line-width* keyword argument. The default value of *line-width* is 76, as specified in RFC2045. You can give `#f` or zero to *line-width* to suppress line splitting.

If a true value is given to *url-safe*, the input bytes will be encoded with an alternative encoding table, which substitutes `+` instead of `-` and `/` instead of `_`. The result will contain filename and url safe characters only. Default value of *url-safe* is false.

`base64-encode-string` *string :key line-width url-safe* [Function]

{`rfc.base64`} Converts contents of *string* to Base64 encoded format. Input string can be either complete or incomplete string; it is always interpreted as a byte sequence.

`base64-decode` *:key url-safe* [Function]

{`rfc.base64`} Reads character stream from the current input port, decodes it from Base64 format and writes the result byte stream to the current output port. The conversion ends when it reads EOF or the termination character (`=`). The characters which does not in legal Base64 encoded character set are silently ignored.

`base64-decode-string` *string :key url-safe* [Function]

{`rfc.base64`} Decodes a Base64 encoded string *string* and returns the result as a string. The conversion terminates at the end of *string* or the termination character (`=`). The characters which does not in legal Base64 encoded character set are silently ignored.

12.31 `rfc.cookie` - HTTP cookie handling

`rfc.cookie` [Module]

Defines a set of functions to parse and construct a “cookie” information defined in RFC 6265.

`parse-cookie-string` *string :optional version* [Function]

{`rfc.cookie`} Parse a cookie string *string*, which is the value of “Cookie” request header. Usually, the same information is available to CGI program via the environment variable `HTTP_COOKIE`.

If the cookie version is known, via “Cookie2” request header, the integer version must be passed to *version*. Otherwise, `parse-cookie-string` figures out the version from *string*.

The result has the following format.

```
((<name> <value> [:path <path>] [:domain <domain>] [:port <port>])
 ...)
```

where `<`*name*`>` is the attribute name, and `<`*value*`>` is the corresponding value. If the attribute doesn’t have value, `<`*value*`>` is `#f`. (Note that it differs from the attribute having null value, `""`.) If the attribute has path, domain or port options, it is given as a form of keyword-value pair.

Note: To retrieve the value of a specific cookie conveniently, you can use `rfc822-header-ref` (see Section 12.29 [RFC822 message parsing], page 712).

`construct-cookie-string` *specs :optional version* [Function]

{`rfc.cookie`} Given list of cookie specs, creates a cookie string suitable for `Set-cookie2` or `Set-cookie` header.

Optional *version* argument specifies cookie protocol version. 0 for the old Netscape style format, and 1 for RFC2965 style format. When omitted, version 1 is assumed.

Each cookie spec has the following format.

```
(<name> <value> [:comment <comment>] [:comment-url <url>]
                [:discard <bool>] [:domain <domain>]
                [:max-age <age>] [:path <path>]
                [:port <port-list>] [:secure <bool>] [:http-only <bool>]
                [:version <version>] [:expires <date>])
```

Where,

`<name>` A string. Name of the cookie.

`<value>` Value of the cookie. May be a string, or `#f` if no value is needed.

`<comment> <url> <domain> <path> <port-list>`
Strings.

`<bool>` Boolean value

`<age> <version>`
Integers

`<date>` Either an integer (seconds since Epoch) or a formatted date string following the netscape cookie specification.

The attribute values are quoted appropriately. If the specified attribute is irrelevant for the *version*, it is ignored. So you can pass the same specs to generate both old-style and new-style cookie strings.

Return value is a list of cookie strings, each of which stands for each cookie. For old-style protocol (using `Set-cookie` header) you must send each of them by individual header. For new-style protocol (using `Set-cookie2` header), you can join them with comma and send it at once. See RFC6265 for further details.

Some examples:

```
(construct-cookie-string
   `(("name" "foo" :domain "foo.com" :path "/"
                   :expires ,(+ (sys-time) 86400) :max-age 86400)))
 ⇒ ("name=foo;Domain=foo.com;Path=/;Max-age=86400")

(construct-cookie-string
   `(("name" "foo" :domain "foo.com" :path "/"
                   :expires ,(+ (sys-time) 86400) :max-age 86400))
   0)
 ⇒
 ("name=foo;Domain=foo.com;Path=/;Expires=Sun, 09-Sep-2001 01:46:40 GMT")
```

12.32 `rfc.ftp` - FTP client

`rfc.ftp` [Module]

This module provides a set of convenient functions to access ftp servers.

`<ftp-connection>` [Class]

{`rfc.ftp`} An object to keep FTP connection to a server. It has the following public slots.

`transfer-type` [Instance Variable of `<ftp-connection>`]

FTP transfer type. Must be one of the following symbols: `ascii`, `binary` (default), and `image`.

`passive` [Instance Variable of `<ftp-connection>`]

True if the client uses passive connection.c

`log-drain` [Instance Variable of `<ftp-connection>`]

This slot must hold a `<log-drain>` instance (see Section 9.16 [User-level logging], page 393) or `#f`. If it has a `<log-drain>` instance, ftp communication logs are put to it.

`<ftp-error>` [Condition Type]

{`rfc.ftp`} This type of exception is thrown when the ftp server returns an error code. Inherits `<error>`. The message field contains the server reply, including the status code.

`call-with-ftp-connection` *host proc :key passive port username password account log-drain* [Function]

{`rfc.ftp`} A high-level convenience routine to open an ftp connection to an ftp server and calls the given procedure.

The server is specified by *host*. Optionally, you can add user name and/or port number by the form *`user@servername:port`*. If present, user and port portion in *host* supersedes the keyword arguments.

If ftp connection to *host* is established successfully, *proc* is called with one argument, which is an instance of `<ftp-connection>`. When *proc* returns, the connection is closed and the return value(s) of *proc* is/are returned from `call-with-ftp-connection`. When an exception is thrown, the ftp connection is closed before the exception escapes from `call-with-ftp-connection`.

When a true value is given to the keyword argument *passive*, created ftp connection will use passive mode to send/receive data. The default is the active mode.

The keyword argument *port*, *username*, and *password* specify the port number, username, and password, respectively. When omitted, the port number defaults to 21, *username* to `"anonymous"`, and *password* to `"anonymous@"`. Note that the port number and/or username are ignored when those information is given in the *host* argument.

If the keyword argument *account* is given, its value is passed to ftp `ACCT` command when requested by the server at login time. The default value is a null string `""`.

The keyword argument *log-drain* is set to the created ftp connection's `log-drain` slot.

`ftp-transfer-type` *conn* [Function]
{`rfc.ftp`} Returns the transfer type of the ftp connection `conn`. Can be used with setter, e.g. `(set! (ftp-transfer-type conn) 'ascii)`.

`ftp-passive?` *conn* [Function]
{`rfc.ftp`} Returns true iff ftp connection uses passive data retrieval.

`ftp-login` *host :key passive port username password account log-drain* [Function]
{`rfc.ftp`} Connects to the ftp server specified by *host*, authenticate the user, and returns a newly created `<ftp-connection>` instance. This procedure is called implicitly when you use `call-with-ftp-connection`. The semantics of the *host* argument and the keyword arguments are the same as `call-with-ftp-connection`.

`ftp-quit` *conn* [Function]
{`rfc.ftp`} Sends ftp `QUIT` command to the connection *conn* and shutdown the connection. This procedure is called implicitly when you use `call-with-ftp-connection`.

Once a connection is shut down, you cannot communicate through this connection.

`ftp-chdir` *conn dirname* [Function]
{`rfc.ftp`} Changes the remote directory to *dirname*.

`ftp-remove` *conn path* [Function]
{`rfc.ftp`} Removes the remote file named by *path*.

`ftp-help` *conn :optional option ...* [Function]
{`rfc.ftp`} Sends ftp `HELP` commands. *Option*s must be strings, and will be passed to the `HELP` command arguments.

`ftp-mkdir` *conn dirname* [Function]
{`rfc.ftp`} Creates a directory *dirname*. Returns the created directory name.

`ftp-current-directory` *conn* [Function]
{`rfc.ftp`} Returns the current remote directory.

`ftp-site` *conn arg* [Function]
{`rfc.ftp`} Sends ftp `SITE` command with the argument *arg*. The `SITE` command's semantics depends on the server. Returns the server reply.

`ftp-rmdir` *conn dirname* [Function]
{`rfc.ftp`} Removes remote directory specified by *dirname*. Returns the server reply.

`ftp-stat` *conn :optional pathname* [Function]
{`rfc.ftp`} Sends ftp `STAT` command to the server. RFC959 defines several different semantics of this command. See RFC959 for the details. Returns the server reply.

`ftp-system` *conn* [Function]

{`rfc.ftp`} Queries the server's operating system by ftp `SYST` command. Returns the server reply without status code.

```
(call-with-ftp-connection "localhost" ftp-system)
  ⇒ "UNIX Type: L8"
```

`ftp-size` *conn path* [Function]

{`rfc.ftp`} Queries the size of the remote file specified by *path*. Returns the integer value.

Note: The size may differ whether the connection is in ascii mode or binary mode; furthermore, some ftp server may returns the value only if the connection is in binary mode. Make sure you have desired transfer type in the connection.

`ftp-mdtm` *conn path* [Function]

{`rfc.ftp`} Queries the modification time of the remote file specified by *path*. This function returns the server's reply as is, including the status code. Use `ftp-mtime` below to obtain a parsed result.

`ftp-mtime` *conn path :optional local-time?* [Function]

{`rfc.ftp`} Queries the modification time of the remote file specified by *path*, and returns the result in a `<date>` object (see Section 11.7 [Time data types and procedures], page 600). If a true value is given to `local-time?`, the returned date is in local time. Otherwise, the returned date is in UTC.

`ftp-noop` *conn* [Function]

{`rfc.ftp`} Sends ftp `NOOP` command and returns the server's reply.

`ftp-list` *conn :optional path* [Function]

{`rfc.ftp`} Returns the information about the files within the remote file or directory specified by *path*, or the current remote directory, much like `ls(1)` format. Returns a list of strings, where each string is for each line of the server's reply. The exact format depends on the server.

`ftp-name-list` *conn :optional path* [Function]
`ftp-ls` *conn :optional path* [Function]

{`rfc.ftp`} Return the list of names in the specified *path*, or the current remote directory, without any other information. `ftp-ls` is just an alias of `ftp-name-list` for the convenience.

Note that the server may return an error if there's no files in the remote directory.

`ftp-get` *conn path :key sink flusher* [Function]

{`rfc.ftp`} Retrieves a remote file *path*. The retrieved data is sent to an output port given to *sink*. Once all the data is retrieved, a procedure given to *flusher* is called with the port *sink* as an argument, and its return value(s) is/are returned from `ftp-get`.

The default values of *sink* and *flusher* are a newly created string port and `get-output-string`, respectively. That is, `ftp-get` returns the retrieved data as a string by default. You don't want this behavior if the retrieved file is huge.

`ftp-put` *conn from-file :optional to-file* [Function]

{`rfc.ftp`} Sends the local file specified by *from-file* to the remote server as the name specified by *to-file*. If *to-file* is omitted, the basename of *from-file* is used. Returns the server response.

`ftp-put-unique` *conn from-file* [Function]

{`rfc.ftp`} Sends the local file specified by *from-file* to the remote server. The remote side filename is guaranteed to be unique. Returns two values—the final server response, and the remote file name. The second value can be `#f` if the remote host doesn't support RFC1123 (which must be rare).

`ftp-rename` *conn from-name to-name* [Function]

{`rfc.ftp`} Renames the remote file specified by *from-name* to the name *to-name*. Returns the final response of the server.

12.33 `rfc.hmac` - HMAC keyed-hashing

`rfc.hmac` [Module]

This module implements HMAC algorithm, Keyed-hashing for message authentication, defined in RFC 2104.

For simple batched keyed hashing, you can use high-level API `hmac-digest` and `hmac-digest-string`. Or you can create `<hmac>` object and update its state as the data coming in.

`<hmac>` [Class]

{`rfc.hmac`} Keeps state information of HMAC algorithm. Key and the hashing algorithm should be given at the construction time, using `:key` and `:hasher` keyword-arguments respectively. You can pass any class object that implements message digest interface (see Section 12.63 [Message digester framework], page 795), such as `<md5>` (see Section 12.38 [MD5 message digest], page 730) or `<sha256>` (see Section 12.41 [SHA message digest], page 736).

Example:

```
(make <hmac> :key (make-byte-string 16 #x0b) :hasher <md5>)
```

`hmac-update!` (*hmac* <*hmac*>) *data* [Method]

{`rfc.hmac`} Updates the internal state of *hmac* by *data*, which must be represented by a (possibly incomplete) string.

`hmac-final!` (*hmac* <*hmac*>) [Method]

{`rfc.hmac`} Finalizes the internal state of *hmac* and returns the hashed string in incomplete string. You can use `digest-hexify` (see Section 12.63 [Message digester framework], page 795) to obtain "hexified" result. Once finalized, you can't call `hmac-update!` or `hmac-final!` on *hmac*.

`hmac-digest` *:key key hasher* [Method]

{`rfc.hmac`} Creates an `<hmac>` object and hash the data stream from the current input port, then returns the hashed result in an incomplete string.

`hmac-digest-string` *string :key key hasher* [Method]

{`rfc.hmac`} Creates an `<hmac>` object and hash the data in *string*, then returns the hashed result in an incomplete string.

12.34 `rfc.http` - HTTP

`rfc.http` [Module]

This module provides a simple client API for HTTP/1.1, defined in RFC2616, "Hypertext Transfer Protocol – HTTP/1.1" ([RFC2616], page 829).

Current API implements only a part of the protocol. It doesn't talk with HTTP/1.0 server yet, and it doesn't support HTTP/1.1 advanced features such as persistent connection. Support for those features may be added in the future versions.

`<http-error>` [Condition Type]

{`rfc.http`} This type of condition is raised when the server terminates connection prematurely or server's response has invalid header fields. Inherits `<error>`.

`http-get` *server request-uri :key sink flusher redirect-handler secure . . .* [Function]
`http-head` *server request-uri :key redirect-handler secure . . .* [Function]
`http-post` *server request-uri body :key sink flusher redirect-handler secure . . .* [Function]
`http-put` *server request-uri body :key sink flusher redirect-handler secure . . .* [Function]
`http-delete` *server request-uri :key sink flusher redirect-handler secure . . .* [Function]

{`rfc.http`} Send http GET, HEAD, POST, PUT and DELETE requests to the http *server*, respectively, and returns the server's reply.

By default, if the server returns 300, 301, 302, 303, 305 and 307 status, these procedures attempts to fetch the redirected URL by the "location" reply message header if it is allowed by RFC2616. This behavior can be turned off or customized by the *redirect-handler* keyword argument; see the "keyword arguments" heading below for the details.

Required arguments: The *server* argument specifies http server name in a string. A server name can be optionally followed by colon and a port number. You can use IP address, too; for IPv6, you have to surround the address in brackets.

Additionally, you can specify `"unix:/path"` where `/path` is the absolute path to the unix domain socket; this allows to connect to httpd listening on unix domain sockets. Examples: `"w3c.org"`, `"mycompany.com:8080"`, `"192.168.0.1:8000"`, `"[::1]:8000"`

The *request-uri* argument can be a string or a list. If it is a string, it's *request-uri* specified in RFC2616; usually, this is the path part of http url. The string is passed to the server as is, so the caller must properly convert character encodings and perform necessary url encodings.

If *request-uri* is a list, it must be in the following form:

```
(path (name value) ...)
```

Here, *path* is a string specifying up to the path component of the request-uri. From provided alist of *name*s and *value*s, http procedures compose a query string in `application/x-www-form-urlencoded` format as defined in HTML4, and append it to *path*. For example, the following two requests have the same effect. Note that url escaping is automatically handled in the second call.

```
(http-get "example.com" "/search?q=foo%20bar&n=20")

(http-get "example.com" '("/search" (q "foo bar") (n 20)))
```

If *request-encoding* keyword argument is also given, *name*s and *value*s are converted into the specified character encoding before url escaping. If it is omitted, gauche's internal character encoding is used.

Some procedures take the third argument, *body*, to specify the body of the request message. It can be a string, which will be copied verbatim to the request body, or a list, which will be encoded in `multipart/form-data` message.

If *body* is a list, it is a list of parameter specs. Each parameter spec is either a list of name and value, e.g. `("submit" "OK")` or a name followed by keyword-value list, e.g. `("upload" :file "logo.png" :content-type "image/png")`.

The first form is for the convenience. It is also compatible to the query parameter list in *request-uri*, so that you can use the same format for GET and POST request. Each value is put in a MIME part with `text/plain` media type, with the character encoding specified by `request-encoding` keyword argument described below.

The second form allows further control over each MIME part's attributes. The following keywords are treated specially.

`:value` Specifies the value of the parameter. The convenience form, `(name val)`, is just an abbreviation of `(name :value val)`.

`:file` Specifies the pathname of the file, whose content is inserted as the value of the parameter. Useful to upload a file. This option has precedence over `:value`. MIME type of the part is set to `application/octet-stream` unless specified otherwise.

`:content-type`
Overrides the MIME type of the part. A charset parameter is added to the content-type if not given in this argument.

`:content-transfer-encoding`
Specifies the value of content-transfer-encoding; currently the following values are supported: `7bit`, `binary`, `quoted-printable` and `base64`. If omitted, `binary` is used.

Other keywords are used as the header of the MIME part.

Return values: All procedures return three values.

The first value is the status code defined in RFC2616 in a string (such as "200" for success, "404" for "not found").

The second value is a list of parsed headers—each element of list is a list of `(header-name value ...)`, where *header-name* is a string name of the header (such as "content-type" or "location"), and *value* is the corresponding value in a string. The header name is converted to lowercase letters. The value is untouched except that "soft line breaks" are removed, as defined in RFC2822. If the server returns more than one headers with the same name, their values are consolidated to one list. Except that, the order of the header list in the second return value is the same as the order in the server's reply.

The third value is for the message body of the server's reply. By default, it is a message body itself in a string. If the server's reply doesn't have a body, the third value is `#f`. You can change how the message body is handled by keyword arguments; for example, you can directly store the returned message body to a file without creating intermediate string. The details are explained below.

Keyword arguments: By default, these procedures only attaches `"Host"` header field to the request message. You can give keyword arguments to add more header fields.

```
(http-get "foo.bar.com" "/index.html"
  :accept-language "ja"
  :user-agent "My Scheme Program/1.0")
```

The following keyword arguments are recognized by the procedure and do not appear in the request headers.

`request-encoding`
When a list is given to the *request-uri* or *body* arguments, the characters in names and values of the parameters are first converted to the character encoding specified by this keyword argument, then encoded into `application/x-www-form-urlencoded` or `multipart/form-data` MIME formats. If this argument is omitted, Gauche's internal character encoding is used.

For `multipart/form-data`, you can override character encodings for individual parameters by giving `content-type` header. See the description of *body* arguments above.

If you give a string to *request-uri* or *body*, it is used without encoding conversion. It is caller's responsibility to ensure desired character encodings are used.

`proxy` Specify http proxy server in a string of a form `hostname` or `hostname:port`. If omitted, the value of the parameter `http-proxy` is used.

`redirect-handler`
Specifies how the redirection is handled when the server responds with 3xx status code. You can pass `#f`, `#t` or a procedure. The default is `#t`.

If `#f` is given, no redirect attempt will be made; the 3xx status code and response is just returned from `http-*` procedures as they are.

If a procedure is given, it is called when the response status code is 3xx. The procedure takes four arguments, the request method (in symbol, e.g. `GET`), the response status code (in string, e.g. `"302"`), the parsed response headers and the response body (a string if there's a body, or `#f` if the response doesn't have a body).

The procedure can return a pair or `#f`. If it is a pair, it should be `(method . url)`, where *method* is a symbol (e.g. `GET`) and *url* is a string representing url. If a pair is returned, the `http-*` procedures tries to send the request with the given method (it allows a redirection of POST request to be GET, for example). If it is `#f`, no further attempt of redirection is made.

If *redirect-handler* is `#t`, which is the default, then it works as if the value of the parameter `http-default-redirect-handler` is passed to *redirect-handler*. The parameter contains a procedure with reasonable default behavior. See the `http-default-redirect-handler` entry below for the details.

A loop in redirection is detected automatically and `<http-error>` is thrown.

`no-redirect`
This is an obsoleted keyword argument kept only for the backward compatibility. If a true value is given, it has the same effect as specifying `#f` to *redirect-handler*.

`secure` If a true value is given, the secure connection is used. The value specifies the secure transport agent to establish https connection. It can be `#t` or a symbol `tls` or `stunnel`. If `#f` is given (default), non-secure plain http is used. See the "Secure connection" section below.

`auth-user, auth-password`
If given, the authorization header using Basic Authentication (RFC2617) is added to the request. In future, we might add support for other authentication scheme.

`sink, flusher`
You can customize how the reply message body is handled by these keyword arguments. You have to pass an output port to *sink*, and a procedure that takes two arguments to *flusher*.

When the procedure starts receiving the message body, it feeds the received chunk to *sink*. When the procedure receives entire message body, *flusher* method is called with *sink* and a list of message header fields (in the same format to be returned in the second value from the procedure). The return value of *flusher* becomes the third return value from the procedure.

So, the default value of *sink* is a newly opened string port and the default value of *flusher* is `(lambda (sink headers) (get-output-string sink))`.

The following example saves the message body directly to a file, without allocating (potentially very big) string buffer.

```
(call-with-output-file "page.html"
  (lambda (out)
    (http-get "www.schemers.org" "/"
       :sink out :flusher (lambda _ #t))))
```

The module also provides some utility procedures.

`http-user-agent` *:optional value* [Parameter]

{`rfc.http`} The value of this parameter is used as a default value to pass to the user-agent header. The default value is something like `gauche.http/*`, where `*` is Gauche's version. An application is encouraged to set this parameter appropriately.

`http-proxy` *:optional value* [Parameter]

{`rfc.http`} This value is used as the default http proxy name by `http-get` etc. The default value is `#f` (no proxy).

`http-default-redirect-handler` *:optional value* [Parameter]

{`rfc.http`} Specifies the behavior of redirection if no `redirect-handler` keyword argument is given to the `http-*` procedures. If you change this value, it must be a procedure that follows the protocol of `redirect-handler`; see the description of `http-*` procedures above.

The default behavior is as follows:

`300`, `301`, `305`, `307`
: Redirect to the url given to the `location` header only if the original request method is `GET` or `HEAD`.

`302`
: Redirect to the url given to the `location` header. If the original request method is `HEAD`, it is used again. Otherwise, `GET` method is used.

  Strictly speaking, this is a violation of RFC2616. However, as the note in RFC2616 says, many user agent do this, so we follow the flock. (We may change this in future.)

`303`
: Redirect to the url given to the `location` header. If the original request method is `HEAD`, it is used again. Otherwise, `GET` method is used.

other than above
: No redirection is made.

The following code is an example of intercepting the default behavior in a specific request:

```
(http-get server uri
  :redirect-handler
  (^[method status headers body]
    (if (and (equal? status "302")
             (not (member method '(GET HEAD))))
        #f
        ((http-default-request-handler) method status headers body))))
```

`http-compose-query` *path params :optional encoding* [Function]

{`rfc.http`} A helper procedure to create a request-uri from a list of query parameters. *Encoding* specifies the character encodings to be used.

```
(http-compose-query "/search" '((q "$foo") (n 20)))
 ⇒ "/search?q=%24foo&n=20"

(http-compose-query "" '((x "a b") (x 2)))
 ⇒ "?x=a%20b&x=2"
```

If *path* is `#f`, only the query parameter part is returned (compare the following example and the last example):

```
(http-compose-query #f '((x "a b") (x 2)))
 ⇒ "x=a%20b&x=2"
```

`http-compose-form-data` *params port :optional encoding* [Function]

{`rfc.http`} A helper procedure to create `multipart/form-data` from a list of parameters. The format of *params* argument is the same as the list format of *body* argument of http request procedures. The result is written to an output port *port*, and the boundary string used to compose MIME message is returned. Alternatively you can pass `#f` to the *port* to get the result in a string. In that case, two values are returned, the MIME message string and the boundary string.

Encoding specifies the character encodings to be used. When omitted, Gauche's native encoding is used.

```
(define p (open-output-string))

(http-compose-form-data '((name "Preludes and Fugues")
                          (composer "Shostakovich, Dmitri")
                          (opus "87"))
                        p)
  ⇒ "boundary-fh87o52rp6zkubp2uhdmo"

(get-output-string p)
  ⇒
  "\r\n--boundary-fh87o52rp6zkubp2uhdmo\r\nContent-type: te
   xt/plain; charset=utf-8\r\nContent-transfer-encoding: bi
   nary\r\ncontent-disposition: form-data; name=title\r\n\r\n
   Preludes and Fugues\r\n--boundary-fh87o52rp6zkubp2uhdmo...
;; (result is truncated)
```

`http-status-code->description` *code* [Function]

{`rfc.http`} Returns a brief description of http status code `code`, which may be an integer or a string (e.g. `"404"`). If `code` isn't one of known code, `#f` is returned.

```
(http-status-code->description 404)
  ⇒ "Not Found"
```

Secure connection

When you pass a true value to `secure` keyword argument, the request-making APIs such as `http-get` use a secure connection. That is, it connects with `https` instead of `http`. The actual value for the keyword argument can be one of the followings:

`#t`
`tls` The `rfc.tls` module is used for the secure connection. See Section 12.42 [Transport layer security], page 737, for the details—you might need to set CA certificate bundle path.

`stunnel` The external process `stunnel` is spawned and used for the secure connection.

`#f` Secure connection is not used.

If specified secure connection subsystem isn't available in the running Gauche, an error is signaled. Use the following procedure to check if you can use secure connections:

`http-secure-connection-available?` *:optional type* [Function]

{`rfc.http`} The *type* argument may be `tls` or `stunnel`. If omitted, `tls` is assumed. Returns `#t` if running Gauche can use secure connection of the given type, `#f` otherwise.

12.35 `rfc.icmp` - ICMP packets

`rfc.icmp` [Module]

{`rfc.icmp`} This module provides some basic utilities to construct and parse ICMP packets.

For the functions below, *buffer* should be a writable u8vector of the enough size.

Parsing functions takes *offset* as well as *buffer*, which specifies the beginning of the ICMP packet. Using the offset you can carry the whole IP packet in *buffer*, without creating a new buffer to extract ICMP portion.

`icmp4-fill-echo!` *buffer ident sequence data* [Function]

{`rfc.icmp`} Fills *buffer* with the ICMPv4 Echo Request packet. *Data* must be a u8vector. The checksum field is left to be zero, which can be filled by `icmp4-fill-checksum!`.

`icmp4-fill-checksum!` *buffer size* [Function]

{`rfc.icmp`} Calculates the ICMPv4 checksum of the packet in the *buffer*, of *size* length (the size of the packet, not the buffer), and fills the checksum field of the packet.

`icmp6-fill-echo!` *buffer ident sequence data* [Function]

{`rfc.icmp`} Fills *buffer* with the ICMPv6 Echo Request packet. *Data* must be a u8vector. The checksum field is left to be zero, which is to be filled by the kernel (so you don't need to fill by yourself).

`icmp-packet-type` *buffer offset* [Function]
`icmp-packet-code` *buffer offset* [Function]
`icmp-packet-ident` *buffer offset* [Function]
`icmp-packet-sequence` *buffer offset* [Function]

{`rfc.icmp`} Extracts type, code, ident and sequence fields of ICMP packet. These functions are common to both ICMPv4/v6.

`icmp4-describe-packet` *buffer offset* [Function]
`icmp6-describe-packet` *buffer offset* [Function]

{`rfc.icmp`} Prints out a simple text description of the given ICMPv4 and v6 packet, respectively.

`icmp4-message-type->string` *type* [Function]
`icmp4-unreach-code->string` *code* [Function]
`icmp4-redirect-code->string` *code* [Function]
`icmp4-router-code->string` *code* [Function]
`icmp4-exceeded-code->string` *code* [Function]
`icmp4-parameter-code->string` *code* [Function]
`icmp4-security-code->string` *code* [Function]
`icmp6-message-type->string` *type* [Function]
`icmp6-unreach-code->string` *code* [Function]
`icmp6-exceeded-code->string` *code* [Function]
`icmp6-parameter-code->string` *code* [Function]

{`rfc.icmp`} Returns a text description of ICMPv4 and ICMPv6 types and codes.

12.36 `rfc.ip` - IP packets

`rfc.ip` [Module]

This module provides some basic utilities to parse raw IP packets.

The *packet* argument in the following functions must be any type of uniform vector (see Section 9.36 [Uniform vectors], page 476), containing a raw IP packet including its IP header. Those functions work for both IPv4 and IPv6 packets; however, reading from a raw IPv6 socket returns a packet without IPv6 header, so you usually don't need to use these functions.

The *offset* argument specifies the beginning of the IP packet in *packet*. If *packet* contains only one IP packet you can pass 0. It is not an optional argument, since these routines may be used in speed-sensitive inner loop.

`ip-version` *packet offset* [Function]
{`rfc.ip`} Returns the IP version number (either 4 or 6) of the given IP packet.

`ip-header-length` *packet offset* [Function]
{`rfc.ip`} Returns the size of IP header of the given packet in octets, including any IP header options.

`ip-protocol` *packet offset* [Function]
{`rfc.ip`} Returns the IP protocol number of the given packet.

`ip-source-address` *packet offset* [Function]
`ip-destination-address` *packet offset* [Function]
{`rfc.ip`} Returns the source and destination address in the given packet in an integer, respectively.

12.37 `rfc.json` - JSON parsing and construction

`rfc.json` [Module]
Procedures to parse JSON (RFC7159) data to S-expressions, and convert S-expressions to JSON representation, are provided.

`<json-parse-error>` [Condition type]
{`rfc.json`} The parser `parse-json` and `parse-json-string` raise this condition when they encounter invalid JSON syntax. It inherits `<error>`, and adds the following slot.

`position` [Instance Variable of `<json-parse-error>`]
The input position, counted in characters, where the error occurred.

`parse-json` *:optional input-port* [Function]
{`rfc.json`} Reads and parses the JSON representation from *input-port* (default is the current input port), and returns the result in an S-expression. May raise a `<json-parse-error>` condition when parse error occurs.

The following table shows how JSON datatypes are mapped to Scheme objects.

`true`, `false`, `null`
Symbols `true`, `false` and `null`. (Customizable by `json-special-handler`)

Arrays
Scheme vectors. (Customizable by `json-array-handler`)

Objects
Scheme assoc-lists, in which keys are strings, and values are Scheme objects. (Customizable by `json-object-handler`)

Numbers
Scheme inexact real numbers.

Strings
Scheme strings.

Since the parser used internally in `parse-json` prefetches characters, some characters after the parsed JSON expression may already been read from *port* when `parse-json` returns. That is, you cannot call `parse-json` repeatedly on *port* to read subsequent JSON expressions. Use `parse-json*` if you need to read multiple JSON expressions.

`parse-json*` *:optional input-port* [Function]

{`rfc.json`} Read JSON repeatedly from *input-port* until it reaches EOF, and returns parsed results as a list.

`parse-json-string` *str* [Function]

{`rfc.json`} Parses the JSON string and returns the result in an S-expression. May raise a `<json-parse-error>` condition when parse error occurs.

See `parse-json` above for the mappings from JSON datatypes to Scheme types.

`json-array-handler` [Parameter]
`json-object-handler` [Parameter]
`json-special-handler` [Parameter]

{`rfc.json`} The value of these parameters must be a procedure that takes one argument: for `json-array-handler`, it is a list of elements of a JSON array, for `json-object-handler`, it is a list of conses of key and value of a JSON object, and for `json-special-handler`, it is one of the symbols `false`, `true` or `null`.

Whenever `parse-json` reads a JSON array, a JSON object, or one of those special values, it calls corresponding parameter to get a Scheme object.

The default value of these parameters are `list->vector`, `identity`, and `identity`, respectively.

The following example maps JSON objects to hash tables.

```
(use gauche.parameter)
(parameterize ([json-object-handler (cut alist->hash-table <> 'string=?)])
  (parse-json-string "{\"a\":1, \"b\":2}"))
 ⇒ #<hash-table ...>
```

`<json-construct-error>` [Condition type]

{`rfc.json`} The converters `construct-json` and `construct-json-string` raise this condition when they cannot convert given Scheme object to JSON. It inherits `<error>`, and adds the following slot.

`object` [Instance Variable of `<json-construct-error>`]

The Scheme object that cannot convert to JSON representation.

`construct-json` *obj :optional output-port* [Function]
`construct-json-string` *obj* [Function]

{`rfc.json`} Creates JSON representation of Scheme object *obj*. `construct-json` writes out the result to *output-port*, whose default is the current output port. `construct-json-string` returns the result in a string. Note that RFC4627 defines JSON text to be an object or an array; so *obj* must be a Scheme object that can be mapped to either a JSON object or a JSON array.

If *obj* contains a Scheme object that cannot be mapped to JSON representation, a `<json-construct-error>` condition is raised.

Scheme objects are mapped to JSON as follows:

symbol `false`, `#f`
: `false`

symbol `true`, `#t`
: `true`

symbol `null`
: `null`

list, instance of `<dictionary>`
JSON object (list must be an assoc list of key and value).

string — string

real number
number

instance of `<sequence>` (except strings and lists)
JSON array

12.38 rfc.md5 - MD5 message digest

`rfc.md5` [Module]

This module implements MD5 message digest algorithm, defined in RFC 1321 ([RFC1321], page 829). The module extends util.digest (see Section 12.63 [Message digester framework], page 795).

`<md5>` [Class]

{`rfc.md5`} The instance of this class keeps internal state of MD5 digest algorithm.

This class implements `util.digest` framework interface, `digest-update!`, `digest-final!`, `digest`, and `digest-string`. See Section 12.63 [Message digester framework], page 795, for detailed explanation of these methods.

Besides the digester framework, this module provides to short-cut procedures.

`md5-digest` [Function]

{`rfc.md5`} Reads data from the current input port until EOF, and returns its digest in an incomplete string.

`md5-digest-string` *string* [Function]

{`rfc.md5`} Digest the data in *string*, and returns the result in an incomplete string.

12.39 rfc.mime - MIME message handling

`rfc.mime` [Module]

This module provides utility procedures to handle Multipurpose Internet Mail Extensions (MIME) messages, defined in RFC2045 thorough RFC2049. Provided APIs include procedures to parse or compose MIME-specific header fields, and parse or compose MIME-encoded message bodies.

This module mainly focuses on providing low-level building-block procedures, on top of which application-specific modules are to be built. For example, `rfc.http` uses this module to compose `multipart/form-data` message for the body of POST requests (see Section 12.34 [HTTP], page 721).

This module is supposed to be used with `rfc.822` module (see Section 12.29 [RFC822 message parsing], page 712).

Utilities for header fields

A few utility procedures to parse and generate MIME-specific header fields.

`mime-parse-version` *field* [Function]

{`rfc.mime`} If *field* is a valid header field for MIME-Version, returns its major and minor versions in a list. Otherwise, returns `#f`. It is allowed to pass `#f` to *field*, so that you can directly pass the result of `rfc822-header-ref` to it. Given parsed header list by `rfc822-read-headers`, you can get mime version (currently, it should be `(1 0)`) by the following code.

```
(mime-parse-version (rfc822-header-ref headers "mime-version"))
```

Note: simple regexp such as `#/\d+\.\d+/` doesn't do this job, for *field* may contain comments between tokens.

`mime-parse-content-type` *field* [Function]

{`rfc.mime`} Parses the "content-type" header field, and returns a list such as:

```
(type subtype (attribute . value) ...)
```

where *type* and *subtype* are MIME media type and subtype in a string, respectively

```
(mime-parse-content-type "text/html; charset=iso-2022-jp")
 ⇒ ("text" "html" ("charset" . "iso-2022-jp"))
```

If *field* is not a valid content-type field, `#f` is returned.

`mime-parse-content-disposition` *field* [Function]

{`rfc.mime`} Parses Content-disposition header field as specified in RFC2183. (mime-parse-content-disposition "attachment; filename=genome.jpeg;\ modification-date=\"Wed, 12 Feb 1997 16:29:51 -0500\";") ⇒ ("attachment" ("filename" . "genome.jpeg") ("modification-date" . "Wed, 12 Feb 1997 16:29:51 -0500"))

`mime-parse-parameters` *:optional iport* [Function]
`mime-compose-parameters` *params :optional oport :key start-column* [Function]

{`rfc.mime`} These are low-level utility procedures to parse and compose *parameter* part of header fields (as appeared in RFC2045 Section 5.1 etc).

`Mime-parse-parameters` reads the parameter part of the header body from an input port *iport*, and returns an assoc list of the parameter names and values. Conversely, `mime-compose-parameters` takes an assoc list of names and values, compose parameter part and emit it to *oport*. When omitted, the current input port and the current output port are used for *iport* and *oport*, respectively. You can pass `#f` to *oport* and `mime-compose-parameters` returns the result in a string instead of emitting it to a port.

```
(call-with-input-string
   "; name=foo; filename=\"foo/bar/baz\""
   mime-parse-parameters)
 ⇒ (("name" . "foo") ("filename" . "foo/bar/baz"))

(mime-compose-parameters
 '(("name" . "foo") ("filename" . "foo/bar/baz"))
 #f)
 ⇒ "; name=foo; filename=\"foo/bar/baz\""
```

`Mime-compose-parameters` tries to insert folding line breaks between parameters to avoid the header line becomes too long. You can pass the beginning column position of the parameter part via *start-column* argument.

We plan to make these procedures handle RFC2231's parameter value extension transparently in future.

`mime-decode-word` *word* [Function]

{`rfc.mime`} Decodes RFC2047-encoded word. If *word* isn't an encoded word, it is returned as is.

Note that this procedure decodes only if the entire *word* is an "encoded word" defined in RFC2047. If you are dealing with a field that may contain multiple encoded word and/or unencoded parts, use `mime-decode-text` below.

```
(mime-decode-word "=?iso-8859-1?q?this=20is=20some=20text?=")
 ⇒ "this is some text"
```

mime-decode-text *text* [Function]

{rfc.mime} Returns a string in which all encoded words contained within *text* are decoded. This procedure can deal with a header field body that may contain mixture of non-encoded and encoded parts, and/or multiple encoded parts. One of such header field is the Subject field of email.

```
(mime-decode-text "This is =?US-ASCII?q?some=20text?=")
  ⇒ "This is some text"
```

Care should be taken if you apply this procedure to a "structured" header field body (see RFC2822 section 2.2.2). The proper way of parsing a structured header field body is to tokenize it first, then to decode each word using `mime-decode-word`. since the decoded text may contain characters that affects the tokenization. (However, if you can just show the header field in human readable way for informational purposes, you may just use `mime-decode-text` on entire header field for the convenience).

mime-encode-word *word :key charset transfer-encoding* [Function]

{rfc.mime} Encodes *word* in the RFC2047 format. The keyword argument *charset* specifies the character encoding scheme in string or symbol. whose default is `utf-8`. If *charset* differs from Gauche's internal encoding and *word* is a complete string, the procedure converts the character encoding to *charset*, then performs transfer encoding.

```
(mime-encode-word "this is some text")
  ⇒ "=?utf-8?B?dGhpcyBpcyBzb21lIHRleHQ=?="
```

The keyword argument *transfer-encoding* specifies how the octets are encoded to transfer-safe characters. You can give a symbol `b`, `B` or `base64` for Base64, and `Q`, `q`, `quoted-printable` for Quoted-printable transfer encodings. An error is raised if you pass values other than those. The default is Base64 encoding.

This procedure does not consider the length of the resulting encoded word, which RFC2047 recommends to be less than 75 octets. Use `mime-encode-text` below to conform the line length limit.

(Note: In most Gauche procedures, a keyword argument `encoding` is used to specify character encodings. In this context we have two encodings, however, and to avoid the confusion we chose to use the terms "charset" and "transfer-encoding" that appear in RFC documents.)

mime-encode-text *text :key charset transfer-encoding line-width start-column force* [Function]

{rfc.mime} Encode *text* in RFC2047 format if necessary, and considering line folding if the result gets too long.

The keyword arguments *charset* and *transfer-encoding* are the same as `mime-encode-word`.

If the *text* only consists of printable ASCII characters, no encoding is done, and only line folding is considered. However, if a true value is given to the *force* argument, even ASCII-only *text* is encoded.

The *line-width* specifies the maximum line width of the result. Its default is 76. If the encoded word gets too long, it is splitted to multiple encoded words and CR LF SPC sequence ("folding white space" defined in RFC2822) are inserted inbetween. You can suppress this behavior by passing `#f` or `0` to *line-width*. Since encoded word needs some overhead characters, it doesn't make much sense to specify small value to `line-width`. Current implementation rejects `line-width` smaller than 30.

The *start-column* keyword argument can be used to shorten the first of folded lines to make room for header field name. For example, if you want to encode the body of a Subject header field, you can pass the value of `(string-length "Subject: ")` so that the encoded result can directly concatenated after the header field name. The default value is 0.

This procedure is not designed to encode parts of structured header fields, which have further restrictions such as which parts can be encoded and where the folding white spaces can be inserted. The robust way is to encode some parts first, then construct a structured header fields, considering line folding.

Streaming parser

The streaming parser is designed so that you can decide how to do with the message body before the entire message is read.

`mime-parse-message` *port headers handler* [Function]

{`rfc.mime`} The fundamental streaming parser. *Port* is an input port from where the message is read. *Headers* is a list of headers parsed by `rfc822-read-headers`; that is, this procedure is supposed to be called after the header part of the message is parsed from *port*:

```
(let* ((headers (rfc822-read-headers port)))
  (if (mime-parse-version (rfc822-header-ref headers "mime-version"))
     ;; parse MIME message
     (mime-parse-message port headers handler)
     ;; retrieve a non-MIME body
     ...))
```

`Mime-parse-message` analyzes *headers*, and calls *handler* on each message body with two arguments:

```
(handler part-info xport)
```

Part-Info is a `<mime-part>` structure described below that encapsulates the information of this part of the message. *Xport* is an input port, initially points to the beginning of the body of message. The handler can read from the port as if it is reading from the original *port*. However, *xport* recognizes MIME boundary internally, and returns EOF when it reaches the end of the part. (Do not read from the original *port* directly, or it will mess up the internal state of *vport*).

Handler can read the part into the memory, or save it to the disk, or even discard the part. Whatever it does, it has to read from *vport* until it returns EOF.

The return value of *handler* will be set in the `content` slot of *part-info*. If the message has nested multipart messages, *handler* is called for each "leaf" part, in depth-first order. *Handler* can know its nesting level by examining *part-info* structure. The message doesn't need to be a multipart type; if it is a MIME `message` type, *handler* is called on the body of enclosed message. If it is other media types such as `text` or `application`, *handler* is called on the (only) message body.

`<mime-part>` [Class]

{`rfc.mime`} A structure that encloses metainformation about a MIME part. It is constructed when the header of the part is read, and passed to the handler that reads the body of the part.

It has the following slots:

`type` [Instance Variable of `<mime-part>`]

MIME media type string. If `content-type` header is omitted to the part, an appropriate default value is set.

`subtype` [Instance Variable of `<mime-part>`]

MIME media subtype string. If `content-type` header is omitted to the part, an appropriate default value is set.

`parameters` [Instance Variable of `<mime-part>`]
: Associative list of parameters given to `content-type` header field.

`transfer-encoding` [Instance Variable of `<mime-part>`]
: The value of `content-transfer-encoding` header field. If the header field is omitted, an appropriate default value is set.

`headers` [Instance Variable of `<mime-part>`]
: The list of header fields, as parsed by `rfc822-read-headers`.

`parent` [Instance Variable of `<mime-part>`]
: If this is a part of multipart message or encapsulated message, points to the enclosing part's `<mime-part>` structure. Otherwise `#f`.

`index` [Instance Variable of `<mime-part>`]
: Sequence number of this part within the same parent.

`content` [Instance Variable of `<mime-part>`]
: If this part is multipart/* or message/* media type, this slot contains a list of parts within it. Otherwise, the return value of *handler* is stored.

`source` [Instance Variable of `<mime-part>`]
: This slot is only used when composing a MIME message. The caller can set this slot a name of the file to be inserted into this part, instead of setting the entire content of the file to the `content` slot. See `mime-compose-message` below for the more details.

`mime-retrieve-body` *part-info xport outp* [Function]
: {`rfc.mime`} A procedure to retrieve message body. It is intended to to be a building block of *handler* to be passed to `mime-parse-message`.

  Part-info is a `<mime-part>` object. *Xport* is an input port passed to the handler, from which the MIME part can be read. This procedure read from *xport* until it returns EOF. It also looks at the `transfer-encoding` of *part-info*, and decodes the body accordingly; that is, base64 encoding and quoted-printable encoding is handled. The result is written out to an output port *outp*.

  This procedure does not handle charset conversion. The caller must use CES conversion port as *outp* (see Section 9.4 [Character code conversion], page 339) if desired.

A couple of convenience procedures are defined for typical cases on top of `mime-retrieve-body`.

`mime-body->string` *part-info xport* [Function]
`mime-body->file` *part-info xport filename* [Function]
: {`rfc.mime`} Reads in the body of mime message, decoding transfer encoding, and returns it as a string or writes it to a file, respectively.

The simplest form of MIME message parser would be like this:

```
(let ((headers (rfc822-read-headers port)))
  (mime-parse-message port headers
                      (cut mime-body->string <> <>)))
```

This reads all the message on memory (i.e. the "leaf" `<mime-part>` objects' `content` field would hold the part's body as a string), and returns the top `<mime-part>` object. Content transfer encoding is recognized and handled, but character set conversion isn't done.

You may want to feed the message body to a file directly, or even want to skip some body according to mime media types and/or other header information. Then you can put the logic in the handler closure. That's the reason that this module provides building blocks, instead of all-in-one procedure.

Message composer

`mime-compose-message` *parts :optional port :key boundary* [Function]
`mime-compose-message-string` *parts :key boundary* [Function]

{`rfc.mime`} Composes a MIME multipart message. `Mime-compose-message` emits the result to an output port *port*, whose default is the current output port. `Mime-compose-message-string` makes the result into a string. You can give a boundary string via *boundary* argument; when omitted, a fresh boundary string is automatically generated by `mime-make-boundary` below.

`Mime-compose-message` returns the boundary string. `Mime-compose-message-string` returns two values, the result string and the boundary string.

The content of the message is provided by the *parts* argument, which can be a list of instances of `<mime-part>` (see above) or lists that describe parts. The list form is supported for the caller's convenience, and internally it is converted to a list of `<mime-part>`s.

The syntax of each part element in *parts* are defined as follow.

```
<part>           : <mime-part> | <mime-part-desc>

<mime-part>      : an instance of the class <mime-part>

<mime-part-desc> : (<content-type> (<header> ...) <body>)
<content-type>   : (<type> <subtype> <header-param> ...)
<header-param>   : (<key> . <value>) ...
<header>         : (<header-name> <encoded-header-value>)
                 | (<header-name> (<header-value> <header-param> ...))
<body>           : a string
                 | (file <filename>)
                 | (subparts <part> ...)
```

Note: In the first form of `<header>`, `<encoded-header-value>` must already be encoded using RFC2047 or RFC2231 if the original value contains non-ascii characters. In the second form, we plan to do RFC2231 encoding on behalf of the caller; but the current version does not implement it. The caller should not pass encoded words in this form, since it may result double-encoding when we implement the auto encoding feature; for the time being, the second form restricts ASCII-only values.

If `<body>` is a string, it is used as the part's content. If `<body>` is `(file filename)`, the content is read from the named file. If `<body>` is `(subparts part ...)`, the part becomes nested MIME part.

It is the caller's responsibility to give the proper content. For example, if `<body>` is in the third form, the part must have `multipart` content type.

The caller needs to provide proper `content-transfer-encoding` header, depending on the application. If none is given, the content is inserted into the message as is, which may be appropriate for some applications, but if you want to use the result in email message you certainly want to encode binary part with base64, for example.

`mime-make-boundary` [Function]

{`rfc.mime`} Returns a unique string that can be used as a boundary of a MIME multipart message.

12.40 `rfc.quoted-printable` - Quoted-printable encoding/decoding

`rfc.quoted-printable` [Module]

This module defines a few functions to encode/decode Quoted-printable format, defined in RFC 2045 ([RFC2045], page 829), section 6.7.

`quoted-printable-encode` *:key line-width binary* [Function]

{`rfc.quoted-printable`} Reads byte stream from the current input port, encodes it in Quoted-printable format and writes the result character stream to the current output port. The conversion ends when it reads EOF from the current input port. The keyword argument *line-width* specifies the maximum line width of the generated output in characters. If the encoded output creates a long line, the procedure inserts a "soft line break" so that the each line is equal to or shorter than this number. Soft line breaks are removed when quoted-printable text is decoded. The default line width is 76. (The minimum meaningful number of line-width is 4). You can suppress soft line breaks by giving `#f` or `0` to *line-width*. By default, `quoted-printable-encode` generates `CR-LF` sequence for each line break in the input ("hard line break"). When a true value is given to the keyword argument *binary*, however, octets `#x0a` and `#x0d` in the input are encoded as `=0A` and `=0D`, respectively. See RFC2045 section 6.7 for the details.

`quoted-printable-encode-string` *string :key line-width binary* [Function]

{`rfc.quoted-printable`} Converts contents of *string* to Quoted-printable encoded format. Input string can be either complete or incomplete string; it is always interpreted as a byte sequence.

The keyword arguments are the same as `quoted-printable-encode`.

`quoted-printable-decode` [Function]

{`rfc.quoted-printable`} Reads character stream from the current input port, decodes it from Quoted-printable format and writes the result byte stream to the current output port. The conversion ends when it reads EOF. If it encounters illegal character sequence (such as '=' followed by non-hexadecimal characters), it copies them literally to the output.

`quoted-printable-decode-string` *string* [Function]

{`rfc.quoted-printable`} Decodes a Quoted-printable encoded string *string* and returns the result as a string.

12.41 `rfc.sha` - SHA message digest

`rfc.sha` [Module]

This module implements US Secure Hash Algorithm defined in RFC 4634. It provides SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 (the latter four are sometimes referred as SHA-2 collectively).

The module extends util.digest (see Section 12.63 [Message digester framework], page 795).

`rfc.sha1` [Module]

This is the old module that provided only SHA-1. It is kept as an alias of `rfc.sha` for the backward compatibility. New code should use `rfc.sha`.

`<sha1>` [Class]
`<sha224>` [Class]
`<sha256>` [Class]
`<sha384>` [Class]
`<sha512>` [Class]

{`rfc.sha`} An instance of these class keeps internal state of SHA digest algorithm.

This class implements `util.digest` framework interface, `digest-update!`, `digest-final!`, `digest`, and `digest-string`. See Section 12.63 [Message digester framework], page 795, for detailed explanation of these methods.

Besides the digester framework, this module provides to short-cut procedures.

`sha1-digest` [Function]
`sha224-digest` [Function]
`sha256-digest` [Function]
`sha384-digest` [Function]
`sha512-digest` [Function]

{`rfc.sha`} Reads data from the current input port until EOF, and returns its digest in an incomplete string.

`sha1-digest-string` *string* [Function]
`sha224-digest-string` *string* [Function]
`sha256-digest-string` *string* [Function]
`sha384-digest-string` *string* [Function]
`sha512-digest-string` *string* [Function]

{`rfc.sha`} Digest the data in *string*, and returns the result in an incomplete string.

12.42 `rfc.tls` - Transport layer security

`rfc.tls` [Module]

This module handles secure connection over TCP socket. This module is used by `rfc.http` (see Section 12.34 [HTTP], page 721).

We haven't yet got other use cases than https connections, so we're not sure how API of this layer should look like. At this moment, we document the minimal features you need to know to use the TLS layer with `rfc.http`.

Gauche supports two TLS subsystems - one based on axTLS (`http://axtls.sourceforge.net/`), and the other based on mbedTLS (`https://tls.mbed.org/`). Whether they're included depends on the configuration options. By default, axTLS support is compiled in, and mbedTLS support is only included if the build platform has mbedTLS library installed. And axTLS is set to be used by default.

Whether the running Gauche has any of TLS support can be checked with a feature identifier `gauche.net.tls`. Availability of each individual subsystems can be checked with feature identifiers `gauche.net.tls.axtls` and `gauche.net.tls.mbedtls`, respectively. See Section 4.12 [Feature conditional], page 68, for more about feature identifiers.

In the current version, we verify certificates by default. You need to specify the location of CA certificates explicitly when you want to verify, unless the CA certificate location is specified at the configuration time. In future, we might set the default CA certificate file automatically so that users don't need to bother by default, but not now.

`<ax-tls>` [Class]

{`rfc.tls`} A class that implements axTLS subsystem interface.

`<mbed-tls>` [Class]

{`rfc.tls`} A class that implements mbedTLS subsystem interface.

`default-tls-class` *:optional class* [Parameter]

{`rfc.tls`} Set/get the default TLS subsystem to be used. Without arguments, it return a class (either `<ax-tls>` or `<mbed-tls>` to be used. With one argument, which must be either `<ax-tls>` or `<mbed-tls>`, changes the default and returns the previous value.

`tls-ca-bundle-path` *:optional path* [Parameter]

{`rfc.tls`} Set/get the CA certificate bundle path to be used. Without arguments, it returns the current path. With one argument, a pathname to the CA bundle file, updates the parameter to the new value and returns the previous value.

Some platform can load CA bundle from system certificate store. If you sets `path` to symbol `system` on such platform, use system certificate store as CA certificate bundle.

If you use mbedTLS with CA bundle file, you need to set this value to the valid CA bundle file path. Unfortunately there's no globally agreed location for such file. If you need one, one choice is to fetch it from `https://curl.haxx.se/ca/cacert.pem`, store it locally and set its path to `tls-ca-bundle-path`. (We can't automatically do that, since we can't securely fetch the file before we get valid CA certs!)

12.43 `rfc.uri` - URI parsing and construction

`rfc.uri` [Module]

Provides a set of procedures to parse and construct Uniform Resource Identifiers defined in RFC 2396 ([RFC2396], page 829), as well as Data URI scheme defined in RFC2397.

First, lets review the structure of URI briefly. The following graph shows how the URI is constructed:

```
URI-+-scheme
    |
    +-specific--+--authority-+--userinfo
                |            +--host
                |            +--port
                +--path
                +--query
                +--fragment
```

Not all URIs have this full hierarchy. For example, `mailto:admin@example.com` has only *scheme* (`mailto`) and *specific* (`admin@example.com`) parts.

Most popular URI schemes, however, organize resources in a tree, so they adopt *authority* (which usually identifies the server) and the hierarchical *path*. In the URI `http://example.com:8080/search?q=key#results`, the authority part is `example.com:8080`, the path is `/search`, the query is `key` and the fragment is `results`. The userinfo can be provided before hostname, such as `anonymous` in `ftp://anonymous@example.com/pub/`.

We have procedures that decompose a URI into those parts, and that compose a URI from those parts.

Parsing URI

`uri-ref` *uri parts* [Function]

{`rfc.uri`} Extract specific part(s) from the given URI. You can fully decompose URI by the procedures described below, but in actual applications, you often need only some of the parts. This procedure comes handy for it.

The *parts* argument may be a symbol, or a list of symbols, to name the desired parts. The recognized symbos are as follows.

`scheme` The scheme part, as string.

`authority`
: The authority part, as string. If URI doesn't have the part, `#f`.

`userinfo` The userinfo part, as string. If URI doesn't have the part, `#f`.

`host` The host part, as string. If URI doesn't have the part, `#f`.

`port` The port part, as integer. If URI doesn't have the part, `#f`.

`path` The path part, as string. If URI isn't hierarchical, this returns the specific part.

`query` The query part, as string. If URI doesn't have the part, `#f`.

`fragment` The fragment part, as string. If URI doesn't have the part, `#f`.

`scheme+authority`
The scheme and authority part.

`host+port`
The host and port part.

`userinfo+host+port`
The userinfo, host and port part.

`path+query`
The path and query part.

`path+query+fragment`
The path, query and fragment part.

```
(define uri "http://foo:bar@example.com:8080/search?q=word#results")

(uri-ref uri 'scheme)             ⇒ "http"
(uri-ref uri 'authority)          ⇒ "//foo:bar@example.com:8080/"
(uri-ref uri 'userinfo)           ⇒ "foo:bar"
(uri-ref uri 'host)               ⇒ "example.com"
(uri-ref uri 'port)               ⇒ 8080
(uri-ref uri 'path)               ⇒ "/search"
(uri-ref uri 'query)              ⇒ "q=word"
(uri-ref uri 'fragment)           ⇒ "results"
(uri-ref uri 'scheme+authority)   ⇒ "http://foo:bar@example.com:8080/"
(uri-ref uri 'host+port)          ⇒ "example.com:8080"
(uri-ref uri 'userinfo+host+port) ⇒ "foo:bar@example.com:8080"
(uri-ref uri 'path+query)         ⇒ "/search?q=word"
(uri-ref uri 'path+query+fragment)⇒ "/search?q=word#results"
```

You can extract multiple parts at once by specifying a list of parts. A list of parts is returned.

```
(uri-ref uri '(host+port path+query))
  ⇒ ("example.com:8080" "/search?q=word")
```

`uri-parse` *uri* [Function]
`uri-scheme&specific` *uri* [Function]
`uri-decompose-hierarchical` *specific* [Function]
`uri-decompose-authority` *authority* [Function]

{`rfc.uri`} General parser of URI. These functions does not decode URI encoding, since the parts to be decoded differ among the uri schemes. After parsing uri, use `uri-decode` below to decode them.

`uri-parse` is the most handy procedure. It breaks the uri into the following parts and returns them as multiple values. If the uri doesn't have the corresponding parts, `#f` are returned for the parts.

- URI scheme as a string (e.g. `"mailto"` in `"mailto:foo@example.com"`).
- User-info in the authority part (e.g. `"anonymous"` in `ftp://anonymous@ftp.example.com/pub/foo`).

- Hostname in the authority part (e.g. `"ftp.example.com"` in `ftp://anonymous@ftp.example.com/pub/foo`).
- Port number in the authority part, as an integer (e.g. `8080` in `http://www.example.com:8080/`).
- Path part (e.g. `"/index.html"` in `http://www.example.com/index.html`).
- Query part (e.g. `"key=xyz&lang=en"` in `http://www.example.com/search?key=xyz&lang=en`).
- Fragment part (e.g. `"section4"` in `http://www.example.com/document.html#section4`).

The following procedures are finer grained and break up uris with different stages.

`uri-scheme&specific` takes a URI *uri*, and returns two values, its scheme part and its scheme-specific part. If *uri* doesn't have a scheme part, `#f` is returned for it.

```
(uri-scheme&specific "mailto:sclaus@north.pole")
  ⇒ "mailto" and "sclaus@north.pole"
(uri-scheme&specific "/icons/new.gif")
  ⇒ #f and "/icons/new.gif"
```

If the URI scheme uses hierarchical notation, i.e. "`//`*`authority`*`/`*`path`*`?`*`query`*`#`*`fragment`*", you can pass the scheme-specific part to `uri-decompose-hierarchical` and it returns four values, *authority*, *path*, *query* and *fragment*.

```
(uri-decompose-hierarchical "//www.foo.com/about/company.html")
  ⇒ "www.foo.com", "/about/company.html", #f and #f
(uri-decompose-hierarchical "//zzz.org/search?key=%3fhelp")
  ⇒ "zzz.org", "/search", "key=%3fhelp" and #f
(uri-decompose-hierarchical "//jjj.jp/index.html#whatsnew")
  ⇒ "jjj.jp", "/index.html", #f and "whatsnew"
(uri-decompose-hierarchical "my@address")
  ⇒ #f, #f, #f and #f
```

Furthermore, you can parse *authority* part of the hierarchical URI by `uri-decompose-authority`. It returns *userinfo*, *host* and *port*.

```
(uri-decompose-authority "yyy.jp:8080")
  ⇒ #f, "yyy.jp" and "8080"
(uri-decompose-authority "[::1]:8080")  ;(IPv6 host address)
  ⇒ #f, "::1" and "8080"
(uri-decompose-authority "mylogin@yyy.jp")
  ⇒ "mylogin", "yyy.jp" and #f
```

`uri-decompose-data` *uri* [Function]

{`rfc.uri`} Parse a Data URI string *uri*. You can either pass the entire uri including `data:` scheme part, or just the specific part. If the passed uri is invalid as a data uri, an error is signalled.

Returns two values: parsed content type and the decoded data. The data is a string if the content type is `text/*`, and a u8vector otherwise.

The content-type is parsed by `mime-parse-content-type` (see Section 12.39 [MIME message handling], page 730). The result format is a list as follows:

```
(type subtype (attribute . value) ...).
```

Here are a couple of examples:

```
(uri-decompose-data
 "data:text/plain;charset=utf-8;base64,KGhlbGxvIHdvcmxkKQ==")
  ⇒ ("text" "plain" ("charset" . "utf-8")) and "(hello world)"
```

```
(uri-decompose-data
 "data:application/octet-stream;base64,AAECAw==")
  ⇒ ("application" "octet-stream") and #u8(0 1 2 3)
```

Constructing URI

`uri-compose` *:key scheme userinfo host port authority path path* query fragment specific* [Function]

{`rfc.uri`} Compose a URI from given components. There can be various combinations of components to create a valid URI—the following diagram shows the possible 'paths' of combinations:

```
        /-----------------specific-------------------\
        |                                            |
 scheme-+------authority-----+-+-------path*---------+-
        |                    | |                     |
        \-userinfo-host-port-/ \-path-query-fragment-/
```

If `#f` is given to a keyword argument, it is equivalent to the absence of that keyword argument. It is particularly useful to pass the results of parsed uri.

If a component contains a character that is not appropriate for that component, it must be properly escaped before being passed to `url-compose`.

Some examples:

```
(uri-compose :scheme "http" :host "foo.com" :port 80
             :path "/index.html" :fragment "top")
  ⇒ "http://foo.com:80/index.html#top"

(uri-compose :scheme "http" :host "foo.net"
             :path* "/cgi-bin/query.cgi?keyword=foo")
  ⇒ "http://foo.net/cgi-bin/query.cgi?keyword=foo"

(uri-compose :scheme "mailto" :specific "a@foo.org")
  ⇒ "mailto:a@foo.org"

(receive (authority path query fragment)
   (uri-decompose-hierarchical "//foo.jp/index.html#whatsnew")
 (uri-compose :authority authority :path path
              :query query :fragment fragment))
  ⇒ "//foo.jp/index.html#whatsnew"
```

`uri-merge` *base-uri relative-uri relative-uri2 . . .* [Function]

{`rfc.uri`} Arguments are strings representing full or part of URIs. This procedure resolves *relative-uri* in relative to *base-uri*, as defined in RFC3986 Section 5.2. “Relative Resolution”.

If more *relative-uri2*s are given, first *relative-uri* is merged to *base-uri*, then the next argument is merged to the resulting uri, and so on.

```
(uri-merge "http://example.com/foo/index.html" "a/b/c")
 ⇒ "http://example.com/foo/a/b/c"

(uri-merge "http://example.com/foo/search?q=abc" "../about#me")
 ⇒ "http://example.com/about#me"

(uri-merge "http://example.com/foo" "http://example.net/bar")
 ⇒ "http://example.net/bar"
```

```
(uri-merge "http://example.com/foo/" "q" "?xyz")
 ⇒ "http://example.com/foo/q?xyz"
```

`uri-compose-data` *data :key content-type encoding* [Function]

{`rfc.uri`} Creates a Data URI of the given *data*, with specified content-type and transfer encoding. Returns a string.

The *data* argument must be a string or a u8vector.

The *content-type* argument can be `#f` (default), a string that represents a content type (e.g. `"text/plain;charset=utf-8"`), or a list form of parsed content type (e.g. `("application" "octet-stream")`. If it is `#f`, `text/plain` with the gauche's native character encoding is used when *data* is a complete string, and `application/octet-stream` is used otherwise.

The *encoding* argument can be either `#f` (default), or a symbol `uri` or `base64`. This is for transfer encoding, not character encoding. If it is `#f`, URI encoding is used for text data and base64 encoding is used for binary data.

```
(uri-compose-data "(hello world)")
 ⇒ "data:text/plain;charset=utf-8,%28hello%20world%29"

(uri-compose-data "(hello world)" :encoding 'base64)
 ⇒ "data:text/plain;charset=utf-8;base64,KGhlbGxvIHdvcmxkKQ=="

(uri-compose-data '#u8(0 1 2 3))
 ⇒ "data:application/octet-stream;base64,AAECAw=="
```

URI Encoding and decoding

`uri-decode` *:key :cgi-decode* [Function]
`uri-decode-string` *string :key :cgi-decode :encoding* [Function]

{`rfc.uri`} Decodes "URI encoding", i.e. %-escapes. `uri-decode` takes input from the current input port, and writes decoded result to the current output port. `uri-decode-string` takes input from *string* and returns decoded string.

If *cgi-decode* is true, also replaces `+` to a space character.

To `uri-decode-string` you can provide the external character encoding by the *encoding* keyword argument. When it is given, the decoded octet sequence is assumed to be in the specified encoding and converted to the Gauche's internal character encoding.

`uri-encode` *:key :noescape* [Function]
`uri-encode-string` *string :key :noescape :encoding* [Function]

{`rfc.uri`} Encodes unsafe characters by %-escape. `uri-encode` takes input from the current input port and writes the result to the current output port. `uri-encode-string` takes input from *string* and returns the encoded string.

By default, characters that are not specified "unreserved" in RFC3986 are escaped. You can pass different character set to *noescape* argument to keep from being encoded. For example, the older RFC2396 has several more "unreserved" characters, and passing `*rfc2396-unreserved-char-set*` (see below) prevents those characters from being escaped.

The multibyte characters are encoded as the octet stream of Gauche's native multibyte representation by default. However, you can pass the `encoding` keyword argument to `uri-encode-string`, to convert *string* to the specified character encoding.

`*rfc2396-unreserved-char-set*` [Constant]
`*rfc3986-unreserved-char-set*` [Constant]

{`rfc.uri`} These constants are bound to character sets that represents "unreserved" characters defined in RFC2396 and RFC3986, respectively. (See Section 6.11 [Character set], page 147, and Section 11.6 [Character-set library], page 600, for operations on character sets).

12.44 `rfc.zlib` - zlib compression library

`rfc.zlib` [Module]

This module provides bindings to zlib compression library. Most features of zlib can be used through this module.

Zlib supports reading and writing of Zlib compressed data format (RFC1950), DEFLATE compressed data format (RFC1951), and GZIP file format (RFC1052). It also provides procedures to calculate CRC32 and Adler32 checksums.

Compression and decompression are done through specialized ports. There are number of parameters to fine-tune compression; refer to zlib documentation for the details.

Condition types

The following condition types are defined to represent errors during processing by zlib.

`<zlib-error>` [Condition Type]

{`rfc.zlib`} Subclass of `<error>` and superclass of the following condition types. This class is an abstract class to catch any of the zlib-specific errors. Zlib-specific errors raised by procedures in `rfc.zlib` are always an instance (or a compound condition including) one of the following specific classes.

`<zlib-need-dict-error>` [Condition Type]
`<zlib-stream-error>` [Condition Type]
`<zlib-data-error>` [Condition Type]
`<zlib-memory-error>` [Condition Type]
`<zlib-version-error>` [Condition Type]

{`rfc.zlib`} Subclasses of `<zlib-error>`. Those condition type correspond to zlib's `Z_NEED_DICT_ERROR`, `Z_STREAM_ERROR`, `Z_DATA_ERROR`, `Z_MEMORY_ERROR`, and `Z_VERSION_ERROR` errors.

When an error occurs during reading data, a compound condition of a subclass of `<zlib-error>` and `<io-read-error>` is raised. When an error occurs without I/O, a simple condition of a subclass of `<zlib-error>` is raised. Errors unrelated to zlib, such as invalid argument error, would be a simple `<error>` condition.

Compression/decompression ports

`<deflating-port>` [Class]
`<inflating-port>` [Class]

{`rfc.zlib`} Compression and decompression functions are provided via ports. A *deflating port* is an output port that compresses the output data. An *inflating port* is an input that reads compressed data and decompress it.

When an inflating port encounters a corrupted compressed data, a compound condition of `<io-read-error>` and `<zlib-data-error>` is raised during read operation.

`open-deflating-port` *drain :key compression-level buffer-size window-bits memory-level strategy dictionary owner?* [Function]

{`rfc.zlib`} Creates and returns an instance of `<deflating-port>`, an output port that compresses the output data and sends the compressed data to another output port *drain*. This combines the functionality of zlib's `deflateInit2()` and `deflateSetDictionary()`.

You can specify an exact integer between 1 and 9 (inclusive) to *compression-level*. Larger integer means larger compression ratio. When omitted, a default compression level is used, which is usually 6.

The following constants are defined to specify *compression-level* conveniently:

`Z_NO_COMPRESSION` [Constant]
`Z_BEST_SPEED` [Constant]
`Z_BEST_COMPRESSION` [Constant]
`Z_DEFAULT_COMPRESSION` [Constant]

{`rfc.zlib`}

The *buffer-size* argument specifies the buffer size of the port in bytes. The default is 4096.

The *window-bits* argument specifies the size of the window in exact integer. Typically the value should be between 8 and 15, inclusive, and it specifies the base two logarithm of the window size used in compression. Larger number yields better compression ratio, but more memory usage. The default value is 15.

There are a couple of special modes specifiable by *window-bits*. When an integer between -8 and -15 is given to *window-bits*, the port produces a raw deflated data, that lacks zlib header and trailer. In this case, Adler32 checksum isn't calculated. The actual window size is determined by the absolute value of *window-bits*.

When *window-bits* is between 24 and 31, the port uses GZIP encoding; that is, instead of zlib wrapper, the compressed data is enveloped by simple gzip header and trailer. The gzip header added by this case doesn't have filenames, comments, header CRC and other data, and have zero modified time, and 255 (unknown) in the OS field. The `zstream-adler32` procedure will return CRC32 checksum instead of Adler32. The actual window size is determined by *window-bits*-16.

The *memory-level* argument specifies how much memory should be allocated to keep the internal state during compression. 1 means smallest memory, which causes slow and less compression. 9 means fastest and best compression with largest amount of memory. The default value is 8.

To fine tune compression algorithm, you can use the *strategy* argument. The following constants are defined as the valid value as *strategy*:

`Z_DEFAULT_STRATEGY` [Constant]

{`rfc.zlib`} The default strategy, suitable for most ordinary data.

`Z_FILTERED` [Constant]

{`rfc.zlib`} Suitable for data generated by filters. Filtered data consists mostly of small values with a random distribution, and this makes the compression algorithm to use more huffman encoding and less string match.

`Z_HUFFMAN_ONLY` [Constant]

{`rfc.zlib`} Force huffman encoding only (no string match).

`Z_RLE` [Constant]

{`rfc.zlib`} Limit match distance to 1 (that is, to force run-length encoding). It is as fast as `Z_HUFFMAN_ONLY` and gives better compression for png image data.

`Z_FIXED` [Constant]

{`rfc.zlib`} Prohibits dynamic huffman encoding. It allows a simple decoder for special applications.

The choice of *strategy* only affects compression ratio and speed. Any choice produces correct and decompressable data.

You can give an initial dictionary to the *dictionary* argument to be used in compression. The compressor and decompressor must use exactly the same dictionary. See the zlib documentation for the details.

By default, a deflating port leaves *drain* open after all conversion is done, i.e. the deflating port itself is closed. If you don't want to bother closing *drain*, give a true value to the *owner?* argument; then *drain* is closed after the deflating port is closed and all data is written out.

Note: You *must* close a deflating port explicitly, or the compressed data can be chopped prematurely. When you leave a deflating port open to be GCed, the finalizer will close it; however, the order in which finalizers are called is undeterministic, and it is possible that the *drain* port is closed before the deflating port is closed. In such cases, the deflating port's attempt to flush the buffered data and trailer will fail.

`open-inflating-port` *source :key buffer-size window-bits dictionary owner?* [Function]

{`rfc.zlib`} Takes an input port *source* from which a compressed data can be read, and creates and returns a new instance of `<inflating-port>`, that is, a port that allows decompressed data from it. This procedure covers zlib's functionality of `inflateInit2()` and `inflateSetDictionary()`.

The meaning of *buffer-size* and *owner* are the same as `open-deflating-port`.

The meaning of *window-bits* is almost the same, except that if a value increased by 32 is given, the inflating port automatically detects whether the source stream is zlib or gzip by its header.

If the input data is compressed with specified dictionary, the same dictionary must be given to the *dictionary* argument. Otherwise, a compound condition of `<io-read-error>` and `<zlib-need-dict-error>` will be raised.

Operations on inflating/deflating ports

`zstream-total-in` *xflating-port* [Function]
`zstream-total-out` *xflating-port* [Function]
`zstream-adler32` *xflating-port* [Function]
`zstream-data-type` *xflating-port* [Function]

{`rfc.zlib`} The *xflating-port* argument must be either inflating and deflating port, or an error is raised.

Returns the value of `total_in`, `total_out`, *adler32*, and `data_type` fields of the `z_stream` structure associated to the given inflating or deflating port, respectively.

The value of `data_type` can be one of the following constants:

`Z_BINARY` [Constant]
`Z_TEXT` [Constant]
`Z_ASCII` [Constant]
`Z_UNKNOWN` [Constant]

{`rfc.zlib`}

`zstream-params-set!` *deflating-port :key compression-level strategy* [Function]

{`rfc.zlib`} Changes compression level and/or strategy during compressing.

`zstream-dictionary-adler32` *deflating-port* [Function]

{`rfc.zlib`} When a dictionary is given to `open-deflating-port`, the dictionary's adler32 checksum is calculated. This procedure returns the checksum. If no dictionary has been given, this procedure returns `#f`.

`deflating-port-full-flush` *deflating-port* [Function]

{`rfc.zlib`} Flush the data buffered in the *deflating-port*, and resets compression state. The decompression routine can skip the data to the full-flush point by `inflate-sync`.

`inflate-sync` *inflating-port* [Function]

{`rfc.zlib`} Skip the (possibly corrupted) compressed data up to the next full-flush point marked by `deflating-port-full-flush`. You may want to use this procedure when you get `<zlib-data-error>`. Returns the number of bytes skipped when the next full-flush point is found, or `#f` when the input reaches EOF before finding the next point.

Miscellaneous API

`zlib-version` [Function]

{`rfc.zlib`} Returns Zlib's version in string.

`deflate-string` *string options . . .* [Function]

{`rfc.zlib`} Compresses the given string and returns zlib-compressed data in a string. All optional arguments are passed to `open-deflating-port` as they are.

`inflate-string` *string options . . .* [Function]

{`rfc.zlib`} Takes zlib-compressed data in string, and returns decompressed data in a string. All optional arguments are passed to `open-inflating-port` as they are.

`gzip-encode-string` *string options . . .* [Function]
`gzip-decode-string` *string options . . .* [Function]

{`rfc.zlib`} Like `deflate-string` and `inflate-string`, but uses the gzip format instead. It is same as giving more than 15 to the *window-bits* argument of `deflate-string` and `inflate-string`.

`crc32` *string :optional checksum* [Function]

{`rfc.zlib`} Returns CRC32 checksum of *string*. If optional *checksum* is given, the returned checksum is an update of *checksum* by *string*.

`adler32` *string :optional checksum* [Function]

{`rfc.zlib`} Returns Adler32 checksum of *string*. If optional *checksum* is given, the returned checksum is an update of *checksum* by *string*.

Calculating Adler32 is faster than CRC32, but it is known to produce uneven distribution of hash values for small input. See RFC3309 for the detailed description. If it matters, use CRC32 instead.

12.45 `slib` - SLIB interface

`slib` [Module]

This module is the interface to the Aubrey Jaffer's SLIB. To use SLIB, say `(use slib)`. SLIB itself is not included in Gauche distribution. If you don't have it on your system, get it from `http://www-swiss.ai.mit.edu/~jaffer/SLIB.html`.

By default, the SLIB installation is searched from the directory specified at the Gauche configuration. If SLIB isn't there, an error is signaled. In that case, you can set the environment variable `SCHEME_LIBRARY_PATH` to point to the SLIB installation path.

This module redefines `require`, shadowing the Gauche's original `require`. If it gets a symbol as an argument, it works as SLIB's `require`, while if it gets a string, it works as Gauche's `require`. The same applies to `provide` and `provided?`.

All SLIB symbol bindings, loaded by `require`, stay in the module `slib`.

```
(use slib)          ; load and set up slib
(require 'getopt)   ; load SLIB's getopt module
(require "foo")     ; load Gauche's foo module
```

12.46 `sxml.ssax` - Functional XML parser

`sxml.ssax` [Module]

`sxml.*` modules are the adaptation of Oleg Kiselyov's SXML framework ([SSAX], page 831), which is based on S-expression representation of XML structure.

SSAX is a parser part of SXML framework. This is a quote from SSAX webpage:

> A SSAX functional XML parsing framework consists of a DOM/SXML parser, a SAX parser, and a supporting library of lexing and parsing procedures. The procedures in the package can be used separately to tokenize or parse various pieces of XML documents. The framework supports XML Namespaces, character, internal and external parsed entities, attribute value normalization, processing instructions and CDATA sections. The package includes a semi-validating SXML parser : a DOM-mode parser that is an instantiation of a SAX parser (called SSAX).

The current version is based on the SSAX CVS version newer than the last 'official' release of SXML toolset (4.9), and SXML-gauche-0.9 package which was based on SXML-4.9. There is an important change from that release. Now the API uses lowercase letter suffix `ssax:` instead of uppercase `SSAX:`—the difference matters since Gauche is case sensitive by default. Alias names are defined for backward compatibility, but the use of uppercase suffixed names are deprecated.

I derived the content of this part of the manual from SSAX source code, just by converting its comments into texinfo format. The original text is by Oleg Kiselyov. Shiro Kawai should be responsible for any typographical error or formatting error introduced by conversion.

The manual entries are ordered in "bottom-up" way, beginning from the lower-level constructs towards the high-level utilities. If you just want to parse XML document and obtain SXML, check out `ssax:xml->sxml` in Section 12.46.4 [SSAX Highest-level parsers - XML to SXML], page 755.

12.46.1 SSAX data types

TAG-KIND
: a symbol '`START`, '`END`, '`PI`, '`DECL`, '`COMMENT`, '`CDSECT` or '`ENTITY-REF` that identifies a markup token.

UNRES-NAME
: a name (called `GI` in the XML Recommendation) as given in an xml document for a markup token: start-tag, `PI` target, attribute name. If a `GI` is an `NCName`, *UNRES-NAME* is this `NCName` converted into a Scheme symbol. If a `GI` is a `QName`, *UNRES-NAME* is a pair of symbols: (*PREFIX* . *LOCALPART*)

RES-NAME
: An expanded name, a resolved version of an *UNRES-NAME*. For an element or an attribute name with a non-empty namespace URI, *RES-NAME* is a pair of symbols, (*URI-SYMB* . *LOCALPART*). Otherwise, it's a single symbol.

ELEM-CONTENT-MODEL

A symbol:

`ANY`	anything goes, expect an END tag.
`EMPTY-TAG`	no content, and no END-tag is coming.
`EMPTY`	no content, expect the END-tag as the next token.
`PCDATA`	expect character data only, and no children elements.
`MIXED`	
`ELEM-CONTENT`	

URI-SYMB

A symbol representing a namespace URI – or other symbol chosen by the user to represent URI. In the former case, *URI-SYMB* is created by `%`-quoting of bad URI characters and converting the resulting string into a symbol.

NAMESPACES

A list representing namespaces in effect. An element of the list has one of the following forms:

`(`*`prefix uri-symb . uri-symb`*`)`

or,

`(`*`prefix user-prefix . uri-symb`*`)`

user-prefix is a symbol chosen by the user to represent the URI.

`(#f `*`user-prefix . uri-symb`*`)`

Specification of the user-chosen prefix and a *uri-symbol*.

`(*DEFAULT* `*`user-prefix . uri-symb`*`)`

Declaration of the default namespace

`(*DEFAULT* #f . #f)`

Un-declaration of the default namespace. This notation represents overriding of the previous declaration

A *NAMESPACES* list may contain several elements for the same *PREFIX*. The one closest to the beginning of the list takes effect.

ATTLIST An ordered collection of (*NAME* . *VALUE*) pairs, where *NAME* is a *RES-NAME* or an *UNRES-NAME*. The collection is an ADT.

STR-HANDLER

A procedure of three arguments: `(`*`string1 string2 seed`*`)` returning a new *seed*. The procedure is supposed to handle a chunk of character data *string1* followed by a chunk of character data *string2*. *string2* is a short string, often `"\n"` and even `""`

ENTITIES

An assoc list of pairs:

`(`*`named-entity-name . named-entity-body`*`)`

where *named-entity-name* is a symbol under which the entity was declared, *named-entity-body* is either a string, or (for an external entity) a thunk that will return an input port (from which the entity can be read). *named-entity-body* may also be `#f`. This is an indication that a *named-entity-name* is currently being expanded. A reference to this *named-entity-name* will be an error: violation of the WFC nonrecursion.

XML-TOKEN

A record with two slots, *kind* and *token*. This record represents a markup, which is, according to the XML Recommendation, "takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, and processing instructions."

kind a *TAG-KIND*

head an *UNRES-NAME*. For xml-tokens of kinds 'COMMENT and 'CDSECT, the head is #f

For example,

```
<P>  => kind='START, head='P
</P> => kind='END, head='P
<BR/> => kind='EMPTY-EL, head='BR
<!DOCTYPE OMF ...> => kind='DECL, head='DOCTYPE
<?xml version="1.0"?> => kind='PI, head='xml
&my-ent; => kind = 'ENTITY-REF, head='my-ent
```

Character references are not represented by xml-tokens as these references are transparently resolved into the corresponding characters.

XML-DECL

A record with three slots, *elems*, *entities*, and *notations*.

The record represents a datatype of an XML document: the list of declared elements and their attributes, declared notations, list of replacement strings or loading procedures for parsed general entities, etc. Normally an xml-decl record is created from a DTD or an XML Schema, although it can be created and filled in in many other ways (e.g., loaded from a file).

elems: an (assoc) list of decl-elem or `#f`. The latter instructs the parser to do no validation of elements and attributes.

decl-elem: declaration of one element: `(elem-name elem-content decl-attrs)`; *elem-name* is an *UNRES-NAME* for the element. *elem-content* is an *ELEM-CONTENT-MODEL*. *decl-attrs* is an *ATTLIST*, of `(attr-name . value)` associations. This element can declare a user procedure to handle parsing of an element (e.g., to do a custom validation, or to build a hash of IDs as they're encountered).

decl-attr: an element of an *ATTLIST*, declaration of one attribute `(attr-name content-type use-type default-value)`: *attr-name* is an *UNRES-NAME* for the declared attribute; *content-type* is a symbol: `CDATA`, *NMTOKEN*, *NMTOKENS*, ...; or a list of strings for the enumerated type. *use-type* is a symbol: `REQUIRED`, `IMPLIED`, `FIXED` default-value is a string for the default value, or `#f` if not given.

`make-empty-attlist` [Function]
`attlist-add` *attlist name-value* [Function]
`attlist-null?` [Function]
`attlist-remove-top` *attlist* [Function]
`attlist->alist` *attlist* [Function]
`attlist-fold` [Function]

{`sxml.ssax`} Utility procedures to deal with attribute list, which keeps name-value association.

`make-xml-token` *kind head* [Function]
`xml-token?` *token* [Function]

{`sxml.ssax`} A constructor and a predicate for a *XML-TOKEN* record.

`xml-token-kind` *token* [Macro]
`xml-token-head` *token* [Macro]

{`sxml.ssax`} Accessor macros of a *XML-TOKEN* record.

12.46.2 SSAX low-level parsing code

They deal with primitive lexical units (Names, whitespaces, tags) and with pieces of more generic productions. Most of these parsers must be called in appropriate context. For example, `ssax:complete-start-tag` must be called only when the start-tag has been detected and its `GI` has been read.

`ssax:skip-S` *port* [Function]

{`sxml.ssax`} Skip the S (whitespace) production as defined by

```
[3] S ::= (#x20 | #x9 | #xD | #xA)
```

The procedure returns the first not-whitespace character it encounters while scanning the *port*. This character is left on the input stream.

`ssax:ncname-starting-char?` *a-char* [Function]

{`sxml.ssax`} Check to see if a-char may start a `NCName`.

`ssax:read-NCName` *port* [Function]

{`sxml.ssax`} Read a `NCName` starting from the current position in the *port* and return it as a symbol.

`ssax:read-QName` *port* [Function]

{`sxml.ssax`} Read a (namespace-) Qualified Name, `QName`, from the current position in the *port*.

From REC-xml-names:

```
[6] QName ::= (Prefix ':')? LocalPart
[7] Prefix ::= NCName
[8] LocalPart ::= NCName
```

Return: an *UNRES-NAME*.

`ssax:Prefix-XML` [Variable]

{`sxml.ssax`} The prefix of the pre-defined XML namespace, i.e. ’`xml`.

`ssax:read-markup-token` *port* [Function]

{`sxml.ssax`} This procedure starts parsing of a markup token. The current position in the stream must be `#\<`. This procedure scans enough of the input stream to figure out what kind of a markup token it is seeing. The procedure returns an xml-token structure describing the token. Note, generally reading of the current markup is not finished! In particular, no attributes of the start-tag token are scanned.

Here’s a detailed break out of the return values and the position in the *port* when that particular value is returned:

`PI-token` only `PI`-target is read. To finish the Processing Instruction and disregard it, call `ssax:skip-pi`. `ssax:read-attributes` may be useful as well (for `PI`s whose content is attribute-value pairs)

`END-token`

The end tag is read completely; the current position is right after the terminating `#\>` character.

`COMMENT` is read and skipped completely. The current position is right after `"-->"` that terminates the comment.

`CDSECT` The current position is right after `"<!CDATA["`. Use `ssax:read-cdata-body` to read the rest.

`DECL` We have read the keyword (the one that follows `"<!"`) identifying this declaration markup. The current position is after the keyword (usually a whitespace character)

`START-token`
We have read the keyword (`GI`) of this start tag. No attributes are scanned yet. We don't know if this tag has an empty content either. Use `ssax:complete-start-tag` to finish parsing of the token.

`ssax:skip-pi` *port* [Function]
{`sxml.ssax`} The current position is inside a `PI`. Skip till the rest of the `PI`.

`ssax:read-pi-body-as-string` *port* [Function]
{`sxml.ssax`} The current position is right after reading the `PITarget`. We read the body of `PI` and return it as a string. The port will point to the character right after '`?>`' combination that terminates `PI`.

```
[16] PI ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
```

`ssax:skip-internal-dtd` *port* [Function]
{`sxml.ssax`} The current pos in the port is inside an internal DTD subset (e.g., after reading `#\[` that begins an internal DTD subset) Skip until the `"]>"` combination that terminates this DTD

`ssax:read-cdata-body` *port str-handler seed* [Function]
{`sxml.ssax`} This procedure must be called after we have read a string `"<![CDATA["` that begins a `CDATA` section. The current position must be the first position of the `CDATA` body. This function reads *lines* of the `CDATA` body and passes them to a *STR-HANDLER*, a character data consumer.

The str-handler is a *STR-HANDLER*, a procedure `string1` *string2 seed*. The first *string1* argument to *STR-HANDLER* never contains a newline. The second *string2* argument often will. On the first invocation of the *STR-HANDLER*, the seed is the one passed to `ssax:read-cdata-body` as the third argument. The result of this first invocation will be passed as the seed argument to the second invocation of the line consumer, and so on. The result of the last invocation of the *STR-HANDLER* is returned by the `ssax:read-cdata-body`. Note a similarity to the fundamental '`fold`' iterator.

Within a `CDATA` section all characters are taken at their face value, with only three exceptions:

- `CR`, `LF`, and `CRLF` are treated as line delimiters, and passed as a single `#\newline` to the *STR-HANDLER*.
- `"]]>"` combination is the end of the `CDATA` section.
- `>` is treated as an embedded `#\>` character. Note, `<` and `&` are not specially recognized (and are not expanded)!

`ssax:read-char-ref` *port* [Function]
{`sxml.ssax`}

```
[66]  CharRef ::=  '&#' [0-9]+ ';'
                 | '&#x' [0-9a-fA-F]+ ';'
```

This procedure must be called after we we have read `"&#"` that introduces a char reference. The procedure reads this reference and returns the corresponding char. The current position in *port* will be after `";"` that terminates the char reference. Faults detected: `WFC: XML-Spec.html#wf-Legalchar`.

According to Section "4.1 Character and Entity References" of the XML Recommendation:

> "[Definition: A character reference refers to a specific character in the ISO/IEC 10646 character set, for example one not directly accessible from available input devices.]"

Therefore, we use a `ucscode->char` function to convert a character code into the character – *regardless* of the current character encoding of the input stream.

`ssax:handle-parsed-entity` *port name entities content-handler str-handler seed* [Function]

{`sxml.ssax`} Expand and handle a parsed-entity reference

- *port* - a PORT
- *name* - the name of the parsed entity to expand, a symbol.
- *entities* - see *ENTITIES*
- *content-handler* - procedure *port entities seed* that is supposed to return a *seed*.
- *str-handler* - a *STR-HANDLER*. It is called if the entity in question turns out to be a pre-declared entity

The result is the one returned by *content-handler* or *str-handler*.

Faults detected:

```
WFC: XML-Spec.html#wf-entdeclared
WFC: XML-Spec.html#norecursion
```

`ssax:read-attributes` *port entities* [Function]

{`sxml.ssax`} This procedure reads and parses a production `Attribute*`

```
[41] Attribute ::= Name Eq AttValue
[10] AttValue ::=  '"' ([^<&"] | Reference)* '"'
                | "'" ([^<&'] | Reference)* "'"
[25] Eq ::= S? '=' S?
```

The procedure returns an *ATTLIST*, of *Name* (as *UNRES-NAME*), *Value* (as string) pairs. The current character on the *port* is a non-whitespace character that is not an ncname-starting character.

Note the following rules to keep in mind when reading an 'AttValue' "Before the value of an attribute is passed to the application or checked for validity, the XML processor must normalize it as follows:

- a character reference is processed by appending the referenced character to the attribute value
- an entity reference is processed by recursively processing the replacement text of the entity [see *ENTITIES*] [named entities amp lt gt quot apos are assumed pre-declared]
- a whitespace character (`#x20`, `#xD`, `#xA`, `#x9`) is processed by appending `#x20` to the normalized value, except that only a single `#x20` is appended for a `"#xD#xA"` sequence that is part of an external parsed entity or the literal entity value of an internal parsed entity
- other characters are processed by appending them to the normalized value "

Faults detected:

```
WFC: XML-Spec.html#CleanAttrVals
WFC: XML-Spec.html#uniqattspec
```

`ssax:resolve-name` *port unres-name namespaces apply-default-ns?* [Function]

{`sxml.ssax`} Convert an *unres-name* to a *res-name* given the appropriate *namespaces* declarations. The last parameter *apply-default-ns?* determines if the default namespace applies (for instance, it does not for attribute names)

Per `REC-xml-names/#nsc-NSDeclared`, "xml" prefix is considered pre-declared and bound to the namespace name "`http://www.w3.org/XML/1998/namespace`".

This procedure tests for the namespace constraints: `http://www.w3.org/TR/REC-xml-names/#nsc-NSDeclared`.

`ssax:uri-string->symbol` *uri-str* [Function]

{`sxml.ssax`} Convert a *uri-str* to an appropriate symbol.

`ssax:complete-start-tag` *tag port elems entities namespaces* [Function]

{`sxml.ssax`} This procedure is to complete parsing of a start-tag markup. The procedure must be called after the start tag token has been read. *Tag* is an *UNRES-NAME*. *Elem s* is an instance of `xml-decl::elems`; it can be `#f` to tell the function to do *no* validation of elements and their attributes.

This procedure returns several values:

elem-gi a *RES-NAME*.

attributes element's attributes, an *ATTLIST* of (*res-name . string*) pairs. The list does *not* include `xmlns` attributes.

namespaces
the input list of namespaces amended with namespace (re-)declarations contained within the start-tag under parsing *ELEM-CONTENT-MODEL*.

On exit, the current position in *port* will be the first character after `#\>` that terminates the start-tag markup.

Faults detected:

```
VC: XML-Spec.html#enum
VC: XML-Spec.html#RequiredAttr
VC: XML-Spec.html#FixedAttr
VC: XML-Spec.html#ValueType
WFC: XML-Spec.html#uniqattspec (after namespaces prefixes are resolved)
VC: XML-Spec.html#elementvalid
WFC: REC-xml-names/#dt-NSName
```

Note, although XML Recommendation does not explicitly say it, *xmlns* and *xmlns:* attributes don't have to be declared (although they can be declared, to specify their default value).

`ssax:read-external-id` *port* [Function]

{`sxml.ssax`} This procedure parses an `ExternalID` production.

```
[75] ExternalID ::= 'SYSTEM' S SystemLiteral
                  | 'PUBLIC' S PubidLiteral S SystemLiteral
[11] SystemLiteral ::= ('"' [^"]* '"') | ("'" [^']* "'")
[12] PubidLiteral ::=  '"' PubidChar* '"' | "'" (PubidChar - "'")* "'"
[13] PubidChar ::=  #x20 | #xD | #xA | [a-zA-Z0-9]
                 | [-'()+,./:=?;!*#@$_%]
```

This procedure is supposed to be called when an `ExternalID` is expected; that is, the current character must be either `#\S` or `#\P` that start correspondingly a `SYSTEM` or `PUBLIC` token. This procedure returns the `SystemLiteral` as a string. A `PubidLiteral` is disregarded if present.

12.46.3 SSAX higher-level parsers and scanners

They parse productions corresponding to the whole (document) entity or its higher-level pieces (prolog, root element, etc).

ssax:scan-Misc *port* [Function]

{sxml.ssax} Scan the Misc production in the context:

```
[1]  document ::=  prolog element Misc*
[22] prolog ::= XMLDecl? Misc* (doctypedec l Misc*)?
[27] Misc ::= Comment | PI |  S
```

The following function should be called in the prolog or epilog contexts. In these contexts, whitespaces are completely ignored. The return value from `ssax:scan-Misc` is either a `PI`-token, a `DECL`-token, a `START` token, or EOF. Comments are ignored and not reported.

ssax:read-char-data *port expect-eof? str-handler seed* [Function]

{sxml.ssax} This procedure is to read the character content of an XML document or an XML element.

```
[43] content ::=
        (element | CharData | Reference | CDSect | PI
         | Comment)*
```

To be more precise, the procedure reads `CharData`, expands `CDSect` and character entities, and skips comments. The procedure stops at a named reference, EOF, at the beginning of a `PI` or a start/end tag.

port a port to read

expect-eof?
a boolean indicating if EOF is normal, i.e., the character data may be terminated by the EOF. EOF is normal while processing a parsed entity.

str-handler
a *STR-HANDLER*.

seed an argument passed to the first invocation of *STR-HANDLER*.

The procedure returns two results: *seed* and *token*.

The *seed* is the result of the last invocation of *str-handler*, or the original seed if *str-handler* was never called.

Token can be either an eof-object (this can happen only if *expect-eof?* was `#t`), or:

- an xml-token describing a *START* tag or an *END*-tag; For a start token, the caller has to finish reading it.
- an xml-token describing the beginning of a `PI`. It's up to an application to read or skip through the rest of this `PI`;
- an xml-token describing a named entity reference.

`CDATA` sections and character references are expanded inline and never returned. Comments are silently disregarded.

As the XML Recommendation requires, all whitespace in character data must be preserved. However, a `CR` character (`#xD`) must be disregarded if it appears before a `LF` character (`#xA`), or replaced by a `#xA` character otherwise. See Secs. 2.10 and 2.11 of the XML Recommendation. See also the canonical XML Recommendation.

ssax:assert-token *token kind gi error-cont* [Function]

{sxml.ssax} Make sure that *token* is of anticipated *kind* and has anticipated *gi*. Note *gi* argument may actually be a pair of two symbols, Namespace URI or the prefix, and of the localname. If the assertion fails, *error-cont* is evaluated by passing it three arguments: *token kind gi*. The result of *error-cont* is returned.

12.46.4 SSAX Highest-level parsers - XML to SXML

These parsers are a set of syntactic forms to instantiate a SSAX parser. A user can instantiate the parser to do the full validation, or no validation, or any particular validation. The user specifies which `PI` he wants to be notified about. The user tells what to do with the parsed character and element data. The latter handlers determine if the parsing follows a SAX or a DOM model.

`ssax:make-pi-parser` *my-pi-handlers* [Macro]

{`sxml.ssax`} Create a parser to parse and process one Processing Element (`PI`).

My-pi-handlers: An assoc list of pairs (*PI-TAG* . *PI-HANDLER*) where *PI-TAG* is an *NCName* symbol, the `PI` target, and *PI-HANDLER* is a procedure *port pi-tag seed* where *port* points to the first symbol after the `PI` target. The handler should read the rest of the `PI` up to and including the combination '`?>`' that terminates the `PI`. The handler should return a new seed. One of the *PI-TAG*s may be a symbol `*DEFAULT*`. The corresponding handler will handle `PI`s that no other handler will. If the `*DEFAULT*` *PI-TAG* is not specified, `ssax:make-pi-parser` will make one, which skips the body of the `PI`.

The output of the `ssax:make-pi-parser` is a procedure *port pi-tag seed*, that will parse the current `PI` accoding to user-specified handlers.

`ssax:make-elem-parser` *my-new-level-seed my-finish-element my-char-data-handler my-pi-handlers* [Macro]

{`sxml.ssax`} Create a parser to parse and process one element, including its character content or children elements. The parser is typically applied to the root element of a document.

my-new-level-seed
: procedure *elem-gi attributes namespaces expected-content seed*
where *elem-gi* is a *RES-NAME* of the element about to be processed. This procedure is to generate the seed to be passed to handlers that process the content of the element.

my-finish-element
: procedure *elem-gi attributes namespaces parent-seed seed*
This procedure is called when parsing of *elem-gi* is finished. The *seed* is the result from the last content parser (or from *my-new-level-seed* if the element has the empty content). *Parent-seed* is the same seed as was passed to *my-new-level-seed*. The procedure is to generate a seed that will be the result of the element parser.

my-char-data-handler
: A *STR-HANDLER*.

my-pi-handlers
: See `ssax:make-pi-handler` above.

The generated parser is a: procedure *start-tag-head port elems entities namespaces preserve-ws? seed*.
The procedure must be called after the start tag token has been read. *Start-tag-head* is an *UNRES-NAME* from the start-element tag. *Elems* is an instance of `xml-decl::elems`. See `ssax:complete-start-tag::preserve-ws?`

Faults detected:

```
VC: XML-Spec.html#elementvalid
WFC: XML-Spec.html#GIMatch
```

`ssax:make-parser` *user-handler-tag user-handler-proc ...* [Macro]

{`sxml.ssax`} Create an XML parser, an instance of the XML parsing framework. This will be a SAX, a DOM, or a specialized parser depending on the supplied user-handlers.

user-handler-tag is a symbol that identifies a procedural expression that follows the tag. Given below are tags and signatures of the corresponding procedures. Not all tags have to be specified. If some are omitted, reasonable defaults will apply.

`tag:` *`DOCTYPE`*

handler-procedure: *port docname systemid internal-subset? seed*

If *internal-subset?* is `#t`, the current position in the port is right after we have read `#\[` that begins the internal DTD subset. We must finish reading of this subset before we return (or must call skip-internal-subset if we aren't interested in reading it). The port at exit must be at the first symbol after the whole DOCTYPE declaration.

The handler-procedure must generate four values:
elems entities namespaces seed
See `xml-decl::elems` for *elems*. It may be `#f` to switch off the validation. *namespaces* will typically contain *USER-PREFIX*es for selected *URI-SYMB*s. The default handler-procedure skips the internal subset, if any, and returns `(values #f '() '() seed)`.

`tag:` *`UNDECL-ROOT`*

handler-procedure: *elem-gi seed*
where *elem-gi* is an *UNRES-NAME* of the root element. This procedure is called when an XML document under parsing contains *no* `DOCTYPE` declaration. The handler-procedure, as a DOCTYPE handler procedure above, must generate four values:
elems entities namespaces seed
The default handler-procedure returns `(values #f '() '() seed)`.

`tag:` *`DECL-ROOT`*

handler-procedure: *elem-gi seed*
where *elem-gi* is an *UNRES-NAME* of the root element. This procedure is called when an XML document under parsing does contains the `DOCTYPE` declaration. The handler-procedure must generate a new `seed` (and verify that the name of the root element matches the doctype, if the handler so wishes). The default handler-procedure is the identity function.

`tag:` *`NEW-LEVEL-SEED`*

handler-procedure: see `ssax:make-elem-parser`, *my-new-level-seed*

`tag:` *`FINISH-ELEMENT`*

handler-procedure: see `ssax:make-elem-parser`, *my-finish-element*

`tag:` *`CHAR-DATA-HANDLER`*

handler-procedure: see `ssax:make-elem-parser`, *my-char-data-handler*

`tag:` *`PI`* handler-procedure: see `ssax:make-pi-parser`.
The default value is `'()`.

The generated parser is a
procedure *PORT SEED*

This procedure parses the document prolog and then exits to an element parser (created by ssax:make-elem-parser) to handle the rest.

```
[1]  document ::=  prolog element Misc*
```

```
[22] prolog ::= XMLDecl? Misc* (doctypedec | Misc*)?
[27] Misc ::= Comment | PI |  S

[28] doctypedecl ::=  '<!DOCTYPE' S Name (S ExternalID)? S?
                       ('[' (markupdecl | PEReference | S)* ']' S?)? '>'
[29] markupdecl ::= elementdecl | AttlistDecl
                     | EntityDecl
                     | NotationDecl | PI
                     | Comment
```

A few utility procedures that turned out useful.

`ssax:reverse-collect-str` *fragments* [Function]

{`sxml.ssax`} given the list of *fragments* (some of which are text strings) reverse the list and concatenate adjacent text strings.

`ssax:reverse-collect-str-drop-ws` *fragments* [Function]

{`sxml.ssax`} given the list of fragments (some of which are text strings) reverse the list and concatenate adjacent text strings. We also drop "unsignificant" whitespace, that is, whitespace in front, behind and between elements. The whitespace that is included in character data is not affected. We use this procedure to "intelligently" drop "insignificant" whitespace in the parsed SXML. If the strict compliance with the XML Recommendation regarding the whitespace is desired, please use the `ssax:reverse-collect-str` procedure instead.

`ssax:xml->sxml` *port namespace-prefix-assig* [Function]

{`sxml.ssax`} This is an instance of a SSAX parser above that returns an SXML representation of the XML document to be read from *port*. *Namespace-prefix-assig* is a list of `(`*`USER-PREFIX . URI-STRING`*`)` that assigns *USER-PREFIX*es to certain namespaces identified by particular *URI-STRING*s. It may be an empty list. The procedure returns an SXML tree. The port points out to the first character after the root element.

12.47 `sxml.sxpath` - SXML query language

`sxml.sxpath` [Module]

SXPath is a query language for SXML, an instance of XML Information set (Infoset) in the form of s-expressions.

It is originally written by Oleg Kiselyov, and improved by Dmitry Lizorkin and Kirill Lisovsky. This module also incorporates various procedures written for SXPath by Dmitry Lizorkin and Kirill Lisovsky.

Current version is based on sxpathlib.scm,v 3.915, sxpath.scm,v 1.1, and sxpath-ext.scm,v 1.911.

This manual is mostly derived from the comments in the original source files.

The module consists of three layers.

1. Basic converters and applicators, which provides the means to access and translate SXML tree.
2. High-level query language compiler, which takes abbreviated SXPath and returns a Scheme function that selects a nodeset that satisfies the specified path from the given nodeset.
3. Extension libraries, which implements SXML counterparts to W3C XPath Core Functions Library.

12.47.1 SXPath basic converters and applicators

A converter is a function

```
type Converter = Node|Nodeset -> Nodeset
```

A converter can also play a role of a predicate: in that case, if a converter, applied to a node or a nodeset, yields a non-empty nodeset, the converter-predicate is deemed satisfied. Throughout this file a nil nodeset is equivalent to `#f` in denoting a failure.

`nodeset?` *x* [Function]
: {`sxml.sxpath`} Returns `#t` if given object is a nodeset.

`as-nodeset` *x* [Function]
: {`sxml.sxpath`} If *x* is a nodeset - returns it as is, otherwise wrap it in a list.

`sxml:element?` *obj* [Function]
: {`sxml.sxpath`} Predicate which returns `#t` if *obj* is SXML element, otherwise returns `#f`.

`ntype-names??` *crit* [Function]
: {`sxml.sxpath`} The function `ntype-names??` takes a list of acceptable node names as a criterion and returns a function, which, when applied to a node, will return `#t` if the node name is present in criterion list and `#f` otherwise.

```
ntype-names?? :: ListOfNames -> Node -> Boolean
```

`ntype??` *crit* [Function]
: {`sxml.sxpath`} The function `ntype??` takes a type criterion and returns a function, which, when applied to a node, will tell if the node satisfies the test.

```
ntype?? :: Crit -> Node -> Boolean
```

The criterion *crit* is one of the following symbols:

`id` tests if the Node has the right name (id)

`@` tests if the Node is an *attributes-list*.

`*` tests if the Node is an *Element*.

`*text*` tests if the Node is a text node.

`*data*` tests if the Node is a data node (text, number, boolean, etc., but not pair).

`*PI*` tests if the Node is a `PI` node.

`*COMMENT*`
tests if the Node is a `COMMENT` node.

`*ENTITY*` tests if the Node is a `ENTITY` node.

`*any*` `#t` for any type of Node.

`ntype-namespace-id??` *ns-id* [Function]
: {`sxml.sxpath`} This function takes a namespace-id, and returns a predicate `Node -> Boolean`, which is `#t` for nodes with this very namespace-id. *ns-id* is a string. `(ntype-namespace-id?? #f)` will be `#t` for nodes with non-qualified names.

`sxml:invert` *pred* [Function]
: {`sxml.sxpath`} This function takes a predicate and returns it inverted . That is if the given predicate yields `#f` or '`()` the inverted one yields the given node (`#t`) and vice versa.

`node-eq?` *other* [Function]
`node-equal?` *other* [Function]

{`sxml.sxpath`} Curried equivalence converter-predicates, i.e.

```
((node-eq? a) b)    ≡ (eq? a b)
((node-equal? a) b) ≡ (equal? a b)
```

`node-pos` *n* [Function]

{`sxml.sxpath`}

```
node-pos:: N -> Nodeset -> Nodeset, or
node-pos:: N -> Converter
```

Select the N'th element of a Nodeset and return as a singular Nodeset; Return an empty nodeset if the Nth element does not exist. `((node-pos 1) Nodeset)` selects the node at the head of the Nodeset, if exists; `((node-pos 2) Nodeset)` selects the Node after that, if exists. N can also be a negative number: in that case the node is picked from the tail of the list. `((node-pos -1) Nodeset)` selects the last node of a non-empty nodeset; `((node-pos -2) Nodeset)` selects the last but one node, if exists.

`sxml:filter` *pred?* [Function]

{`sxml.sxpath`}

```
filter:: Converter -> Converter
```

A filter applicator, which introduces a filtering context. The argument converter is considered a predicate, with either `#f` or nil result meaning failure.

`take-until` *pred?* [Function]

{`sxml.sxpath`}

```
take-until:: Converter -> Converter, or
take-until:: Pred -> Node|Nodeset -> Nodeset
```

Given a converter-predicate and a nodeset, apply the predicate to each element of the nodeset, until the predicate yields anything but `#f` or nil. Return the elements of the input nodeset that have been processed till that moment (that is, which fail the predicate). `take-until` is a variation of the filter above: `take-until` passes elements of an ordered input set till (but not including) the first element that satisfies the predicate. The nodeset returned by `((take-until (not pred)) nset)` is a subset – to be more precise, a prefix – of the nodeset returned by `((filter pred) nset)`.

`take-after` *pred?* [Function]

{`sxml.sxpath`}

```
take-after:: Converter -> Converter, or
take-after:: Pred -> Node|Nodeset -> Nodeset
```

Given a converter-predicate and a nodeset, apply the predicate to each element of the nodeset, until the predicate yields anything but `#f` or nil. Return the elements of the input nodeset that have not been processed: that is, return the elements of the input nodeset that follow the first element that satisfied the predicate. `take-after` along with `take-until` partition an input nodeset into three parts: the first element that satisfies a predicate, all preceding elements and all following elements.

`map-union` *proc lst* [Function]

{`sxml.sxpath`} Apply proc to each element of lst and return the list of results. If proc returns a nodeset, splice it into the result.

From another point of view, `map-union` is a function Converter->Converter, which places an argument-converter in a joining context.

node-reverse *node-or-nodeset* [Function]

{sxml.sxpath}

```
node-reverse :: Converter, or
node-reverse:: Node|Nodeset -> Nodeset
```

Reverses the order of nodes in the nodeset. This basic converter is needed to implement a reverse document order (see the XPath Recommendation).

node-trace *title* [Function]

{sxml.sxpath}

```
node-trace:: String -> Converter
```

`(node-trace title)` is an identity converter. In addition it prints out a node or nodeset it is applied to, prefixed with the 'title'. This converter is very useful for debugging.

What follow are Converter combinators, higher-order functions that transmogrify a converter or glue a sequence of converters into a single, non-trivial converter. The goal is to arrive at converters that correspond to XPath location paths.

From a different point of view, a combinator is a fixed, named *pattern* of applying converters. Given below is a complete set of such patterns that together implement XPath location path specification. As it turns out, all these combinators can be built from a small number of basic blocks: regular functional composition, map-union and filter applicators, and the nodeset union.

select-kids *test-pred?* [Function]

{sxml.sxpath}

```
select-kids:: Pred -> Node -> Nodeset
```

Given a Node, return an (ordered) subset its children that satisfy the Pred (a converter, actually).

```
select-kids:: Pred -> Nodeset -> Nodeset
```

The same as above, but select among children of all the nodes in the Nodeset.

node-self *pred* [Function]

{sxml.sxpath}

```
node-self:: Pred -> Node -> Nodeset, or
node-self:: Converter -> Converter
```

Similar to select-kids but apply to the Node itself rather than to its children. The resulting Nodeset will contain either one component, or will be empty (if the Node failed the Pred).

node-join *. selectors* [Function]

{sxml.sxpath}

```
node-join:: [LocPath] -> Node|Nodeset -> Nodeset, or
node-join:: [Converter] -> Converter
```

join the sequence of location steps or paths as described in the title comments above.

node-reduce *. converters* [Function]

{sxml.sxpath}

```
node-reduce:: [LocPath] -> Node|Nodeset -> Nodeset, or
node-reduce:: [Converter] -> Converter
```

A regular functional composition of converters. From a different point of view, `((apply node-reduce converters) nodeset)` is equivalent to `(foldl apply nodeset converters)` i.e., folding, or reducing, a list of converters with the nodeset as a seed.

node-or . *converters* [Function]

{sxml.sxpath}

```
node-or:: [Converter] -> Converter
```

This combinator applies all converters to a given node and produces the union of their results. This combinator corresponds to a union, '|' operation for XPath location paths.

node-closure *test-pred?* [Function]

{sxml.sxpath}

```
node-closure:: Converter -> Converter
```

Select all *descendants* of a node that satisfy a converter-predicate. This combinator is similar to `select-kids` but applies to grand... children as well. This combinator implements the `"descendant::"` XPath axis. Conceptually, this combinator can be expressed as

```
(define (node-closure f)
     (node-or
       (select-kids f)
(node-reduce (select-kids (ntype?? '*)) (node-closure f))))
```

This definition, as written, looks somewhat like a fixpoint, and it will run forever. It is obvious however that sooner or later `(select-kids (ntype?? '*))` will return an empty nodeset. At this point further iterations will no longer affect the result and can be stopped.

12.47.2 SXPath query language

sxpath *abbrpath . ns-binding* [Function]

{sxml.sxpath} Evaluates an abbreviated SXPath. Returns a procedure that when applied on a node or nodeset will return a nodeset matching the given path.

```
sxpath:: AbbrPath -> Converter, or
sxpath:: AbbrPath -> Node|Nodeset -> Nodeset
```

AbbrPath is a list or a string. If it is a list, it is translated to the full SXPath according to the following rewriting rules. More informal explanation follows shortly. If it is a string, it is an XPath query.

Note that these are abstract rules to show how it works, and not the running code examples. The nonterminals *sxpath1* and *sxpathr* don't exist as APIs. The term *txpath* is an internal function that interprets XPath query given as a string.

```
(sxpath '()) -> (node-join)
(sxpath '(path-component ...)) ->
              (node-join (sxpath1 path-component) (sxpath '(...)))
(sxpath1 '//) -> (node-or
                   (node-self (ntype?? '*any*))
                   (node-closure (ntype?? '*any*)))
(sxpath1 '(equal? x)) -> (select-kids (node-equal? x))
(sxpath1 '(eq? x))    -> (select-kids (node-eq? x))
(sxpath1 '(or@ ...))  -> (select-kids (ntype-names??
                                        (cdr '(or@ ...))))
(sxpath1 '(not@ ...)) -> (select-kids (sxml:invert
                                       (ntype-names??
                                        (cdr '(not@ ...)))))
(sxpath1 '(ns-id:* x)) -> (select-kids
                                     (ntype-namespace-id?? x))
(sxpath1 ?symbol)     -> (select-kids (ntype?? ?symbol))
(sxpath1 ?string)     -> (txpath ?string)
```

```
(sxpath1 procedure)   -> procedure
(sxpath1 '(?symbol ...)) -> (sxpath1 '((?symbol) ...))
(sxpath1 '(path reducer ...)) ->
               (node-reduce (sxpath path) (sxpathr reducer) ...)
(sxpathr number)      -> (node-pos number)
(sxpathr path-filter) -> (filter (sxpath path-filter))
```

SXPath in its simplest form is a list of path components. The result procedure will follow the same path and return the matching node list. For example `(one two three)` will find element `one` then `two` inside it and `three` inside element `two`. The equivalent XPath would be `one/two/three`.

There are a few special path components (see `ntype??` for the complete list):

`*` matches an element node.

`//` matches any one or many consecutive path components.

`*text*` matches a text node (`text()` in XPath).

`*data*` matches any data node (e.g. text, number, boolean, etc., but not pair).

`@` selects the attribute list node.

A path component could be a list in one of these forms:

`(equal? x)`
: matches if the node under examination matches *x* using `node-equal?`

`(eq? x)`
: matches if the node under examination matches *x* using `node-eq?`

`(or@ ...)`
: matches if the element name is one of the specified symbols.

`(not@ ...)`
: matches if the element name is not one of the specified symbols.

`(ns-id:* x)`
: matches the node if it's with namespace *x*

`(<path> n)`
: matches the *n*-th node matching same path component. *n* starts from 1. Negative numbers start from the end of the node list backward. This is `path[n]` syntax in XPath.

`(<path> (<predicate>...))`
: matches a path component *path* and `(sxpath (<predicate>...))` on those nodes are not empty. This is `path[predicate...]` syntax in XPath.

If the path component is a string, it is interpreted as an XPath query string.

If the path component is a procedure, the procedure takes three arguments: the nodeset being examined, the root node and the variable bindings.

The root node is usually the entire sxml being applied. However if you apply the result sxpath procedure with two arguments, root-node will be the second argument.

When applied with three arguments, the variable bindings are the third one. This lets you pass arguments to the procedure.

```
;; select all <book> elements whose style attribute value is equal to
;; the <bookstore> element's specialty attribute value.
(sxpath "//book[/bookstore/@specialty=@style]")
;; a similar query but this time make sure specialty of _all_
;; bookstores is matched
```

```
(let ([match-specialty (lambda (node root var-binding)
                         (let ([style (car ((sxpath '(@ style *text*)) node))]
                               [all-specialty ((sxpath '(bookstore @ specialty *text
                           (fold (lambda (specialty last-result)
                                   (and last-result (string=? style specialty)))
                                 #t
                                 all-specialty)))])
  (sxpath `(// (book (,match-specialty)))))

;; select all <bookstore> elements that are inside top-level <book>
;; element
(sxpath '(book bookstore))
;; select all <bookstore> elements from anywhere
(sxpath '(// bookstore))
;; select attribute "name" in the top-level <book> element
(sxpath '(book @ name))
;; select all <bookstore> and <bookshop> elements that are inside
;; top-level <book> element
(sxpath '(book (or@ bookstore bookshop)))
;; select all elements except <movie> that are inside top-level <book>
;; element
(sxpath '(book (not@ movie @)))
;; select the attribute "name" of the second <bookstore> element
(sxpath '(book (bookstore 2) @ name))
;; select the attribute "name" of all <bookstore> elements that has
;; attribute "recommended"
(sxpath '(book (bookstore (@ recommended)) @ name))
;; select the attribute "name" of all <bookstore> elements whose
;; "rating" attribute is 3
(sxpath '(book (bookstore (@ rating (eq? 3))) @ name))
;; select the attribute "rating" whose value is greater than 3 from
;; all <bookstore> elements
(let ([greater (lambda (nodeset root-node var-binding)
                 (filter (lambda (node)
                           (> (string->number (sxml:string-value node))
                              3))
                         nodeset))])
  (sxpath `(book bookstore @ rating ,greater)))
```

Some wrapper functions around `sxpath`:

`if-sxpath` *path* [Function]

{`sxml.sxpath`} `sxpath` always returns a list, which is `#t` in Scheme. `if-sxpath` returns `#f` instead of empty list.

`if-car-sxpath` *path* [Function]

{`sxml.sxpath`} Returns first node found, if any. Otherwise returns `#f`.

`car-sxpath` *path* [Function]

{`sxml.sxpath`} Returns first node found, if any. Otherwise returns empty list.

`sxml:id-alist` *node . lpaths* [Function]

{`sxml.sxpath`} Built an index as a list of `(ID_value . element)` pairs for given node. *lpaths* are location paths for attributes of type ID.

12.47.3 SXPath extension

SXML counterparts to W3C XPath Core Functions Library.

`sxml:string` *object* [Function]

{`sxml.sxpath`} The counterpart to XPath `string` function (section 4.2 XPath Rec.) Converts a given object to a string. NOTE:

1. When converting a nodeset - a document order is not preserved
2. *number->string* function returns the result in a form which is slightly different from XPath Rec. specification

`sxml:boolean` *object* [Function]

{`sxml.sxpath`} The counterpart to XPath `boolean` function (section 4.3 XPath Rec.) Converts its argument to a boolean.

`sxml:number` *obj* [Function]

{`sxml.sxpath`} The counterpart to XPath `number` function (section 4.4 XPath Rec.) Converts its argument to a number NOTE:

1. The argument is not optional (yet?).
2. `string->number` conversion is not IEEE 754 round-to-nearest.
3. NaN is represented as 0.

`sxml:string-value` *node* [Function]

{`sxml.sxpath`} Returns a string value for a given node in accordance to XPath Rec. 5.1 - 5.7

`sxml:node?` *node* [Function]

{`sxml.sxpath`} According to XPath specification 2.3, this test is true for any XPath node. For SXML auxiliary lists and lists of attributes has to be excluded.

`sxml:attr-list` *obj* [Function]

{`sxml.sxpath`} Returns the list of attributes for a given SXML node. Empty list is returned if the given node is not an element, or if it has no list of attributes

`sxml:id` *id-index* [Function]

{`sxml.sxpath`} Select SXML element by its unique IDs. (XPath Rec. 4.1) Returns a converter that takes *object*, which is a nodeset or a datatype which can be converted to a string by means of a '`string`' function.

id-index is `( (id-value . element) (id-value . element) ... )`.

This index is used for selection of an element by its unique ID.

Comparators for XPath objects:

`sxml:equality-cmp` *bool-op number-op string-op* [Function]

{`sxml.sxpath`} A helper for XPath equality operations: `=` , `!=` *bool-op*, *number-op* and *'string-op* are comparison operations for a pair of booleans, numbers and strings respectively.

`sxml:equal?` *a b* [Function]

`sxml:not-equal?` *a b* [Function]

{`sxml.sxpath`} Counterparts of XPath equality operations: `=` , `!=`, using default equality tests.

`sxml:relational-cmp` *op* [Function]

{`sxml.sxpath`} Creates a relational operation (`<` , `>` , `<=` , `>=`) for two XPath objects. `op` is comparison procedure: `<` , `>` , `<=` or `>=`.

XPath axises. An order in resulting nodeset is preserved.

`sxml:attribute` *test-pred?* [Function]
{`sxml.sxpath`} Attribute axis.

`sxml:child` *test-pred?* [Function]
{`sxml.sxpath`} Child axis. This function is similar to '`select-kids`', but it returns an empty child-list for PI, Comment and Entity nodes.

`sxml:parent` *test-pred?* [Function]
{`sxml.sxpath`} Parent axis.

Given a predicate, it returns a function `RootNode -> Converter` which yields a `node -> parent` converter then applied to a rootnode.

Thus, such a converter may be constructed using `((sxml:parent test-pred) rootnode)` and returns a parent of a node it is applied to. If applied to a nodeset, it returns the list of parents of nodes in the nodeset. The rootnode does not have to be the root node of the whole SXML tree – it may be a root node of a branch of interest. The `parent::` axis can be used with any SXML node.

`sxml:ancestor` *test-pred?* [Function]
{`sxml.sxpath`} Ancestor axis

`sxml:ancestor-or-self` *test-pred?* [Function]
{`sxml.sxpath`} Ancestor-or-self axis

`sxml:descendant` *test-pred?* [Function]
{`sxml.sxpath`} Descendant axis

`sxml:descendant-or-self` *test-pred?* [Function]
{`sxml.sxpath`} Descendant-or-self axis

`sxml:following` *test-pred?* [Function]
{`sxml.sxpath`} Following axis

`sxml:following-sibling` *test-pred?* [Function]
{`sxml.sxpath`} Following-sibling axis

`sxml:namespace` *test-pred?* [Function]
{`sxml.sxpath`} Namespace axis

`sxml:preceding` *test-pred?* [Function]
{`sxml.sxpath`} Preceding axis

`sxml:preceding-sibling` *test-pred?* [Function]
{`sxml.sxpath`} Preceding-sibling axis

Popular shortcuts:

`sxml:child-nodes` *nodeset* [Function]
{`sxml.sxpath`}

```
((sxml:child sxml:node?) nodeset)
```

`sxml:child-elements` *nodeset* [Function]
{`sxml.sxpath`}

```
((select-kids sxml:element?) nodeset)
```

12.48 `sxml.tools` - Manipulating SXML structure

`sxml.tools` [Module]

This module is a port of Kirill Lisofsky's sxml-tools, a collection of convenient procedures that work on SXML structure. The current version is derived from sxml-tools CVS revision 3.13.

The manual entry is mainly derived from the comments in the original source code.

12.48.1 SXML predicates

`sxml:empty-element?` *obj* [Function]

{`sxml.tools`} A predicate which returns `#t` if given element *obj* is empty. Empty element has no nested elements, text nodes, `PI`s, Comments or entities but it may contain attributes or namespace-id. It is a SXML counterpart of XML `empty-element`.

`sxml:shallow-normalized?` *obj* [Function]

{`sxml.tools`} Returns `#t` if the given *obj* is shallow-normalized SXML element. The element itself has to be normalized but its nested elements are not tested.

`sxml:normalized?` *obj* [Function]

{`sxml.tools`} Returns `#t` if the given *obj* is normalized SXML element. The element itself and all its nested elements have to be normalised.

`sxml:shallow-minimized?` *obj* [Function]

{`sxml.tools`} Returns `#t` if the given *obj* is shallow-minimized SXML element. The element itself has to be minimised but its nested elements are not tested.

`sxml:minimized?` *obj* [Function]

{`sxml.tools`} Returns `#t` if the given *obj* is minimized SXML element. The element itself and all its nested elements have to be minimised.

12.48.2 SXML accessors

`sxml:name` *obj* [Function]

{`sxml.tools`} Returns a name of a given SXML node. It's just an alias of `car`, but introduced for the sake of encapsulation.

`sxml:element-name` *obj* [Function]

{`sxml.tools`} A version of `sxml:name`, which returns `#f` if the given *obj* is not a SXML element. Otherwise returns its name.

`sxml:node-name` *obj* [Function]

{`sxml.tools`} Safe version of `sxml:name`, which returns `#f` if the given *obj* is not a SXML node. Otherwise returns its name.

`sxml:ncname` *obj* [Function]

{`sxml.tools`} Returns Local Part of Qualified Name (Namespaces in XML production [6]) for given obj, which is `":"`-separated suffix of its Qualified Name. If a name of a node given is `NCName` (Namespaces in XML production [4]), then it is returned as is. Please note that while SXML name is a symbol this function returns a string.

`sxml:name->ns-id` *sxml-name* [Function]

{`sxml.tools`} Returns namespace-id part of given name, or `#f` if it's LocalName `sxml:content` *obj* [Function]

{`sxml.tools`} Returns the content of given SXML element or nodeset (just text and element nodes) representing it as a list of strings and nested elements in document order. This list is empty if *obj* is empty element or empty list.

`sxml:content-raw` *obj* [Function]

{`sxml.tools`} Returns all the content of normalized SXML element except *attr-list* and *aux-list*. Thus it includes `PI`, `COMMENT` and `ENTITY` nodes as well as `TEXT` and `ELEMENT` nodes returned by `sxml:content`. Returns a list of nodes in document order or empty list if *obj* is empty element or empty list. This function is faster than `sxml:content`.

In SXML normal form, an element is represented by a list as this:

```
(name attr-list aux-list content ...)
```

where *attr-list* is a list beginning with `@`, and *aux-list* is a list beginning with `@@`.

In the minimized form, *Aux-list* can be omitted when it is empty. *Attr-list* can be omitted when it is empty *and aux-list* is absent.

The following procedures extract *attr-list* and *aux-list*.

`sxml:attr-list-node` *obj* [Function]

{`sxml.tools`} Returns *attr-list* for a given *obj*, or `#f` if it is absent

`sxml:attr-as-list` *obj* [Function]

{`sxml.tools`} Returns *attr-list* wrapped in list, or `'((@))` if it is absent and *aux-list* is present, or `'()` if both lists are absent.

`sxml:aux-list-node` *obj* [Function]

{`sxml.tools`} Returns *aux-list* for a given *obj*, or `#f` if it is absent.

`sxml:aux-as-list` *obj* [Function]

{`sxml.tools`} Returns *aux-list* wrapped in list, or `'()` if it is absent.

`sxml:attr-list-u` *obj* [Function]

{`sxml.tools`} Returns the list of attributes for given element or nodeset. Analog of `((sxpath '(@ *)) obj)`. Empty list is returned if there is no list of attributes.

The `-u` suffix indicates it can be used for non-normalized SXML node. ('u' stands for 'universal').

`sxml:aux-list` *obj* [Function]

{`sxml.tools`} Returns the list of auxiliary nodes for given element or nodeset. Analog of `((sxpath '(@@ *)) obj)`. Empty list is returned if a list of auxiliary nodes is absent.

`sxml:aux-list-u` *obj* [Function]

{`sxml.tools`} Returns the list of auxiliary nodes for given element or nodeset. Analog of `((sxpath '(@@ *)) obj)`. Empty list is returned if a list of auxiliary nodes is absent.

The `-u` suffix indicates it can be used for non-normalized SXML node. ('u' stands for 'universal').

`sxml:aux-node` *obj aux-name* [Function]

{`sxml.tools`} Return the first aux-node with *aux-name* given in SXML element *obj* or `#f` is such a node is absent. Note: it returns just the *first* node found even if multiple nodes are present, so it's mostly intended for nodes with unique names .

`sxml:aux-nodes` *obj aux-name* [Function]

{`sxml.tools`} Return a list of aux-node with *aux-name* given in SXML element *obj* or `'()` if such a node is absent.

`sxml:attr` *obj attr-name* [Function]
{`sxml.tools`} Accessor for an attribute *attr-name* of given SXML element *obj*. It returns: the value of the attribute if the attribute is present, or `#f` if there is no such an attribute in the given element.

`sxml:num-attr` *obj attr-name* [Function]
{`sxml.tools`} Accessor for a numerical attribute *attr-name* of given SXML element *obj*. It returns: a value of the attribute as the attribute as a number if the attribute is present and its value may be converted to number using `string->number`, or `#f` if there is no such an attribute in the given element or its value can't be converted to a number.

`sxml:attr-u` *obj attr-name* [Function]
{`sxml.tools`} Accessor for an attribute *attr-name* of given SXML element *obj* which may also be an attributes-list or nodeset (usually content of SXML element).

It returns: the value of the attribute if the attribute is present, or `#f` if there is no such an attribute in the given element.

The `-u` suffix indicates it can be used for non-normalized SXML node. ('u' stands for 'universal').

`sxml:ns-list` *obj* [Function]
{`sxml.tools`} Returns the list of namespaces for given element. Analog of `((sxpath '(@@ *NAMESPACES* *)) obj)` Empty list is returned if there is no list of namespaces.

`sxml:ns-id->nodes` *obj namespace-id* [Function]
{`sxml.tools`} Returns the list of namespace-assoc's for given *namespace-id* in SXML element *obj*. Analog of `((sxpath '(@@ *NAMESPACES* namespace-id)) obj)`. Empty list is returned if there is no namespace-assoc with *namespace-id* given.

`sxml:ns-id->uri` *obj namespace-id* [Function]
{`sxml.tools`} Returns a URI for *namespace-id* given, or `#f` if there is no namespace-assoc with *namespace-id* given.

`sxml:ns-uri->id` *obj uri* [Function]
{`sxml.tools`} Returns a namespace-id for namespace URI given.

`sxml:ns-id` *ns-assoc* [Function]
{`sxml.tools`} Returns namespace-id for given namespace-assoc list.

`sxml:ns-uri` *ns-assoc* [Function]
{`sxml.tools`} Returns URI for given namespace-assoc list.

`sxml:ns-prefix` *ns-assoc* [Function]
{`sxml.tools`} It returns namespace prefix for given namespace-assoc list. Original (as in XML document) prefix for namespace-id given has to be strored as the third element in namespace-assoc list if it is different from namespace-id. If original prefix is omitted in namespace-assoc then namespace-id is used instead.

12.48.3 SXML modifiers

Constructors and mutators for normalized SXML data. These functions are optimized for normalized SXML data. They are not applicable to arbitrary non-normalized SXML data.

Most of the functions are provided in two variants:

1. side-effect intended functions for linear update of given elements. Their names are ended with exclamation mark. Note that the returned value of this variant is unspecified, unless explicitly noted. An example: `sxml:change-content!`.

2. pure functions without side-effects which return modified elements. An example: `sxml:change-content`.

`sxml:change-content` *obj new-content* [Function]
`sxml:change-content!` *obj new-content* [Function]

{`sxml.tools`} Change the content of given SXML element to *new-content*. If *new-content* is an empty list then the *obj* is transformed to an empty element. The resulting SXML element is normalized.

`sxml:change-attrlist` *obj new-attrlist* [Function]
`sxml:change-attrlist!` *obj new-attrlist* [Function]

{`sxml.tools`} The resulting SXML element is normalized. If *new-attrlist* is empty, the cadr of *obj* is `(@)`.

`sxml:change-name` *obj new-name* [Function]
`sxml:change-name!` *obj new-name* [Function]

{`sxml.tools`} Change a name of SXML element destructively.

`sxml:add-attr` *obj attr* [Function]

{`sxml.tools`} Returns SXML element *obj* with attribute *attr* added, or `#f` if the attribute with given name already exists. *attr* is `(attr-name attr-value)`. Pure functional counterpart to `sxml:add-attr!`.

`sxml:add-attr!` *obj attr* [Function]

{`sxml.tools`} Add an attribute *attr* for an element *obj*. Returns `#f` if the attribute with given name already exists. The resulting SXML node is normalized. Linear update counterpart to `sxml:add-attr`.

`sxml:change-attr` *obj attr* [Function]

{`sxml.tools`} Returns SXML element *obj* with changed value of attribute *attr*, or `#f` if where is no attribute with given name. *attr* is `(attr-name attr-value)`.

`sxml:change-attr!` *obj attr* [Function]

{`sxml.tools`} Change value of the attribute for element *obj*. *attr* is `(attr-name attr-value)`. Returns `#f` if where is no such attribute.

`sxml:set-attr` *obj attr* [Function]
`sxml:set-attr!` *obj attr* [Function]

{`sxml.tools`} Set attribute *attr* of element *obj*. If there is no such attribute the new one is added.

`sxml:add-aux` *obj aux-node* [Function]

{`sxml.tools`} Returns SXML element *obj* with an auxiliary node *aux-node* added.

`sxml:add-aux!` *obj aux-node* [Function]

{`sxml.tools`} Add an auxiliary node *aux-node* for an element *obj*.

`sxml:squeeze` *obj* [Function]
`sxml:squeeze!` *obj* [Function]

{`sxml.tools`} Eliminates empty lists of attributes and aux-lists for given SXML element *obj* and its descendants ("minimize" it). Returns a minimized and normalized SXML element.

`sxml:clean` *obj* [Function]

{`sxml.tools`} Eliminates empty lists of attributes and all aux-lists for given SXML element *obj* and its descendants. Returns a minimized and normalized SXML element.

12.48.4 SXPath auxiliary utilities

These are convenience utilities to extend SXPath functionalities.

`sxml:add-parents` *obj . top-ptr* [Function]

{`sxml.tools`} Returns an SXML nodeset with a 'parent pointer' added. A parent pointer is an aux node of the form `(*PARENT* thunk)`, where *thunk* returns the parent element.

`sxml:node-parent` *rootnode* [Function]

{`sxml.tools`} Returns a fast 'node-parent' function, i.e. a function of one argument - SXML element - which returns its parent node using `*PARENT*` pointer in aux-list. `'*TOP-PTR*` may be used as a pointer to root node. It return an empty list when applied to root node.

`sxml:lookup` *id index* [Function]

{`sxml.tools`} Lookup an element using its ID.

12.48.5 SXML to markup conversion

Procedures to generate XML or HTML marked up text from SXML. For more advanced conversion, see the SXML serializer (Section 12.49 [Serializing XML and HTML from SXML], page 771).

`sxml:clean-feed` *. fragments* [Function]

{`sxml.tools`} Filter the 'fragments'. The fragments are a list of strings, characters, numbers, thunks, `#f` – and other fragments. The function traverses the tree depth-first, and returns a list of strings, characters and executed thunks, and ignores `#f` and `'()`.

If all the meaningful fragments are strings, then `(apply string-append ... )` to a result of this function will return its string-value.

It may be considered as a variant of Oleg Kiselyov's `SRV:send-reply`: While `SRV:send-reply` displays fragments, this function returns the list of meaningful fragments and filter out the garbage.

`sxml:attr->xml` *attr* [Function]

{`sxml.tools`} Creates the XML markup for attributes.

`sxml:string->xml` *string* [Function]

{`sxml.tools`} Return a string or a list of strings where all the occurrences of characters `<`, `>`, `&`, `"`, or `'` in a given string are replaced by corresponding character entity references. See also `sxml:string->html`.

`sxml:sxml->xml` *tree* [Function]

{`sxml.tools`} A version of dispatch-node specialized and optimized for SXML->XML transformation.

`sxml:attr->html` *attr* [Function]

{`sxml.tools`} Creates the HTML markup for attributes.

`sxml:string->html` *string* [Function]

{`sxml.tools`} Given a string, check to make sure it does not contain characters `<`, `>`, &, `"` that require encoding. See also `html-escape-string` in Section 12.55 [Simple HTML document construction], page 784.

`sxml:non-terminated-html-tag?` *tag* [Function]

{`sxml.tools`} This predicate yields `#t` for `"non-terminated"` HTML 4.0 tags.

`sxml:sxml->html` *tree* [Function]

{`sxml.tools`} A version of dispatch-node specialized and optimized for SXML->HTML transformation.

12.49 `sxml.serializer` - Serializing XML and HTML from SXML

`sxml.serializer` [Module]

This module contains a full-featured serializer from SXML into XML and HTML, partially conforming to XSLT 2.0 and XQuery 1.0 Serialization (`http://www.w3.org/TR/2005/CR-xslt-xquery-serialization-20051103/`). It's more powerful than sxml:sxml->xml and sxml:sxml->html from sxml.tools.

The manual entry is mainly derived from the comments in the original source code.

12.49.1 Simple SXML serializing

The SXML serializer provides some convenient high-level converters which should be enough for most tasks.

`srl:sxml->xml` *sxml-obj :optional port-or-filename* [Function]

{`sxml.serializer`} Serializes the *sxml-obj* into XML, with indentation to facilitate readability by a human.

If *port-or-filename* is not supplied, the functions return a string that contains the serialized representation of the *sxml-obj*.

If *port-or-filename* is supplied and is a port, the functions write the serialized representation of *sxml-obj* to this port and return an unspecified result.

If *port-or-filename* is supplied and is a string, this string is treated as an output filename, the serialized representation of *sxml-obj* is written to that filename and an unspecified result is returned. If a file with the given name already exists, the effect is unspecified.

`srl:sxml->xml-noindent` *sxml-obj :optional port-or-filename* [Function]

{`sxml.serializer`} Serializes the *sxml-obj* into XML, without indentation.

Argument *port-or-filename* works like described in `srl:sxml->xml`.

`srl:sxml->html` *sxml-obj :optional port-or-filename* [Function]

{`sxml.serializer`} Serializes the *sxml-obj* into HTML, with indentation to facilitate readability by a human.

Argument *port-or-filename* works like described in `srl:sxml->xml`.

`srl:sxml->html-noindent` *sxml-obj :optional port-or-filename* [Function]

{`sxml.serializer`} Serializes the *sxml-obj* into HTML, without indentation.

Argument *port-or-filename* works like described in `srl:sxml->xml`.

12.49.2 Custom SXML serializing

These functions provide full access to all configuration parameters of the XML serializer.

`srl:parameterizable` *sxml-obj :optional port-or-filename params** [Function]

{`sxml.serializer`} Generalized serialization procedure, parameterizable with all the serialization parameters supported by this implementation.

sxml-obj - an SXML object to serialize

port-or-filename - either `#f`, a port or a string; works like in srl:sxml->xml (Section 12.49.1 [Simple SXML serializing], page 771).

params - each parameter is a cons of param-name (a symbol) and param-value. The available parameter names and their values are described below:

`method` - Either the symbol `xml` or `html`. For a detailed explanation of the difference between XML and HTML methods, see XSLT 2.0 and XQuery 1.0 Serialization (`http://www.w3.org/TR/2005/CR-xslt-xquery-serialization-20051103/`).

`indent` - Whether the output XML should include whitespace for human readability (`#t` or `#f`). You can also supply a string, which will be used as the indentation unit.

`omit-xml-declaration?` - Whether the XML declaration should be omitted. Default: `#t`.

`standalone` - Whether to define the XML document as standalone in the XML declaration. Should be one of the symbols `yes`, `no` or `omit`, the later causing standalone declaration to be suppressed. Default: `omit`.

`version` - The XML version used in the declaration. A string or a number. Default: `"1.0"`.

`cdata-section-elements` - A list of SXML element names (as symbols). The contents of those elements will be escaped as CDATA sections.

`ns-prefix-assig` - A list of `(cons prefix namespace-uri)`, where each `prefix` is a symbol and each `namespace-uri` a string. Will serialize the given namespaces with the corresponding prefixes.

ATTENTION: If a parameter name is unexpected or a parameter value is ill-formed, the parameter is silently ignored!

Example usage:

```
(srl:parameterizable
  '(tag (@ (attr "value")) (nested "text node") (empty))
  (current-output-port)
  '(method . xml)  ; XML output method is used by default
  '(indent . "\t")  ; use a single tabulation to indent
  '(omit-xml-declaration . #f)  ; add XML declaration
  '(standalone . yes)  ; denote a standalone XML document
  '(version . "1.0"))  ; XML version
```

```
param ::= (cons param-name param-value)
param-name ::= symbol

cdata-section-elements
value ::= (listof sxml-elem-name)
sxml-elem-name ::= symbol

indent
value ::= 'yes | #t | 'no | #f | whitespace-string

method
value ::= 'xml | 'html

ns-prefix-assig
value ::= (listof (cons prefix namespace-uri))
prefix ::= symbol
namespace-uri ::= string

omit-xml-declaration?
value ::= 'yes | #t | 'no | #f

standalone
value ::= 'yes | #t | 'no | #f | 'omit

version
value ::= string | number
```

`srl:sxml->string` *sxml-obj cdata-section-elements indent method ns-prefix-assig omit-xml-declaration? standalone version* [Function]

{`sxml.serializer`} Same as `srl:parameterizable` returning a string and without the overhead of parsing parameters. This function interface may change in future versions of the library.

`srl:display-sxml` *sxml->obj port-or-filename cdata-section-elements indent method ns-prefix-assig omit-xml-declaration? standalone version* [Function]

{`sxml.serializer`} Same as `srl:parameterizable` writing output to *port-or-filename* and without the overhead of parsing parameters. This function interface may change in future versions of the library.

12.50 `text.console` - Text terminal control

`text.console` [Module]

This module provides a simple interface for character terminal control. Currently we support vt100 compatible terminals and Windows console.

This module doesn't depend on external library such as `curses` and works with Gauche alone, but what it can do is limited; for example, you can't get an event when shift key alone is pressed. For finer controls, you need some extension libraries.

For an example of the features in this module, see `snake.scm` in the examples directory of Gauche source distribution.

Console objects

`<vt100>` [Class]

{`text.console`} Represents a vt100-compatible terminal. An instance of this class can be passed to the "console" argument of the following generic functions.

`iport` [Instance Variable of `<vt100>`]

Input port connected to the terminal. The default value is the standard input port.

`oport` [Instance Variable of `<vt100>`]

Output port connected to the terminal. The default value is the standard output port.

`input-delay` [Instance Variable of `<vt100>`]

The terminal send back special keys encoded in an input escape sequence. In order to distinguish such keys from the actual ESC key, we time the input—if the subsequent input doesn't come within `input-delay` microseconds, we interpret the input as individual keystroke, rather than a part of an escape sequence. The default value is `1000` (1ms).

`<windows-console>` [Class]

Represents Windows console. This class is defined on all platforms, but its useful methods are only available on Windows-native runtime.

It doesn't have public slots.

The application has to check the runtime to see what kind of console is available. A suggested flow is as follows.

- If `has-windows-console?` returns true, create `<windows-console>` instance. You don't need `cond-expand`; `has-windows-console?` returns `#f` on non-Windows platforms.
- Check the environment variable `TERM`. If it is set and satisfies `vt100-compatible?`, you can create `<vt100>` instance. (Note: It is possible that you end up using `<vt100>` console on Windows; e.g. `gosh` running on MSYS shell.)

- Otherwise, console isn't available.

The following procedure packages this flow.

`make-default-console` *:key if-not-available* [Function]

{`text.console`} Determines a suitable console class of the running process and returns its instance.

If no suitable console is available, the behavior depends on the *if-not-available* keyword argument. If it is `:error`, which is default, an error is signalled. If it is `#f`, the procedure returns `#f`.

`vt100-compatible?` *string* [Function]

{`text.console`} Given the string value of the environment variable `TERM`, returns `#t` if the terminal can be handled by `<vt100>` console, `#f` otherwise.

Console control

`call-with-console` *console proc :key mode* [Generic function]

{`text.console`} Takes over the control of the console, and calls *proc* with *console* as the only argument. The console is set to the *mode*, which must be a symbol `with-terminal-mode` accepts: `raw`, `rare` or `cooked`. By default the console is set to `rare` mode, which turn off the echoing and passes most of keystrokes to the program, but it intercepts terminal controls (like `Ctrl-C` for interrupt and `Ctrl-Z` for suspend; the actual key depends on terminal settings, though.)

If *proc* raises an unhandled error, this generic function resets the terminal mode before returning. It does not clear the screen.

`putch` *console char* [Generic function]

{`text.console`} Display a character at the current cursor position, and move the current cursor position.

`putstr` *console string* [Generic function]

{`text.console`} Display a string from the current cursor position, and move the current cursor position.

`beep` *console* [Generic function]

{`text.console`} Ring the beep, or flash the screen (visible bell) if possible.

`getch` *console* [Generic function]

{`text.console`} Fetch a keypress from the console. This blocks until any key is pressed.

The return value may be one of the following values:

A character
: A key for the character is pressed. It may be a control code if the control key is pressed with the key; that is, if the user presses Ctrl-A, `#\x01` will be returned.

A symbol
: Indicates a special key; the following keys are supported: `KEY_UP`, `KEY_DOWN`, `KEY_LEFT`, `KEY_RIGHT`, `KEY_HOME`, `KEY_END`, `KEY_INS`, `KEY_DEL`, `KEY_PGDN`, `KEY_PGUP`, `KEY_F1`, `KEY_F2`, `KEY_F3`, `KEY_F4`, `KEY_F5`, `KEY_F6`, `KEY_F7`, `KEY_F8`, `KEY_F9`, `KEY_F10`, `KEY_F11`, `KEY_F12`. (Note: DELETE key is usually mapped to `#\x7f`, but it depends on the terminal).

A list of symbol `ALT` and a character.
: Indicates the character key is pressed with Alt key. For example, if the user presses Alt-a, `(ALT #\a)` is returned (assuming CAPSLOCK is off).

EOF
: Indicates the input is closed somehow.

Modifier keys except `ALT` are not treated separately but included in the returned keycode. Assuming CAPSLOCK is off, if the user press `a`, `Shift+a`, and `Ctrl+a`, the returned value is `#\a`, `#\A` and `#\x01`, respectively. `Ctrl+Shift+a` can't be distinguished from `Ctrl+a`. `ALT+a`, `ALT+Shift+a`, and `ALT+Ctrl+a` will be `(ALT #\a)`, `(ALT #\A)` and `(ALT #\x01)`, respectively.

`chready?` *console* [Generic function]

{`text.console`} Returns true if there's a key sequence to be read in the console's input.

`query-cursor-position` *console* [Generic function]

{`text.console`} Returns two values, the current cursor's x and y position. The top-left corner is (0,0).

`move-cursor-to` *console row column* [Generic function]

{`text.console`} Move cursor to the specified position. The top-left corner is (0,0).

`reset-terminal` *console* [Generic function]

{`text.console`} Reset terminal. Usually this sets the character attributes to the default, clears the screen, and moves the cursor to (0, 0).

`clear-screen` *console* [Generic function]

{`text.console`} Clear entire screen.

`clear-to-eol` *console* [Generic function]

{`text.console`} Clear characters from the current cursor position to the end of the line.

`clear-to-eos` *console* [Generic function]

{`text.console`} Clear characters from the current cursor position to the end of the screen.

`hide-cursor` *console* [Generic function]
`show-cursor` *console* [Generic function]

{`text.console`} Hide/show the cursor.

`cursor-down/scroll-up` *console* [Generic function]

{`text.console`} If the cursor is at the bottom line of the screen, scroll up the contents and clear the bottom line; the cursor stays the same position. If the cursor is not at the bottom line of the screen, move the cursor down.

`cursor-up/scroll-down` *console* [Generic function]

{`text.console`} If the cursor is at the top line of the screen, scroll down the contents and clear the top line; the cursor stays the same position. If the cursor is not at the top line of the screen, move the cursor up.

`query-screen-size` *console* [Generic function]

{`text.console`} Returns two values, the width and height of the screen.

Note: This may affect what's shown in the console. It is recommended that you only call this before redrawing the entire screen and save the result.

`set-character-attribute` *console spec* [Generic function]

{`text.console`} Set the console so that the subsequent characters will be written with attributes specified by *spec*.

The character attributes spec is a list in the following format:

```
(<fgcolor> [<bgcolor> . <option> ...])
```

where:

```
<fgcolor> : <color> | #f     ; #f means default
<bgcolor> : <color> | #f
```

```
<color>  : black | red | green | yellow | blue | magenta | cyan | white
<option> : bright | reverse | underscore
```

For example, you can set characters to be written in red with black background and underscore, you can call:

```
(set-character-attribute con '(red black underscore))
```

That the options may seem rather limited in the age of full-color bitmap displays. That's what it used to be, young lads.

`reset-character-attribute` *console* [Generic function]

{`text.console`} Reset character attributes to the default.

`with-character-attribute` *console attrs thunk* [Generic function]

{`text.console`} Sets the console's attributes to *attrs* and calls *thunk*, then restores the attributes. Even if *thunk* throws an error, attributes are restored.

Note: You should be able to nest this, but currently nesting isn't working.

12.51 `text.csv` - CSV tables

`text.csv` [Module]

Provides a function to parse/generate CSV (comma separated value) tables, including the format defined in RFC4180. You can customize the separator and quoter character to deal with variations of CSV formats.

CSV table is consisted by a series of *records*, separated by a newline. Each record contains number of *fields*, separated by a separator character (by default, a comma). A field can contain comma or newline if quoted, i.e. surrounded by double-quote characters. To include double-quote character in a quoted field, use two consecutive double-quote character. Usually, the whitespaces around the field are ignored.

Since use cases of CSV-like files vary, we provide layered API to be combined flexibly.

Low-level API

The bottom layer of API is to convert text into list of lists and vice versa.

`make-csv-reader` *separator :optional* (*quote-char* `#\"`) [Function]

{`text.csv`} Returns a procedure with one optional argument, an input port. When the procedure is called, it reads one record from the port (or, if omitted, from the current input port) and returns a list of fields. If input reaches EOF, it returns EOF.

`make-csv-writer` *separator :optional newline* (*quote-char* `#\"`) *special-char-set* [Function]

{`text.csv`} Returns a procedure with two arguments, output port and a list of fields. When the procedure is called, it outputs a *separator*-separated fields with proper escapes, to the output port. Each field value must be a string. The separator argument can be a character or a string.

You can also specify the record delimiter string by *newline*; for example, you can pass `"\r\n"` to prepare a file to be read by Windows programs.

The output of field is quoted when it contains special characters— which automatically includes characters in *separator*, *quote-char* and *newline* argument, plus the characters in the char-set given to *special-char-set*; its default is `#[;\s]`.

Middle-level API

Occasionally, CSV files generated from spreadsheet contains superfluous rows/columns and we need to make sense of them. Here are some utilities to help them.

A typical format of such spreadsheet-generated CSV file has the following properties:

1. There's a "header row" near the top; not necessarily the very first row, but certainly it comes before any real data. It signifies the meaning of each column of the data. There may be superfluous columns inserted just for cosmetics, and sometimes the order of columns are changed when the original spreadsheet is edited. So we need some flexibility to interpret the input data.
2. "Record rows" follow the header row. It contains actual data. There may be superfluous rows inserted just for cosmetics. Also, it's often the case that the end of data isn't marked clearly (you find large number of rows of empty strings, for example).

The main purpose of middle-level CSV parser is to take the output of low-level parser, which is a list of lists of strings, and find the header row, and then convert the subsequent record rows into tuples according to the header row. A tuple is just a list of strings, but ordered in the same way as the specified header spec.

`csv-rows->tuples` *rows header-specs :key required-slots allow-gap?* [Function]

{`text.csv`} Convert input rows (a list of lists of strings) to a list of tuples. A tuple is a list of slot values.

First, it looks for a header row that matches the given *header-spec*. Once the header row is found, parse the subsequent rows as record row according to the header and convert them to tuples. If no header is found, `#f` is returned.

Header-specs is a list of header spec, each of which can be either a string, a regexp, or a predicate on a string. If it's a string, a column that exactly matches the string is picked. If it's a regexp, a column that matches the regexp is picked. And if it's a predicate, as you might have already guessed, a column that satisfies the predicate is picked.

The order fo *header-specs* determines the order of columns of output tuples.

Required-slots determines if the input row is a valid record row or not. The structure of *required-slots* is as follows:

```
<required-slots> : (<spec> ...)
<spec> : <header-spec> | (<header-spec> <predicate>)
```

The `<header-spec>` compared to the elements of *header-slot* (by `equal?`) to figure out which columns to check. A single `<header-spec>` in `<spec>` means that the column shouldn't be empty for a valid record row. If <*spec*> is a list of `<header-spec>` and `<predicate>`, then the value of the column corresponds to the <*header-spec*> is passed to <*predicate*> to determine if it's a valid record row.

If *required-slots* is omitted or an empty list, any row with at least one non-empty column to be included in the tuple.

If *allow-gap?* is `#t`, it keeps reading rows until the end, skipping invalid rows. If *allow-gap?* is `#f` (default), it stops reading once it sees an invalid row after headers.

Let's see an example. Suppose we have the following CSV file as `data.csv`. It has extra rows and columns, as is often seen in spreadsheet-exported files.

```
,,,,,,,,
"Exported data",,,,,,,,
,,,,,,,,
,,Year,Country,,Population,GDP,,Note
,,1958,"Land of Lisp",,39994,"551,435,453",,
,,1957,"United States of Formula Translators",,115333,"4,343,225,434",,Estimated
,,1959,"People's Republic of COBOL",,82524,"3,357,551,143",,
```

```
,,1970,"Kingdom of Pascal",,3785,,,"GDP missing"
,,,,,,,,
,,1962,"APL Republic",,1545,"342,335,151",,
```

You can extract tuples of Country, Year, GDP and Population, as follows:

```
(use text.csv)
(use gauche.generator)

(call-with-input-file "data.csv"
  (^p (csv-rows->tuples
       (generator->list (cute (make-csv-reader #\,) p))
       '("Country" "Year" "GDP" "Population"))))
 ⇒
  (("Land of Lisp" "1958" "551,435,453" "39994")
   ("United States of Formula Translators" "1957" "4,343,225,434" "115333")
   ("People's Republic of COBOL" "1959" "3,357,551,143" "82524")
   ("Kingdom of Pascal" "1970" "" "3785"))
```

Note that irrelevant rows are skipped, and columns in the results are ordered as specified in the *header-specs*.

Since there's a gap (empty row) after the "Kingdom of Pascal" entry, `csv-rows->tuples` stops processing there by default. If you want to include "APL Republic", you have to pass `:allow-gap? #t` to `csv-rows->tuples`.

The next example gives `:required-slots` option to eliminate rows with missing some of Year, Country or GDP—thus "Kingdom of Pascal" is omitted from the result, while "APL Republic" is included because of `:allow-gap?` argument. (It also checks Year has exactly 4 digits.)

```
(call-with-input-file "data.csv"
  (^p (csv-rows->tuples
       (generator->list (cute (make-csv-reader #\,) p))
       '("Country" "Year" "GDP" "Population")
        :required-slots '(("Year" #/^\d{4}$/) "Country" "GDP")
        :allow-gap? #t)))
 ⇒
 (("Land of Lisp" "1958" "551,435,453" "39994")
  ("United States of Formula Translators" "1957" "4,343,225,434" "115333")
  ("People's Republic of COBOL" "1959" "3,357,551,143" "82524")
  ("APL Republic" "1962" "342,335,151" "1545"))
```

The following two procedures are ingredients of `csv-rows->tuples`:

`make-csv-header-parser` *header-specs* [Function]

{`text.csv`} Create a procedure that takes a row (a list of strings) and checks if if it matches the criteria specified by *header-specs*. (See `csv-rows->tuples` above about *header-specs*.) If the input satisfies the spec, it returns a permuter vector that maps the tuple positions to the input column numbers. Otherwise, it returns `#f`.

The permuter vector is a vector of integers, where K-th element being I means the K-th item of the tuple should be taken from I-th column.

Let's see the example. Suppose we know that the input contains the following row as the header row:

```
(define *input-row* '("" "" "Year" "Country" "" "Population" "GDP" "Notes"))
```

We want to detect that row, but we only needs Country, Year, GDP and Population columns, in that order. So we create a header parser as follows:

```
(define header-parser
  (make-csv-header-parser '("Country" "Year" "GDP" "Population")))
```

Applying this header parser to the input data returns the permuter vector:

```
(header-parser *input-row*)
```

```
⇒ #(3 2 6 5)
```

It means, the first item of tuple (Country) is in the 3rd column of the input, the second item of tuple (Year) is in the 2nd column of the input, and so on. This permuter vector can be used to parse record rows to get tuples.

`make-csv-record-parser` *header-slots permuter :optional required-slots* [Function]

{`text.csv`} Create a procedure that converts one input row into a tuple.

Permuter is the vector returned by `make-csv-header-parser`.

See `cvs-rows->tuples` above for *header-slots* and *required-slots* arguments.

12.52 `text.diff` - Calculate difference of text streams

`text.diff` [Module]

This module calculates the difference of two text streams or strings, using `util.lcs` (see Section 12.66 [The longest common subsequence], page 798).

`diff` *src-a src-b :key reader eq-fn* [Function]

{`text.diff`} Generates an "edit list" from text sources *src-a* and *src-b*.

Each of text sources, *src-a* and *src-b*, can be either an input port or a string. If it is a string, it is converted to a string input port internally. Then, the text streams from both sources are converted to sequences by calling *reader* repeatedly on them; the default of *reader* is `read-line`, and those sequences are passed to `lcs-edit-list` to calculate the edit list. The equality function *eq-fn* is also passed to `lcs-edit-list`.

An edit list is a set of commands that turn the text sequence from `src-a` to the one from `src-b`. See the description of `lcs-edit-list` for the detailed explanation of the edit list.

```
(diff "a\nb\nc\nd\n" "b\ne\nd\nf\n")
⇒
  (((- 0 "a"))
   ((- 2 "c") (+ 1 "e"))
   ((+ 3 "f")))
```

`diff-report` *src-a src-b :key reader eq-fn writer* [Function]

{`text.diff`} A convenience procedure to take the diff of two text sources and display the result nicely. This procedure calls `lcs-fold` to calculate the difference of two text sources. The meanings of *src-a*, *src-b*, *reader* and *eq-fn* are the same as `diff`'s.

Writer is a procedure that takes two arguments, the text element and a type, which is either a symbol `+`, a symbol `-`, or `#f`. If the text element is only in *src-a*, *writer* is called with the element and `-`. If the text element is only in *src-b*, it is called with the element and `+`. If the text element is in both sources, it is called with the element and `#f`. The default procedure of *writer* prints the passed text element to the current output port in unified-diff-like format:

```
(diff-report "a\nb\nc\nd\n" "b\ne\nd\nf\n")
```

displays:

```
- a
  b
- c
+ e
  d
+ f
```

12.53 `text.edn` - EDN parsing and construction

`text.edn` [Module]

EDN (Extensible Data Notation) is a subset of Clojure literals for data exchange. This module provides utilities to read and write EDN format. See `https://github.com/edn-format/edn` for the details of EDN.

EDN	Gauche	Note
`true`	`#t`	
`false`	`#f`	
`nil`	`nil`	Clojure's nil is not a symbol but a special value; since Clojure can't have a symbol named nil, we can map it to Gauche's symbol `nil`.
number	`<real>`	Integers and floating point numbers. The `N` and `M` suffixes in Clojure are ignored.
symbol	`<symbol>`	Clojure's symbol name has some restrictions, so not all Gauche symbols map to EDN symbols. Clojure's namespace-prefixed symbol, e.g. `foo/bar` simply maps to Gauche's symbol `foo/bar`; we provide utility procedure to extract namespace and basename parts.
keyword	`<keyword>`	Clojure has keywords distinct from symbols. They are mapped to Gauche's keywords (which is a subtype of symbols). Gauche's keywords can also be symbols, but no Clojure symbols begin with : so there won't be a conflict.
list	`<list>`	Clojure lists are Gauche lists. Note that Clojure doesn't allow improper lists.
vector	`<vector>`	Clojure vectors are Gauche vectors.
map	`<hash-table>`	Clojure's map becomes Gauche's hashtable with `edn-comparator` for hashing and comparison.
set	`<set>`	Clojure's set becomes Gauche's set with `edn-comparator` for comparison. See Section 10.3.5 [R7RS sets], page 525, for interface of sets.
tagged object	`<edn-object>`	Tagged objects are mapped to `<edn-object>` by default. You can customize the parser/writer to map tagged objects with a specific tag to a specific Gauche objects.

Parsing

`<edn-parse-error>` [Condition Type]

When the parser encounters an error, this condition is thrown. Inherits `<error>`.

`parse-edn` *:optional iport* [Function]

{`text.edn`} Read one EDN representation from the given input port, and returns Gauche object created from it. If *iport* is omitted, current input port is assumed.

When the parser encounters unparsable sequence, it raises `<edn-parse-error>`.

Note that *iport* may be read ahead for characters. Suppose the input consists of `abc{:a b}`, i.e. a symbol immediately followed by a map. The parser need to read `{` to know the end of the symbol. The read-ahead brace isn't pushed back to the *iport*. So it would be a problem if you keep reading more EDN subsequently. Use `parse-edn*` if you want to read multiple objects.

`parse-edn*` *:optional iport* [Function]

{`text.edn`} Read EDN representations repeatedly from the given input port and returns a list of them. If *iport* is omitted, current input port is assumed.

When the parser encounters unparsable sequence, it raises `<edn-parse-error>`.

`parse-edn-string` *str* [Function]

{`text.edn`} A convenience procedure to parse EDN representation in a string *str*, and returns the read object.

When the parser encounters unparsable sequence, it raises `<edn-parse-error>`.

```
(parse-edn-string "[1 2 (3 4) {:a 5}]")
 ⇒ #(1 2 (3 4) #<hash-table general 0x1f05780>)
```

Constructing

`construct-edn` *obj :optional oport* [Function]

{`text.edn`} Write out an EDN representation of object *obj* to the output port *oport*. If *oport* is omitted, current output port is assumed.

If *obj* contains an object that doesn't have a defined EDN representation, a generic function *edn-write* is called on it. See Customization heading below. If no method is defined for the object, an error is signaled.

`construct-edn-string` *obj* [Function]

{`text.edn`} Returns an EDN representation of *obj* in a string.

If *obj* contains an object that doesn't have a defined EDN representation, a generic function *edn-write* is called on it. See Customization heading below. If no method is defined for the object, an error is signaled.

(construct-edn-string '#(1 2 "abc")) ⇒ "[1 2 \"abc\"]"

Utilities

`edn-equal?` *a b* [Function]

{`text.edn`} Test equality of two objects that are read from EDN representation.

`edn-comparator` [Variable]

{`text.edn`} A comparator that uses `edn-equal?` for the equality predicate. Corresponding has function is also included. EDN maps and sets become Gauche hash-tables and sets with this comparator.

`edn-map` *key value ...* [Function]
`edn-set` *item ...* [Function]

{`text.edn`} Convenience procedures to create hash-tables and sets compatible for EDN.

`<edn-object>` [Class]

{`text.edn`} EDN tagged object becomes an instance of this class by default. The instance has the following slots, both are immutable:

`tag` [Instance Variable of `<edn-object>`]

Object's tag. A symbol.

`payload` [Instance Variable of `<edn-object>`]

Object's payload. Can be any object that can be representable in EDN.

For example, when you read `#myobject {:a 1 :b 2}`, the tag is `myobject` and the payload is a hashtable containing mapping `{:a 1 :b 2}`.

`make-edn-object` *tag payload* [Function]

{`text.edn`} Returns a new `<edn-object>` instance. Note: Arguments are not checked. It's caller's responsibility to pass valid arguments to guarantee it's serializable as EDN.

`edn-object?` *obj* [Function]

{`text.edn`} Returns `#t` iff *obj* is an instance of `<edn-object>`.

`edn-object-tag` *edn-object* [Function]
`edn-object-payload` *edn-object* [Function]

{`text.edn`} Returns the tag and the payload of *edn-object*, respectively.

`edn-symbol-prefix` *symbol* [Function]
`edn-symbol-basename` *symbol* [Function]

{`text.edn`} Return prefix and basename part of the symbol, respectively.

```
(edn-symbol-prefix 'foo/bar) ⇒ foo
(edn-symbol-basename 'foo/bar) ⇒ bar

(edn-symbol-prefix 'bar) ⇒ #f
(edn-symbol-basename 'bar) ⇒ bar
```

`edn-valid-symbol-name?` *str* [Function]

{`text.edn`} Returns `#t` iff a string *str* can be a valid Clojure symbol name. It may have namespace prefix.

Customization

You can map EDN tagged objects to other Gauche objects.

`register-edn-object-handler!` *tag handler* [Function]

{`text.edn`} *Tag* is a symbol, and *handler* is `#f` or a procedure that takes a tag symbol and a payload object.

Tag must have a name valid as Clojure symbol, or an error is signaled.

After the parser reads a tagged object with a symbol *tag* and payload, it calls *handler*, and the returned object becomes the result of the parser, instead of `<edn-object>`. Registering `#f` removes the previously registered handler.

This procedure is thread-safe.

The following example makes EDN `#u8vector[1 2 3 4]` to be read as `#u8(1 2 3 4)`:

```
(register-edn-object-handler! 'u8vector
                              (^[tag vec] (vector->u8vector vec)))
```

`edn-object-handler` *tag* [Function]

{`text.edn`} Returns a handler registered with a symbol *tag*. If a handler is not registered for *tag*, `#f` is returned. This procedure is thread-safe.

`edn-write` *obj* [Generic Function]

{`text.edn`} Write EDN representation of *obj* to the current output port. The `construct-edn` procedure calls this internally.

To write out a Gauche object as EDN tagged object, define a method to this generic function. In the method you can call `edn-write` recursively to write out components of the object.

The following example writes `#u8(1 2 3 4)` as EDN `#u8vector[1 2 3 4]`:

```
(define-method edn-write ((x <u8vector>))
  (display "#u8vector")
  (edn-write (u8vector->vector x)))
```

12.54 `text.gettext` - Localized messages

`text.gettext` [Module]

This module provides utilities to deal with localized messages. The API is compatible to GNU's gettext, and the messages are read from `*.po` and `*.mo` files, so that you can use the GNU gettext toolchain to prepare localized messages. However, the code is written from scratch by Alex Shinn and doesn't depend on GNU's gettext library.

This implementation extends GNU's gettext API in the following ways:

- It can read from multiple message files in cascaded way, allowing applications to share a part of message files.
- It supports multiple locale/domain simultaneously.

SRFI-29 (see Section 11.9 [Localization], page 606) provides another means of message localization. A portable program may wish to use srfi-29, but generally `text.gettext` is recommended in Gauche scripts because of its flexibility and compatibility to existing message files.

Gettext-compatible API

`textdomain` *domain-name :optional locale dirs cdir cached? lookup-cached?* [Function]

{`text.gettext`} Sets up the default domain and other parameters for the application. The setting affects to the following `gettext` call.

Domain is a string or list of strings specifying the domain (name of `.mo` or `.po` files) as in C gettext. You can pass `#f` as *domain-name* just to get the default domain accessor procedure. You can alo pass multiple domains to *domain-name*.

```
(textdomain '("myapp" "gimp"))  ; search 1st myapp, then gimp
(gettext "/File/Close")         ; "Close" from gimp unless overridden
```

Locale is a string or list of strings in the standard Unix format of `LANG[_REGION][.ENCODING]`. You can also pass a list of locales to specify fallbacks.

```
(textdomain "myapp" '("ru" "uk"))  ; search 1st Russian then Ukranian,
(gettext "Hello, World!")          ; which are somewhat similar
```

Dirs is the search path of directories which should hold the `LOCALE/CDIR/` directories which contain the actual message catalogs. This is always appended with the system default, e.g. `"/usr/share/locale"`, and may also inherit from the `GETTEXT_PATH` colon-delimited environment variable.

Cdir is the category directory, defaulting to either the `LC_CATEGORY` environment variable or the appropriate system default (e.g. `LC_MESSAGES`). You generally won't need this.

Cached? means to cache individual messages, and defaults to `#t`.

Lookup-cached? means to cache the lookup dispatch generated by these parameters, and defaults to `#t`.

`Textdomain` just passes these parameters to the internal `make-gettext`, and binds the result to the global dispatch used by `gettext`. You may build these closures manually for convenience in using multiple separate domains or locales at once (useful for server environments). See the description of `make-gettext` below.

`Textdomain` returns an *accessor procedure* which packages information of the domain. See `make-gettext` below for the details.

`gettext` *msg-id* [Function]

{`text.gettext`} Returns a translated message of *msg-id*. If there's no translated message, *msg-id* itself is returned.

`ngettext` *msg-id :optional msg-id2 num* [Function]

{`text.gettext`} Similar to *gettext*, but it can be used to handle plural forms. Pass a singular form to *msg-id*, and plural form to `msg-id2`. The *num* argument is used to determine the plural form. If no message catalog is found, *msg-id* is returned when *num* is 1, and *msg-id2* otherwise.

`bindtextdomain` *domain dirs* [Function]

{`text.gettext`} Sets the search path of domain *domain* to *dirs*, which may be just a single directory name or a list of directory names.

`dgettext` *domain msg-id* [Function]

`dcgettext` *domain msg-id locale* [Function]

{`text.gettext`} Returns a translated message of *msg-id* in *domain*. `Dcgettext` takes *locale* as well.

Low-level flexible API

The following procedure is more flexible interface, on top of which the gettext-compatible APIs are written.

`make-gettext` *:optional domain locale dirs gettext-cached? lookup-cached?* [Function]

{`text.gettext`} Creates and returns an *accessor procedure*, which encapsulates methods to retrieve localized messages.

The meaning of arguments are the same as `textdomain` above. Indeed, `textdomain` just calls `make-gettext`, and later it binds the result to the global parameter. If you wish to have multiple independent domains within a single program, you can call `make-gettext` directly and manage the created accessor procedure by yourself.

```
(define my-gettext (make-gettext "myapp"))
(define _ (my-gettext 'getter))
(_ "Hello, World!")
```

12.55 `text.html-lite` - Simple HTML document construction

`text.html-lite` [Module]

Provides procedures to construct an HTML document easily. For example, you can construct an HTML table by the following code:

```
(html:table
  (html:tr (html:th "Item No") (html:th "Quantity"))
  (html:tr (html:td 1) (html:td 120))
  (html:tr (html:td 2) (html:td 30))
  (html:tr (html:td 3) (html:td 215)))
```

See the description of `html:element` below for details.

This module does little check for the constructed html documents, such as whether the attributes are valid, and whether the content of the element matches DTD. It does not provide a feature to parse the html document neither. Hence the name ‘lite’.

`html-escape` [Function]

`html-escape-string` *string* [Function]

{`text.html-lite`} Escapes the “unsafe” characters in HTML. `html-escape` reads input string from the current input port and writes the result to the current output port. `html-escape-string` takes the input from *string* and returns the result in a string.

html-doctype *:key type* [Function]

{text.html-lite} Returns a doctype declaration for an HTML document. *type* can be either one of the followings (default is `:html-4.01-strict`).

`:html-4.01-strict, :html-4.01, :strict`
: HTML 4.01 Strict DTD

`:html-4.01-transitional, :transitional`
: HTML 4.01 Transitional DTD

`:html-4.01-frameset, :frameset`
: HTML 4.01 Frameset DTD

`:xhtml-1.0-strict, :xhtml-1.0`
: XHTML 1.0 Strict DTD

`:xhtml-1.0-transitional`
: XHTML 1.0 Transitional DTD

`:xhtml-1.0-frameset`
: XHTML 1.0 Frameset DTD

`:xhtml-1.1`
: XHTML 1.1 DTD

html:*element args* ... [Function]

{text.html-lite} Construct an HTML element *element*. Right now, the following elements are provided. (The elements defined in HTML 4.01 DTD, `http://www.w3.org/TR/html4/sgml/dtd.html`).

```
a        abbr      acronym   address     area      b
base     bdo       big       blockquote  body      br
button   caption   cite      code        col       colgroup
dd       del       dfn       div         dl        dt
em       fieldset  form      frame       frameset
h1       h2        h3        h4          h5        h6
head     hr        html      i           iframe    img
input    ins       kbd       label       legend    li
link     map       meta      noframes    noscript  object
ol       optgroup  option    p           param     pre
q        samp      script    select      small     span
strong   style     sub       sup         table     tbody
td       textarea  tfoot     th          thead     title
tr       tt        ul        var
```

The result of these functions is a tree of text segments, which can be written out to a port by `write-tree` or can be converted to a string by `tree->string` (see Section 12.61 [Lazy text construction], page 793).

You can specify attributes of the element by using a keyword-value notation before the actual content.

```
(tree->string (html:a :href "http://foo/bar" "foobar"))
  ⇒
  "<a href=\"http://foo/bar\">foobar</a\n>"

(tree->string
  (html:table :width "100%" :cellpadding 0 "content here"))
  ⇒
```

```
"<table width=\"100%\" cellpadding=\"0\">content here</table\n>"
```

The boolean value given to the attribute has a special meaning. If `#t` is given, the attribute is rendered without a value. If `#f` is given, the attribute is not rendered.

```
(tree->string (html:table :border #t))
  ⇒ "<table border></table\n>"

(tree->string (html:table :border #f))
  ⇒ "<table></table\n>"
```

Special characters in attribute values are escaped by the function, but the ones in the content are not. It is caller's responsibility to escape them.

The functions signal an error if a content is given to the HTML element that doesn't take a content. They do not check if the given attribute is valid, neither if the given content is valid for the element.

Note: You might have noticed that these procedures insert a newline before `>` of the closing tag. That is, the rendered HTML would look like this:

```
<table><tr><td>foo</td
><td>bar</td
></tr
></table
>
```

We intentionally avoid inserting newlines after the closing tag, since *it depends on the surrounding context whether the newline is significant or not.* We may be able to insert newlines after the elements directly below a `<head>` element, for example, but we cannot in a `<p>` element, without affecting the content.

There are three possible solutions: (1) not to insert newlines at all, (2) to insert newlines within tags, and (3) to insert newlines only at the safe position. The first one creates one long line of HTML, and although it is still valid HTML, it is inconvenient to handle it with line-oriented tools. The third one requires the rendering routine to be aware of DTD. So we took the second approach.

12.56 `text.parse` - Parsing input stream

`text.parse` [Module]

A collection of utilities that does simple parsing from the input port. The API is inspired, and compatible with Oleg Kiselyov's input parsing library ([OLEG1], page 831). His library is used in lots of other libraries, notably, a full-Scheme XML parser/generator SSAX ([SSAX], page 831).

You can use this module in place of his `input-parse.scm` and `look-for-str.scm`.

I reimplemented the functions to be efficient on Gauche. Especially, usage of `string-set!` is totally avoided. I extended the interface a bit so that they can deal with character sets and predicates, as well as a list of characters.

These functions work sequentially on the given input port, that is, they read from the port as much as they need, without buffering extra characters.

`find-string-from-port?` *str in-port :optional max-no-chars* [Function]

{`text.parse`} Looks for a string *str* from the input port *in-port*. The optional argument *max-no-chars* limits the maximum number of characters to be read from the port; if omitted, the search span is until EOF.

If *str* is found, this function returns the number of characters it has read. The next read from *in-port* returns the next char of *str*. If *str* is not found, it returns `#f`.

Note: Although this procedure has '`?`' in its name, it may return non-boolean value, contrary to the Scheme convention.

`peek-next-char` *:optional port* [Function]
{`text.parse`} Discards the current character and peeks the next character from *port*. Useful to look ahead one character. If *port* is omitted, the current input port is used.

In the following functions, *char-list* refers to one of the followings:

- A character set.
- A list of characters, character sets and/or symbol `*eof*`.

That denotes a set of characters. If a symbol `*eof*` is included, the EOF condition is also included. Without `*eof*`, the EOF condition is regarded as an error.

`assert-curr-char` *char-list string :optional port* [Function]
{`text.parse`} Reads a character from *port*. If it is included in *char-list*, returns the character. Otherwise, signals an error with a message containing *string*. If *port* is omitted, the current input port is used.

`skip-until` *char-list/number :optional port* [Function]
{`text.parse`} *char-list/number* is either a char-list or a number. If it is a number; it reads that many characters and returns `#f`. If the input is not long enough, an error is signaled. If *char-list/number* is a char-list, it reads from *port* until it sees a character that belongs to the char-list. Then the character is returned. If *port* is omitted, the current input port is used.

`skip-while` *char-list :optional port* [Function]
{`text.parse`} Reads from *port* until it sees a character that does not belong to *char-list*. The character remains in the stream. If it reaches EOF, an EOF is returned. If *port* is omitted, the current input port is used.

This example skips whitespaces from input. Next read from port returns the first non-whitespace character.

```
(skip-while #[\s] port)
```

`next-token` *prefix-char-list break-char-list :optional comment port* [Function]
{`text.parse`} Skips any number of characters in *prefix-char-list*, then collects the characters until it sees *break-char-list*. The collected characters are returned as a string. The break character remains in the *port*.

If the function encounters EOF and `*eof*` is not included in *break-char-list*, an error is signaled with *comment* is included in the message.

`next-token-of` *char-list/pred :optional port* [Function]
{`text.parse`} Reads and collects the characters as far as it belongs to *char-list/pred*, then returns them as a string. The first character that doesn't belong to *char-list/pred* remains on the port.

char-list/pred may be a char-list or a predicate that takes a character. If it is a predicate, each character is passed to it, and the character is regarded to "belong to" *char-list/pred* when it returns a true value.

`read-string` *n :optional port* [Function]
{`text.parse`} This is like built-in `read-string` (see Section 6.22.7.1 [Reading data], page 227), except that this returns `""` when the input already reached EOF.

Provided for the compatibility for the code that depends Oleg's library.

12.57 `text.progress` - Showing progress on text terminals

`text.progress` [Module]

This module provides a utility to report a progress of processing on a text terminal, using characters to display bar chart.

The generic format of a progress bar consists of a single line of text, which is splitted into several parts; a header, which displays the title; followed by a bar, a numeric part, and a time part, as shown in the following example (only the line beginning with "foo" is actually displayed).

```
<-header-> <-------bar---------> <-num-><-time->      <---info---->
foo       |#############        |123/211   01:21 ETA  compiling...
          ^
          separator
```

Various things like the character used in the bar chart or the format of the numeric progress can be configured.

Internally a progress bar maintains two numbers, the maximum (goal) value and the current value. The bar shows the proportion of the current value relative to the maximum value. The numeric progress shows the current value over the maximum value by default, but you can configure it to show only the current value or percentage, for example.

A progress bar also has two states, "in progress" and "finished". When it is in progress, every time the text is displayed it is followed by `#\return`, so that the next display overwrites the bar, and the time part shows ETA (estimated time of arrival). Once it becomes finished, the last line of text is displayed with `#\newline`, and the time part shows the actual time it took to finish.

This module provides only one procedure, `make-text-progress-bar`, which packages the progress bar feature in a closure and returns it.

`make-text-progress-bar` *:key header header-width bar-char bar-width num-width num-format time-width info info-width separator-char max-value port* [Function]

{`text.progress`} Returns a procedure that packages operations on the progress bar. The procedure can be called with a symbol indicating an operation, and an optional numeric argument.

proc `'show`
: Redisplays the progress bar. All other operations implies redisplay, so you don't need to use this unless you have a specific reason to redisplay the current state.

proc `'set` *value*
: Sets the current value to *value*, then redisplays the progress bar. If *value* exceeds the max value, it is clipped by the max value.

proc `'inc` *value*
: Increments the current value by *value*, then redisplays the progress bar. If the current value exceeds the max value, it is clipped by the max value.

proc `'finish`
: Puts the progress bar to the "finished" state, then redisplays it. The time part shows the total elapsed time, and the line is terminated by `#\newline` so that it won't be clobbered. Once a progress bar becomes "finished", there's no way to put it back "in progress".

proc 'set-info *text*
: Changes the text displayed in the “info” part. To use the info part, you have to give a positive value to *info-width* keyword argument of `make-text-progress-bar`.

proc 'set-header *text*
: Changes the text displayed in the “header’ area.

The keyword arguments are used to customize the display:

header
: The text to be displayed in the header part. This can be changed later, by sending `set-header` message to the created progress bar.

header-width
: The width of the header part, in number of characters. The header text is displayed left-aligned in the part. If the header text is longer than the width, the excess characters are omitted. The default is 14.

bar-char
: A character used to draw a bar chart. The default is `#\#`.

bar-width
: The width of the bar chart part, in number of characters. The default is 40.

num-width
: The width of the numeric part, in number of characters. The default is 9. Setting this to 0 hides the numeric part.

num-format
: A procedure to format the numeric part. Two arguments are passed; the current value and the maximum value. It must return a string. The default is the following procedure.

```
(lambda (cur max)
  (format "~d/~d" cur max))
```

time-width
: The width of the time part, in number of characters. The default is 7. Settings this to 0 hides the time part.

info
: The text to be displayed in the info part. This text can be changed later by sending `set-info` message to the created progress bar. Note that you have to give a positive number to *info-width* keyword argument to enable the info part.

info-width
: The width of the info part. The default value is zero, which means the info part is not displayed.

separator-char
: A character put around the bar part. Default is `#\|`. You can pass `#f` not to display the separators.

max-value
: The maximum value of the progress bar. Must be a positive real number. Default is 100.

port
: An output port to which the progress bar is displayed. The default value is the current output port when `make-text-progress-bar` is called.

Here’s a simple example, using customized numeric part:

```
(use text.progress)

(define (main args)
  (define (num-format cur max)
    (format "~d/~d(~3d%)" cur max
```

```
                 (round->exact (/. (* cur 100) max))))

       (let ((p (make-text-progress-bar :header "Example"
                                        :header-width 10
                                        :bar-char #\o
                                        :num-format num-format
                                        :num-width 13
                                        :max-value 256)))
         (do ((i 0 (+ i 1)))
             ((= i 256) (p 'finish))
           (p 'inc 1)
           (sys-select #f #f #f 50000))))
```

12.58 `text.sql` - SQL parsing and construction

`text.sql` [Module]

This module provides a utility to parse and construct SQL statement.

It is currently under development, and we only have a tokenization routine. The plan is to define S-expression syntax of SQL and provides a routine to translate one form to the other.

Note: If you're looking for a routine to escape strings to be safe in SQL, see `dbi-escape-sql` in Section 12.17.1 [DBI user API], page 677.

`sql-tokenize` *sql-string* [Function]

{`text.sql`} Tokenize a SQL statement *sql-string*. The return value is a list of tokens, where each token is represented by one of the following forms.

```
<symbol>               Special delimiter.  One of the followings:
                       + - * / < = > <> <= >= ||
<character>            Special delimiter.  One of the followings:
                       #\, #\. #\( #\) #\;
<string>               Regular identifier
(delimited <string>)  Delimited identifier
(parameter <num>)     Positional parameter (?)
(parameter <string>)  Named parameter (:foo)
(string    <string>)  Character string literal
(number    <string>)  Numeric literal
(bitstring <string>)  Binary string.  <string> is like "01101"
(hexstring <string>)  Binary string.  <string> is like "3AD20"
```

If it encounters an untokenizable string, it raises an `<sql-parse-error>` condition.

`<sql-parse-error>` [Condition Type]

{`text.sql`} A condition to indicate an SQL parse error. Inherits `<error>`.

`sql-string` [Instance Variable of `<sql-parse-error>`]

Holds the source SQL string.

12.59 `text.template` - Simple template expander

`text.template` [Module]

This module lifts Gauche's built-in string interpolation feature to be more general template engine.

Gauche's string interpolation syntax is expanded at read time and then handled by macro expanders, and becomes a simple Scheme code fragment. For example, if you have this:

```
(let ([x 10])
  #"The square of x is ~(* x x).")
```

It is eventually converted to this after macro expansion:

```
(let ([x 10])
  (string-append '"The square of x is " (x->string (* x.0 x.0)) '"."))
```

It is a kind of template expansion, but you have to have the template string as a literal, so it's restricted. With this module, you can feed template string and the bindings of the value at the runtime:

```
(define *template* "The square of x is ~(* x x).")

(expand-template-string *template*
  (make-template-environment :bindings '(x 10)))
  ⇒ "The square of x is 100."
```

The syntax of template strings is the same as string interpolation (see Section 6.12.4 [String interpolation], page 155); that is, tokens following `~` is read as a Scheme expression. In case if the token is a symbol and you need to delimit it from subsequent characters, you can use symbol escape by `|`.

You also need to provide a *template environment*, where the expressions in the template is evaluated. Note that, unlike string interpolation, those expressions can't refer to the local bindings.

`expand-template-string` *template env* [Function]

Expands a template string *template* with a template environment *env*, and returns the result string.

`expand-template-file` *filename env :optional paths* [Function]

Reads a template string from a file named by *filename*, expands it with a template environment *env*, and returns the result string.

If *filename* is not an absolute path, it is looked in the directories listed in *paths*.

`make-template-environment` *:key extends imports bindings* [Function]

Creates and returns a template environment. A template environment is like a module (see Section 4.13 [Modules], page 71): It maps symbols to values, and it can import bindings from other modules, or extend other modules.

The keyword arguments *extends* and *imports* must be a list of symbols; they specify names of modules to inherit from or to import from.

The keyword arguments *bindings* must either be a dictionary (anything that inherits `<dictionary>`), or a key-value list. The mappings represented by it are incorporated to the environment.

12.60 `text.tr` - Transliterate characters

`text.tr` [Module]

This module implements a transliterate function, that substitutes characters of the input string. This functionality is realized in Unix `tr(1)` command, and incorporated in various programs such as `sed(1)` and `perl`.

Gauche's `tr` is aware of multibyte characters.

tr *from-list to-list :key :complement :delete :squeeze :table-size :input :output* [Function]

{text.tr} Reads from *input* and writes to *output*, with transliterating characters in *from-list* to the corresponding ones in *to-list*. Characters that doesn't appear in *from-list* are passed through.

The default values of *input* and *output* are current input port and current output port, respectively.

Both *from-list* and *to-list* must be strings. They may contain the following special syntax. Other characters that doesn't fits in the syntax are taken as they are.

x-y Expanded to the increasing sequence of characters from `x` to `y`, inclusive. The order is determined by the internal character encoding system; generally it is safer to limit use of this within the range of the same character class. The character `x` must be before `y`.

x*n Repeat `x` for `n` times. `n` is a decimal number notation. Meaningful only in *to-list*; it is an error to use this form in *from-list*. If `n` is omitted or zero, `x` is repeated until *to-list* matches the length of *from-list* (any character after it is ignored).

\x Represents `x` itself. Use this escape to avoid a special character to be interpreted as itself. Note that if you place a backslash in a string, you must write `\\`, for the Scheme reader also interprets backslash as a special character.

There's no special sequence to represent non-graphical characters, for you can put such characters by the string syntax.

Here's some basic examples.

```
;; swaps case of input
(tr "A-Za-z" "a-zA-Z")

;; replaces 7-bit non-graphical characters to '?'
(tr "\x00-\x19\x7f" "?*")
```

If *to-list* is shorter than *from-list*, the behavior depends on the keyword argument *delete*. If a true value is given, characters that appear in *from-list* but not in *to-list* are deleted. Otherwise, the extra characters in *from-list* are just passed through.

When a true value is specified to *complement*, the character set in *from-list* is complemented. Note that it implies *huge* set of characters, so it is not very useful unless either output character set is a single character (using '*') or used with `delete` keyword.

When a true value is specified to *squeeze*, the sequence of the same replaced characters is squeezed to one. If *to-list* is empty, the sequence of the same characters in *from-list* is squeezed.

Internally, `tr` builds a table to map the characters for efficiency. Since Gauche can deal with potentially huge set of characters, it limits the use of the table for only smaller characters (<256 by default). If you want to transliterate multibyte characters on the large text, however, you might want to use larger table, trading off the memory usage. You can specify the internal table size by *table-size* keyword argument. For example, if you transliterate lots of EUC-JP hiragana text to katakana, you may want to set table size greater than 42483 (the character code of the last katakana).

Note that the pre-calculation to build the transliterate table needs some overhead. If you want to call `tr` many times inside loop, consider to use `build-transliterator` described below.

`string-tr` *string from-list to-list :key :complement :delete :squeeze :table-size* [Function]

{`text.tr`} Works like `tr`, except that input is taken from a string *string*.

`build-transliterator` *from-list to-list :key :complement :delete :squeeze :table-size :input :output* [Function]

{`text.tr`} Returns a procedure that does the actual transliteration. This effectively "precompiles" the internal data structure. If you want to run `tr` with the same sets repeatedly, you may build the procedure once and apply it repeatedly, saving the overhead of initialization.

A note for an edge case: When *input* and/or *output* keyword arguments are omitted, the created transliterator is set up to use current-input-port and/or current-output-port at the time transliterator is called.

```
(with-input-from-file "huge-file.txt"
  (lambda ()
    (let loop ((line (read-line)))
      (unless (eof-object? line) (tr "A-Za-z" "a-zA-Z")))))

;; runs more efficiently...

(with-input-from-file "huge-file.txt"
  (lambda ()
    (let ((ptr (build-transliterator "A-Za-z" "a-zA-Z")))
      (let loop ((line (read-line)))
        (unless (eof-object? line) (ptr))))))
```

12.61 `text.tree` - Lazy text construction

`text.tree` [Module]

Defines simple but commonly used functions for a text construction.

When you generate a text by a program, It is a very common operation to concatenate text segments. However, using string-append repeatedly causes unnecessary copying of intermediate strings, and sometimes such intermediate strings are discarded due to the error situation (for example, think about constructing an HTML document in the CGI script).

The efficient technique is to delay concatenation of those text segments until it is needed. In Scheme it is done very easily by just consing the text segments together, thus forming a tree of text, and then traverse the tree to construct a text. You can even directly writes out the text during traversal, avoiding intermediate string buffer. (Hans Boehm's "cord" library, which comes with his garbage collector library, uses this technique and proves it is very efficient for editor-type application).

Although the traversal of the tree can be written in a few lines of Scheme, I provide this module in the spirits of OnceAndOnlyOnce. Also it's easier if we have a common interface.

`write-tree` *tree :optional out* [Generic Function]

{`text.tree`} Writes out an *tree* as a tree of text, to the output port *out*. If *out* is omitted, the current output port is used.

Two methods are defined for this generic function, as shown below. If you have more complex behavior, you can define more methods to customize the behavior.

`write-tree` ((*tree* <*list*>) *out*) [Method]

`write-tree` ((*tree* <*top*>) *out*) [Method]

{`text.tree`} Default methods. For a list, `write-tree` is recursively called for each element. Any objects other than list is written out using `display`.

`tree->string` *tree* [Function]

{`text.tree`} Just calls the `write-tree` method for *tree* using an output string port, and returns the result string.

12.62 `util.combinations` - Combination library

`util.combinations` [Module]

This module implements several useful procedures of combinations, permutations and related operations.

Most procedures in the module have two variants: a procedure without star (e.g. `permutations`) treats all elements in the given set distinct, while a procedure with star (e.g. `permutations*`) considers duplication. The procedures with star take optional *eq* argument that is used to test equality, which defaults to `eqv?`.

`permutations` *set* [Function]
`permutations*` *set :optional eq* [Function]

{`util.combinations`} Returns a list of all permutations of a list *set*.

```
(permutations '(a b c))
  ⇒ ((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))

(permutations '(a a b))
  ⇒ ((a a b) (a b a) (a a b) (a b a) (b a a) (b a a))

(permutations* '(a a b))
  ⇒ ((a a b) (a b a) (b a a))
```

The number of possible permutations explodes if *set* has more than several elements. Use with care. If you want to process each permutation at a time, consider `permutations-for-each` below.

`permutations-for-each` *proc set* [Function]
`permutations*-for-each` *proc set :optional eq* [Function]

{`util.combinations`} For each permutation of a list *set*, calls *proc*. Returns an undefined value.

`combinations` *set n* [Function]
`combinations*` *set n :optional eq* [Function]

{`util.combinations`} Returns a list of all possible combinations of *n* elements out of a list *set*.

```
(combinations '(a b c) 2)
  ⇒ ((a b) (a c) (b c))

(combinations '(a a b) 2)
  ⇒ ((a a) (a b) (a b))

(combinations* '(a a b) 2)
  ⇒ ((a a) (a b))
```

Watch out the explosion of combinations when *set* is large.

`combinations-for-each` *proc set n* [Function]
`combinations*-for-each` *proc set n :optional eq* [Function]

{`util.combinations`} Calls *proc* for each combination of *n* elements out of *set*. Returns an undefined value.

`power-set` *set* [Function]
`power-set*` *set :optional eq* [Function]

{`util.combinations`} Returns power set (all subsets) of a list *set*.

```
(power-set '(a b c))
  ⇒ (() (a) (b) (c) (a b) (a c) (b c) (a b c))

(power-set* '(a a b)
  ⇒ (() (a) (b) (a a) (a b) (a a b))
```

`power-set-for-each` *proc set* [Function]
`power-set*-for-each` *proc set :optional eq* [Function]

{`util.combinations`} Calls *proc* for each subset of *set*.

`power-set-binary` *set* [Function]

{`util.combinations`} Returns power set of *set*, like `power-set`, but in different order. `Power-set-binary` traverses subset space in depth-first order, while `power-set` in breadth-first order.

```
(power-set-binary '(a b c))
  ⇒ (() (c) (b) (b c) (a) (a c) (a b) (a b c))
```

`cartesian-product` *list-of-sets* [Function]
`cartesian-product-right` *list-of-sets* [Function]

{`util.combinations`} Returns a cartesian product of sets in *list-of-sets*. `Cartesian-product` construct the result in left fixed order (the rightmost element varies first), while `cartesian-product-right` in right fixed order (the leftmost element varies first).

```
(cartesian-product '((a b c) (0 1)))
  ⇒ ((a 0) (a 1) (b 0) (b 1) (c 0) (c 1))

(cartesian-product-right '((a b c) (0 1)))
  ⇒ ((a 0) (b 0) (c 0) (a 1) (b 1) (c 1))
```

12.63 `util.digest` - Message digester framework

`util.digest` [Module]

This module provides a base class and common interface for message digest algorithms, such as MD5 (see Section 12.38 [MD5 message digest], page 730) and SHA (see Section 12.41 [SHA message digest], page 736).

`<message-digest-algorithm-meta>` [Class]

{`util.digest`} A metaclass of message digest algorithm implementation.

`hmac-block-size` [Instance Variable of `<message-digest-algorithm-meta>`]

Specifies the block size (in bytes), which is specific to each algorithm. (This is a slot for each *class* object that implements the algorithm, not for instance of such classes. Only the author of such digest classes needs to care. See `ext/digest/sha.scm` in the source tree for more details.)

`<message-digest-algorithm>` [Class]

{`util.digest`} A base class of message digest algorithm implementation.

The concrete subclass of message digest algorithm has to implement the following methods.

`digest-update!` *algorithm data* [Generic function]
{`util.digest`} Takes the instance of massage-digest algorithm, and updates it with the data *data*, which can be either a u8vector or a (possibly incomplete) string.

`digest-final!` *algorithm* [Generic function]
{`util.digest`} Finalizes the instance of message-digest algorithm, and returns the digest result in an incomplete string.

`digest` *class* [Generic function]
{`util.digest`} A wrapper of digest routines. Given message-digest algorithm *class*, this function reads the input data from current input port until EOF, and returns the digest result in an incomplete string.

`digest-string` *class string* [Generic function]
{`util.digest`} A wrapper of digest routines. Given message-digest algorithm *class*, this function reads the input data from *string*, and returns the digest result in an incomplete string.

`digest-hexify` *digest-result* [Function]
{`util.digest`} An utility procedure. Given the result of digest, *digest-result*, which can be an u8vector or a (possibly incomplete) string, converts it to a hexified string.

12.64 `util.dominator` - Calculate dominator tree

`util.dominator` [Module]
Dominator tree is an auxiliary structure for control flow graphs. It is frequently used in the flow analysis of compilers, but also useful for handling general directed graphs.

`calculate-dominators` *start upstreams downstreams node-comparator* [Function]
{`util.dominator`} The four arguments represent a directed, possibly cyclic, graph. Here, we use `Node` to denote an abstract type of a node of the graph. It can be anything—the algorithm is oblivious on the actual type of nodes.

start `:: Node`
: The start node, or the enter node, of the graph.

upstreams `:: Node -> (Node ...)`
: A procedure that takes a node, and returns its upstream (immediate ancestor) nodes.

downstreams `:: Node -> (Node ...)`
: A procedure that takes a node, and returns its downstream (immediate descendant) nodes.

node-comparator
: A comparator that is used to determine if two nodes are equal to each other. It doesn't need to have comparison procedure (we don't need to see which is smaller than the other), but it has to have hash function, for we use hashtables internally. (See Section 6.2.4 [Basic comparators], page 103, for the details of comparators.)

The procedure returns a list of `(node1 node2)`, where `node2` is the immediate dominator of `node1`.

If there are node in the given graph that are unreachable from *start*, such nodes are ignored and not included in the result.

(A bit of explanation: Suppose you want to go to node X from *start*. There may be multiple routes, but if you have to pass node Y no matter which route you take, then Y is a dominator of X. There may be many dominators of X. Among them, there's always one dominator such that all other X's dominators are also its dominators—in other words, the closest dominator of X—which is called the immediate dominator of X.)

Let's see an example. You have this directed graph:

```
      A (start)
      |
      v
      B <-------+
      |         |
------+-----    |
|          |    |
v          v    |
C -------> D ---+
|          |
v          v
E <------- F
```

Let's represent the graph by a list of `(x y z ...)` where `x` can directly go to either `y z ...`.

```
(define *graph* '((A B)
                  (B C D)
                  (C D E)
                  (D F B)
                  (F E)))
```

Then you can calculate the immediate dominator of each node as follows:

```
(calculate-dominators 'A
  (^n (filter-map (^g (and (memq n (cdr g)) (car g))) *graph*))
  (^n (assoc-ref *graph* n '()))
  eq-comparator)
  ⇒ ((E B) (F D) (D B) (C B) (B A))
```

That is, `E`'s immediate dominator is `B`, `F`'s is `D`, and so on.

The result itself can be viewed as a tree. It is called a dominator tree.

```
      F
      |
      v
E     D     C
|     |     |
|     v     |
+---> B <---+
      |
      v
      A
```

12.65 `util.isomorph` - Determine isomorphism

`util.isomorph` [Module]

Provides a procedure that determines whether two structures are isomorphic.

`isomorphic?` *obj1 obj2 :optional context* [Function]

{`util.isomorph`} Returns `#t` if *obj1* and *obj2* are isomorphic.

context is used if you want to call `isomorphic?` recursively inside `object-isomorphic?` described below.

```
(isomorphic? '(a b) '(a b)) ⇒ #t
```

```
(define x (cons 0 0))
(define y (cons 0 0))
(isomorphic? (cons x x)
             (cons x y))
 ⇒ #f
(isomorphic? (cons x x)
             (cons y y))
 ⇒ #t
```

`object-isomorphic?` *obj1 obj2 context* [Generic Function]

{`util.isomorph`} With this method, you can customize how to determine isomorphism of two objects. Basically, you will call `isomorphic?` recursively for each slots of object you want to traverse; the method should return `#t` if all of the test succeeds, or return `#f` otherwise. *context* is an opaque structure that keeps the traversal context, and you should pass it to `isomorphic?` as is.

The default method returns `#t` if *obj1* and *obj2* are equal (in the sense of `equal?`).

12.66 `util.lcs` - The longest common subsequence

`util.lcs` [Module]

This module implements the algorithm to find the longest common subsequence of two given sequences. The implemented algorithm is based on Eugene Myers' O(ND) algorithm ([Myers86], page 829).

One of the applications of this algorithm is to calculate the difference of two text streams; see Section 12.52 [Calculate difference of text streams], page 779.

`lcs` *seq-a seq-b :optional eq-fn* [Function]

{`util.lcs`} Calculates and returns the longest common sequence of two lists, *seq-a* and *seq-b*. Optional *eq-fn* specifies the comparison predicate; if omitted, `equal?` is used.

```
(lcs '(x a b y) '(p a q b))
 ⇒ (a b)
```

`lcs-with-positions` *seq-a seq-b :optional eq-fn* [Function]

{`util.lcs`} This is the detailed version of `lcs`. The arguments are the same.

Returns a list of the following structure:

```
(length ((elt a-pos b-pos) ...))
```

Length is an integer showing the length of the found LCS. What follows is a list of elements of LCS; each sublist consists of the element, the integer position of the element in *seq-a*, then the integer position of the element in *seq-b*.

```
(lcs-with-positions '(a) '(a))
 ⇒ (1 ((a 0 0)))

(lcs-with-positions '(x a b y) '(p q a b))
 ⇒ (2 ((a 1 2) (b 2 3)))

(lcs-with-positions '(x a b y) '(p a q b))
 ⇒ (2 ((a 1 1) (b 2 3)))

(lcs-with-positions '(x y) '(p q))
 ⇒ (0 ())
```

`lcs-fold` *a-proc b-proc both-proc seed a b :optional eq-fn* [Function]

{`util.lcs`} A fundamental iterator over the "edit list" derived from two lists *a* and *b*.

A-proc, *b-proc*, *both-proc* are all procedures that take two arguments. The second argument is a intermediate state value of the calculation. The first value is an element only in *a* for *a-proc*, or an element only in *b* for *b-proc*, or an element in both *a* and *b* for *both-proc*. The return value of each procedure is used as the state value of the next call of either one of the procedures. *Seed* is used as the initial value of the state value. The last state value is returned from `lcs-fold`.

The three procedures are called in the following order: Suppose the sequence *a* consists of *a'ca"*, and *b* consists of *b'cb"*, where *a'*, *b'*, *a"*, and *b"* are subsequences, and *c* is the head of the LCS of *a* and *b*. Then *a-proc* is called first on each element in *a'*, *b-proc* is called second on each element in *b'*, then *both-proc* is called on *c*. Afterwards, the process is repeated using *a"* and *b"*.

`lcs-edit-list` *a b :optional eq-fn* [Function]

{`util.lcs`} Calculates 'edit-list' from two lists *a* and *b*, which is the smallest set of commands (additions and deletions) that changes *a* into *b*. This procedure is built on top of `lcs-fold` above.

Returns a list of *hunk*s, which is a contiguous section of additions and deletions. Each hunk consists of a list of directives, which is a form of:

```
(+|- position element)
```

Here's an example. Suppose *a* and *b* are the following lists, respectively.

```
a ≡ ("A" "B" "C" "E" "H" "J" "L" "M" "N" "P")
b ≡ ("B" "C" "D" "E" "F" "J" "K" "L" "M" "R" "S" "T")
```

Then, `(lcs-edit-list a b equal?)` returns the following list.

```
(((- 0 "A"))
 ((+ 2 "D"))
 ((- 4 "H") (+ 4 "F"))
 ((+ 6 "K"))
 ((- 8 "N") (- 9 "P") (+ 9 "R") (+ 10 "S") (+ 11 "T"))
)
```

The result consists of five hunks. The first hunk consists of one directive, `(- 0 "A")`, which means the element `"A"` at the position 0 of list *a* has to be deleted. The second hunk also consists of one directive, `(+ 2 "D")`, meaning the element `"D"` at the position 2 of list *b* has to be added. The third hunk means `"H"` at the position 4 of list *a* should be removed and `"F"` at the position 4 of list *b* should be added, and so on.

If you are familiar with Perl's Algorithm::Diff module, you may notice that this is the same structure that its `diff` procedure returns.

12.67 `util.levenshtein` - Levenshtein edit distance

`util.levenshtein` [Module]

This module provides procedures to calculate edit distance between two sequences. Edit distance is the minimum number of edit operations required to match one sequence to another. Three algorithms are implemented:

Levenshtein distance
: Count deletion of one element, insertion of one element, and substitution of one element.

Damerau-Levenshtein distance
: Besides deletion, insertion and substitution, we allow transposition of adjacent elements.

Restricted edit distance
: Also called optimal string alignment distance. Like Damerau-Levenshtein, but once transposition is applied, no further editing on those elements are allowed.

These algorithms are often used to compare strings, but the procedures in this module can handle any type of sequence (see Section 9.29 [Sequence framework], page 441).

`l-distance` *seq-A seq-B :key elt= cutoff* [Function]
`l-distances` *seq-A seq-Bs :key elt= cutoff* [Function]
`re-distance` *seq-A seq-B :key elt= cutoff* [Function]
`re-distances` *seq-A seq-Bs :key elt= cutoff* [Function]
`dl-distance` *seq-A seq-B :key elt= cutoff* [Function]
`dl-distances` *seq-A seq-Bs :key elt= cutoff* [Function]

{`util.levenshtein`} Calculates Levenshtein distance (`l-*`), restricted edit distance (`re-*`) and Damerau-Levenshtein distance (`dl-*`) between sequences, respectively. Each algorithm comes in two flavors: The singular form `*-distance` takes two sequences, *seq-A* and *seq-B*, and calculates distance between them. The plural form `*-distances` takes a sequence *seq-A* and a list of sequences *seq-Bs*, and calculates distances between *seq-A* and each in *seq-Bs*.

If you need to calculate distances from a single sequence to many sequences, using the plural version is much faster than repeatedly calling the singular version, for the plural version can reuse internal data structures and save allocation and setup time.

Sequences can be any object that satisfy the `<sequence>` protocol (see Section 9.29 [Sequence framework], page 441).

The keyword argument *elt=* is used to compare elements in the sequences. Its default is `eqv?`.

The keyword argument *cutoff* must be, if given, a nonnegative exact integer. Once the possible minimum distance between two sequences becomes greater than this number, the algorithm stops and gives `#f` as the result, and moves on to the next calculation. This is useful when you run the algorithm on large set of sequences and you only need to look for the pairs closer than the certain limit.

In our implementation, Levenshtein is the fastest, Damerau-Levenshtein is the slowest and Restricted edit is somewhere inbetween. If you don't need to take into account of transpositions, use Levenshtein; it counts 2 for `cat -> act`, while other algorithms yield 1 for it. If you need to consider transpositions, choose either `re-` or `dl-`. The catch in `re-` is that it does not satisfy triangular inequality, i.e. for given three sequences X, Y and Z, (Damerau-)Levenshtein distance L always satisfy L(X;Z) <= L(X;Y) + L(Y;Z), but restricted edit distance doesn't guarantee that.

```
(l-distance "cat" "act")  ⇒ 2
(l-distances "cat" '("Cathy" "scathe" "stack")
  :elt= char-ci=?)
  ⇒ (2 3 4)

(re-distance "cat" "act") ⇒ 1

(re-distances "pepper"
  '("peter" "piper" "picked" "peck" "pickled" "peppers")
  :cutoff 4)
  ⇒ (2 2 4 4 #f 1)
```

```
(dl-distance '(a b c d e) '(c d a b e)) ⇒ 4
```

Note: If you pass list of sequences to the second argument of the singular version by accident, you might not get an error immediately because a list is also a sequence.

12.68 `util.match` - Pattern matching

`util.match` [Module]

This module is a port of Andrew Wright's pattern matching macro library. It is widely used in Scheme world, and ported to various Scheme implementations, including Chez Scheme, PLT Scheme, Scheme48, Chicken, and SLIB. It is similar to, but more powerful than Common Lisp's `destructuring-bind`.

This version retains compatibility of the original Wright's macro, except (1) *box* is not supported since Gauche doesn't have one, and (2) structure matching is integrated to Gauche's object system.

We show a list of APIs first, then the table of complete syntax of patterns, followed by examples.

Pattern matching API

`match` *expr clause . . .* [Macro]

{`util.match`} Each *clause* is either one of the followings:

```
(pat body ...)
(pat (=> identifier) body ...)
```

First, the *expr* is matched against *pat* of each clauses. The detailed syntax of the pattern is explained below.

If a matching *pat* is found, the *pattern variables* in *pat* are bound to the corresponding elements in *expr*, then *body . . .* are evaluated. Then `match` returns the value(s) of the last expression of *body . . .*.

If the clause is the second form, *identifier* is also bound to the failure continuation of the *clause*. It is a procedure with no arguments, and when called, it jumps back to the matcher as if the matching of *pat* is failed, and `match` continues to try the rest of clauses. So you can perform extra tests within *body . . .* and if you're not satisfied you can reject the match by calling `(identifier)`. See the examples below for more details.

If no *pat* matches, `match` reports an error.

`match-lambda` *clause . . .* [Macro]

{`util.match`} Creates a function that takes one argument and performs `match` on it, using *clause . . .*. It's functionally equivalent to the following expression:

```
(lambda (expr) (match expr clause ...))
```

Example:

```
(map (match-lambda
       ((item price-per-lb (quantity 'lbs))
        (cons item (* price-per-lb quantity)))
       ((item price-per-lb (quantity 'kg))
        (cons item (* price-per-lb quantity 2.204))))
     '((apple      1.23 (1.1 lbs))
       (orange     0.68 (1.4 lbs))
       (cantaloupe 0.53 (2.1 kg))))
 ⇒ ((apple . 1.353) (orange . 0.952)
          (cantaloupe . 2.4530520000000005))
```

match-lambda* *clause* . . . [Macro]

{util.match} Like match-lambda, but performs match on the list of whole arguments. It's functionally equivalent to the following expression:

```
(lambda expr (match expr clause ...))
```

match-let ((*pat expr*) . . .) *body-expr* . . . [Macro]
match-let *name* ((*pat expr*) . . .) *body-expr* . . . [Macro]
match-let* ((*pat expr*) . . .) *body-expr* . . . [Macro]
match-letrec ((*pat expr*) . . .) *body-expr* . . . [Macro]

{util.match} Generalize let, let*, and letrec to allow patterns in the binding position rather than just variables. Each *expr* is evaluated, and then matched to *pat*, and the bound pattern variables are visible in *body-expr* . . . .

```
(match-let (
             (((ca . cd) ...)   '((a . 0) (b . 1) (c . 2)))
           )
  (list ca cd))
 ⇒ ((a b c) (0 1 2))
```

If you're sick of parenthesis, try match-let1 below.

match-let1 *pat expr body-expr* . . . [Macro]

{util.match} This is a Gauche extension and isn't found in the original Wright's code. This one is equivalent to the following code:

```
(match-let ((pat expr)) body-expr ...)
```

Syntactically, match-let1 is very close to the Common Lisp's destructuring-bind.

```
(match-let1 ('let ((var val) ...) body ...)
            '(let ((a b) (c d)) foo bar baz)
  (list var val body))
 ⇒ ((a c) (b d) (foo bar baz))
```

match-define *pat expr* [Macro]

{util.match} Like toplevel define, but allows a pattern instead of variables.

```
(match-define (x . xs) (list 1 2 3))

x  ⇒ 1
xs ⇒ (2 3)
```

Pattern syntax

Here's a summary of pattern syntax. The asterisk (*) after explanation means Gauche's extension which does not present in the original Wright's code.

```
pat : patvar                      ;; anything, and binds pattern var
    | _                           ;; anything
    | ()                          ;; the empty list
    | #t                          ;; #t
    | #f                          ;; #f
    | string                      ;; a string
    | number                      ;; a number
    | character                   ;; a character
    | 'sexp                       ;; an s-expression
    | 'symbol                     ;; a symbol (special case of s-expr)
    | (pat1 ... patN)             ;; list of n elements
    | (pat1 ... patN . patN+1)    ;; list of n or more
```

```
| (pat1 ... patN patN+1 ooo)   ;; list of n or more, each element
                               ;;   of remainder must match patN+1
| #(pat1 ... patN)             ;; vector of n elements
| #(pat1 ... patN patN+1 ooo)  ;; vector of n or more, each element
                               ;;   of remainder must match patN+1
| ($ class pat1 ... patN)      ;; an object (patK matches in slot order)
| (struct class pat1 ... patN) ;; ditto (*)
| (@ class (slot1 pat1) ...)   ;; an object (using slot names) (*)
| (object class (slot1 pat1) ...) ;; ditto (*)
| (= proc pat)                 ;; apply proc, match the result to pat
| (and pat ...)                ;; if all of pats match
| (or pat ...)                 ;; if any of pats match
| (not pat ...)                ;; if all pats don't match at all
| (? predicate pat ...)        ;; if predicate true and all pats match
| (set! patvar)                ;; anything, and binds setter
| (get! patvar)                ;; anything, and binds getter
| `qp                          ;; a quasi-pattern

patvar : a symbol except _, quote, $, struct, @, object, =, and, or,
         not, ?, set!, get!, quasiquote, ..., ___, ..k, __k.


ooo : ...                      ;; zero or more
    | ___                      ;; zero or more
    | ..k                      ;; k or more, where k is an integer.
                               ;;   Example: ..1, ..2 ...
    | __k                      ;; k or more, where k is an integer.
                               ;;   Example: __1, __2 ...
```

- A bare symbol is a "pattern variable"; it matches anything, and the matched part of the expression is bound to the symbol. The following symbols have special meanings and cannot be used as a pattern variable: `_`, `quote`, `$`, `struct`, `@`, `object`, `=`, `and`, `or`, `not`, `?`, `set!`, `get!`, `quasiquote`, `...`, `___`, and `..k` and `__k` where *k* is an integer.
- A symbol `_` matches anything, without binding a pattern variable. It can be used to show "don't care" placeholder.
- Literals such as emptylist, booleans, strings, numbers, characters and keywords match the same object (in the sense of `equal?`).
- Quoted expression matches the same expression (in the sense of `equal?`). You can use a quoted symbol to match the symbol itself.
- A keyword, without a quote, used to match the same keyword object. Since we're in the process of unifying keywords and symbols, the user is recommended to write keywords with a quote in a pattern in order to match the keyword in the input. See Section 6.8.1 [Keyword and symbol integration], page 141, for the details.
- A list and a vector in general match a list or a vector whose elements matches the elements in the pattern recursively, unless the first element of the list is one of the special symbols listed above, it has a special meaning.

  As a special case, the last element of a vector or a list can be followed by a symbol `...`. In that case, the pattern just before the symbol `...` can be applied repeatedly until it consumes all the elements in the given expression. A symbol `___` can be used in place of `...`; it is useful when you want to produce a pattern by syntax-rules macro.

For a list pattern, you can also use a symbol `..1`, `..2`, ..., which specifies the minimum number of repetition.

- `($ class pat1 ...)` matches an instance of a class `class`. Each pattern `pat1 ...` matches each value of slots, in order of `(class-slots class)` by default. (Records are exception; they match the same order as their default constructor since 0.9.6.)

  `(struct class pat1 ...)` has the same meaning. Although the original Wright's code doesn't have `struct`, PLT Scheme has it in its extended match feature, and it is more descriptive.

  This is an adaptation of the original feature that can match structures. It is useful to match a simple instance that you know the order of slots; for example, a simple record created by `define-record-type` (see Section 9.26 [Record types], page 433) would be easy to match by positioned values.

  If the instance's class uses inheritances, it is a bit difficult to match by positions. You can use `@` or `object` pattern below to match using slot names.
- `(object class (slot1 pat1) ...)` matches an instance of a class `class` whose value of *slot1* ... matches *pat1* .... This is Gauche's extension. `@` can be used in place of `object`, but `object` is recommended because of descriptiveness.
- `(= proc pat)` first applies *proc* to the corresponding expression, then match the result with *pat*.
- `(and pat ...)`, `(or pat ...)`, and `(not pat ...)` are boolean operations of patterns.
- `(? predicate pat ...)` first applies a predicate to the corresponding expression, and if it returns true, applies each `pat ...` to the expression.
- `(set! patvar)` matches anything, and binds an one-argument procedure to a pattern variable *patvar*. If the procedure is called, it replaces the value of matched pattern for the given argument.
- `(get! patvar)` matches anything, and binds a zero-argument procedure to a pattern variable *patvar*. If the procedure is called, it returns the matched value.
- `` `qp `` is a quasipattern. *qp* is quoted, in the sense that it matches itself, *except* the pattern that is unquoted. (Don't confuse quasipattern to quasiquote, though the functions are similar. Quasiquote turns off evaluation except unquoted subtree. Quasiquote turns off the special pattern syntax except unquoted subtree. See the examples below).

Pattern examples

A simple structure decomposition:

```
(match '(0 (1 2) (3 4 5))
  [(a (b c) (d e f))
   (list a b c d e f)])
 ⇒ (0 1 2 3 4 5)
```

Using predicate patterns:

```
(match 123
  [(? string? x) (list 'string x)]
  [(? number? x) (list 'number x)])
 ⇒ (number 123)
```

Extracting variables and expressions from `let`. Uses repetition and predicate patterns:

```
(define let-analyzer
  (match-lambda
    [('let (? symbol?)
           ((var expr) ...)
```

```
          body ...)
       (format "named let, vars=~s exprs=~s" var expr)]
      [('let ((var expr) ...)
          body ...)
       (format "normal let, vars=~s exprs=~s" var expr)]
      [_
       (format "malformed let")]))

(let-analyzer '(let ((a b) (c d)) e f g))
 ⇒ "normal let, vars=(a c) exprs=(b d)"

(let-analyzer '(let foo ((x (f a b)) (y (f c d))) e f g))
 ⇒ "named let, vars=(x y) exprs=((f a b) (f c d))"

(let-analyzer '(let (a) b c d))
 ⇒ "malformed let"
```

Using `=` function application. The pattern variable *m* is matched to the result of application of the regular expression.

```
(match "gauche-ref.texi"
  ((? string? (= #/(.*)\.([^.]+)$/ m))
   (format "base=~a suffix=~a" (m 1) (m 2))))
 ⇒ "base=gauche-ref suffix=texi"
```

An example of quasipattern. In the first expression, the pattern except `value` is quoted, so the symbols `the`, `answer`, and `is` are not pattern variables but literal symbols. The second expression shows that; input symbol `was` does not match the literal symbol `is` in the pattern. If we don't use quasiquote, all symbols in the pattern are pattern variables, so any four-element list matches as the third expression shows.

```
(match '(the answer is 42)
  [`(the answer is ,value) value]
  [else #f])
 ⇒ 42

(match '(the answer was 42)
  [`(the answer is ,value) value]
  [else #f])
 ⇒ #f

(match '(a b c d)
  [(the answer is value) value]
  [else #f])
 ⇒ d
```

An example of matching records. The following code implements "rotation" operation to balance a red-black tree.

```
(define-record-type T #t #t
  color left value right)

(define (rbtree-rotate t)
  (match t
    [(or ($ T 'B ($ T 'R ($ T 'R a x b) y c) z d)
         ($ T 'B ($ T 'R a x ($ T 'R b y c)) z d)
```

```
            ($ T 'B a x ($ T 'R ($ T 'R b y c) z d))
            ($ T 'B a x ($ T 'R b y ($ T 'R c z d))))
         (make-T 'R (make-T 'B a x b) y (make-T 'B c z d))]
        [else t]))
```

12.69 `util.record` - SLIB-compatible record type

`util.record` [Module]

This module provides a Guile and SLIB compatible record type API. It is built on top of Gauche's object system.

See also Section 9.26 [Record types], page 433, which provides a convenience macro `define-record-type`.

`make-record-type` *type-name field-names* [Function]

{`util.record`} Returns a new class which represents a new record type. (It is what is called *record-type descriptor* in SLIB). In Gauche, the new class is a subclass of `<record>` (see Section 9.26 [Record types], page 433).

type-name is a string that is used for debugging purposes. It is converted to a symbol and set as the name of the new class. *field-names* is a list of symbols of the names of fields. Each field is implemented as a slot of the new class.

In the following procedures, *rtd* is the record class created by `make-record-type`.

`record-constructor` *rtd :optional field-names* [Function]

{`util.record`} Returns a procedure that constructs an instance of the record type of given *rtd*. The returned procedure takes exactly as many arguments as *field-names*, which defaults to `'()`. Each argument sets the initial value of the corresponding field in *field-names*.

`record-predicate` *rtd* [Function]

{`util.record`} Returns a procedure that takes one argument, which returns `#t` iff the given argument is of type of *rtd*.

`record-accessor` *rtd field-name* [Function]

{`util.record`} Returns an accessor procedure for the field named by *field-name* of type *rtd*. The accessor procedure takes an instance of *rtd*, and returns the value of the field.

`record-modifier` *rtd field-name* [Function]

{`util.record`} Returns a modifier procedure for the field named by *field-name* of type *rtd*. The modifier procedure takes two arguments, an instance of *rtd* and a value, and sets the value to the specified field.

```
(define rtd (make-record-type "my-record" '(a b c)))

rtd ⇒ #<class my-record>

(define make-my-record (record-constructor rtd '(a b c)))

(define obj (make-my-record 1 2 3))

obj ⇒ #<my-record 0x819d9b0>

((record-predicate? rtd) obj)  ⇒ #t

((record-accessor rtd 'a) obj) ⇒ 1
```

```
((record-accessor rtd 'b) obj) ⇒ 2
((record-accessor rtd 'c) obj) ⇒ 3

((record-modifier rtd 'a) obj -1)

((record-accessor rtd 'a) obj) ⇒ -1
```

12.70 `util.relation` - Relation framework

`util.relation` [Module]

Provides a set of common operations for relations.

Given set of values S1, S2, ..., Sn, a relation R is a set of tuples such that the first element of a tuple is from S1, the second from S2, ..., and the n-th from Sn. In another word, R is a subset of Cartesian product of S1, ..., Sn. (The definition, as well as the term *relation*, is taken from the Codd's 1970 paper, "A Relational Model of Data for Large Shared Data Banks", in CACM 13(6) pp.377–387.)

This definition can be applied to various datasets: A set of Gauche object system instances is a relation, if you view each instance as a tuple and each slot value as the actual values. A list of lists can be a relation. A stream that reads from CSV table produces a relation. Thus it would be useful to provide a module that implements generic operations on relations, no matter how the actual representation is.

From the operational point of view, we can treat any datastructure that provides the following four methods; `relation-rows`, which retrieves a collection of tuples (rows); `relation-column-names`, `relation-accessor`, and `relation-modifier`, which provide the means to access meta-information. All the rest of relational operations are built on top of those primitive methods.

A concrete implementation of relation can use duck typing, i.e. it doesn't need to inherit a particular base class to use the relation methods. However, for the convenience, a base class `<relation>` is provided in this module. It works as a mixin class—a concrete class typically wants to inherit `<relation>` and `<collection>` or `<sequence>`. Check out the sample implementations in the `lib/util/relation.scm` in the source tree, if you're curious.

This module is still under development. The plan is to build useful relational operations on top of the common methods.

Basic class and methods

`<relation>` [Class]

{`util.relation`} An abstract base class of relations.

`relation-column-names` (*r* <*relation*>) [Method]

{`util.relation`} A subclass must implement this method. It should return a sequence of names of the columns. The type of column names is up to the relation; we don't place any restriction on it, as far as they are different each other in terms of `equal?`.

`relation-accessor` (*r* <*relation*>) [Method]

{`util.relation`} A subclass must implement this method. It should return a procedure that takes two arguments, a row from the relation *r* and a column name, and returns the value of the specified column.

`relation-modifier` (*r* <*relation*>) [Method]

{`util.relation`} A subclass must implement this method. It should returns a procedure that takes three arguments, a row from the relation *r*, a column name, and a value to set.

If the relation is read-only, this method returns `#f`.

relation-rows (*r* <*relation*>) [Method]
{util.relation} A subclass must implement this method. It should return the underlying instance of `<collection>` or its subclass (e.g. `<sequence>`)

The rest of method are built on top of the above four methods. A subclass of `<relation>` may overload some of the methods below for better performance, though.

relation-column-name? (*r* <*relation*>) *column* [Method]
{util.relation} Returns true iff *column* is a valid column name for the relation *r*.

relation-column-getter (*r* <*relation*>) *column* [Method]
relation-column-setter (*r* <*relation*>) *column* [Method]
{util.relation} Returns a procedure to access the specified column of a row from the relation *r*. `Relation-column-getter` should return a procedure that takes one argument, a row. `Relation-column-setter` should return a procedure that takes two arguments, a row and a new value to set.

If the relation is read-only, `relation-column-setter` returns `#f`.

relation-ref (*r* <*relation*>) *row column :optional default* [Method]
{util.relation} *Row* is a row from the relation *r*. Returns value of the *column* in *row*. If *column* is not a valid column name, *default* is returned if it is given, otherwise an error is signaled.

relation-set! (*r* <*relation*>) *row column value* [Method]
{util.relation} *Row* is a row from the relation *r*. Sets *value* as the value of *column* in *row*. This may signal an error if the relation is read-only.

relation-column-getters (*r* <*relation*>) [Method]
relation-column-setters (*r* <*relation*>) [Method]
{util.relation} Returns full list of getters and setters. Usually the default method is sufficient, but the implementation may want to cache the list of getters, for example.

relation-coercer (*r* <*relation*>) [Method]
{util.relation} Returns a procedure that coerces a row into a sequence. If the relation already uses a sequence to represent a row, it can return row as is.

relation-insertable? (*r* <*relation*>) [Method]
{util.relation} Returns true iff new rows can be inserted to the relation *r*.

relation-insert! (*r* <*relation*>) *row* [Method]
{util.relation} Insert a row *row* to the relation *r*.

relation-deletable? (*r* <*relation*>) [Method]
{util.relation} Returns true iff rows can be deleted from the relation *r*.

relation-delete! (*r* <*relation*>) *row* [Method]
{util.relation} Deletes a row *row* from the relation *r*.

relation-fold (*r* <*relation*>) *proc seed column ...* [Method]
{util.relation} Applies *proc* to the values of *column* ... of each row, passing *seed* as the state value. That is, for each row in *r*, *proc* is called as follows:

```
(proc v_0 v_1 ... v_i seed)

 where v_k = (relation-ref r row column_k)
```

The result of the call becomes a new seed value, and the final result is returned from *relation-fold*.

For example, if a relation has a column named `amount`, and you want to sum up all of them in a relation *r*, you can write like this:

```
(relation-fold r + 0 'amount)
```

Concrete classes

`<simple-relation>` [Class]
{`util.relation`}

`<object-set-relation>` [Class]
{`util.relation`}

12.71 `util.stream` - Stream library

`util.stream` [Module]
This module provides a library of lazy streams, including the functions and syntaxes defined in srfi-40 and srfi-41, the latter of which became a part of R7RS large (as `(scheme stream)`,

12.71.1 Stream primitives

`stream?` *obj* [Function]
[R7RS stream] {`util.stream`} Returns `#t` iff *obj* is a stream created by a procedure of `util.stream`.

`stream-null` [Variable]
[R7RS stream] {`util.stream`} The singleton instance of NULL stream.

`stream-cons` *object stream* [Macro]
[R7RS stream] {`util.stream`} A fundamental constructor of a stream. Adds *object* to the head of a *stream*, and returns a new stream.

`stream-null?` *obj* [Function]
[R7RS stream] {`util.stream`} Returns `#t` iff *obj* is the null stream.

`stream-pair?` *obj* [Function]
[R7RS stream] {`util.stream`} Returns `#t` iff *obj* is a non-null stream.

`stream-car` *s* [Function]
[R7RS stream] {`util.stream`} Returns the first element of the stream *s*.

`stream-cdr` *s* [Function]
[R7RS stream] {`util.stream`} Returns the remaining elements of the stream *s*, as a stream.

`stream-delay` *expr* [Macro]
[SRFI-40] {`util.stream`} Returns a stream which is a delayed form of *expr*.

As a rule of thumb, any stream-producing functions should wrap the resulting expression by `stream-delay`. (Or you can use `stream-lambda`, `stream-let` or `stream-define`, described below.)

`stream-lambda` *formals body body2 . . .* [Macro]
[R7RS stream] {`util.stream`} A convenience macro to create a function that returns a stream. Effectively, `(stream-lambda formals body body2 ...)` is the same as `(lambda formals (stream-delay body body2 ...))`.

12.71.2 Stream constructors

stream *obj* . . . [Function]

[SRFI-40] {util.stream} Returns a new stream whose elements are *obj* . . . .

Note: This differs from srfi-41's (scheme.stream's) stream, which is a macro so that arguments are lazily evaluated. Srfi-41's stream is provided as stream+ in this module.

```
(stream 1 2 3))       ⇒ a stream that contains (1 2 3)
(stream 1 (/ 1 0))) ⇒ error
```

stream+ *expr* . . . [Macro]

{util.stream} Returns a new stream whose elements are the result of *expr* . . . .

This is the same as srfi-41(scheme.stream)'s stream. Each *expr* isn't evaluated until it is accessed.

```
(define s (stream+ 1 (/ 1 0)))  ;; doesn't yield an error

(stream-car s)  ⇒ 1

(stream-cadr s) ⇒ error
```

stream-unfold *f p g seed* [Function]

[R7RS stream] {util.stream} Creates a new stream whose element is determined as follows:

- A "go" predicate *p* is called on the current seed value. If it yields #f, the stream terminates.
- Otherwise, (f s) is the element of the stream, and (g s) becomes the next seed value, where s is the current seed value. The initial seed value is given by *seed*.

Note: Unfortunately, the order of arguments differs from other *-unfold procedures, which takes p f g (predicate, value-generator and seed-generator). Furthermore, the predicate is stop-predicate (returning true stops iteration).

```
(stream->list
 (stream-unfold integer->char (cut < <> 58) (cut + 1 <>) 48))
 ⇒ (#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9)
```

stream-unfoldn *f seed n* [Function]

[SRFI-40] {util.stream} Creates *n* streams related each other, whose contents are generated by *f* and *seed*.

The *f* is called with the current seed value, and returns *n*+1 values:

```
(f seed)
  => seed result_0 result_1 ... result_n-1
```

The first value is to be the next seed value. *Result_k* must be one of the following forms:

(val) *val* will be the next car of *k*-th stream.

#f No new information for *k*-th stream.

() The end of *k*-th stream has been reached.

The following example creates two streams, the first one produces an infinite series of odd numbers and the second produces evens.

```
gosh> (define-values (s0 s1)
        (stream-unfoldn (lambda (i)
                          (values (+ i 2)          ;; next seed
                                  (list i)         ;; for the first stream
```

```
                                  (list (+ i 1)))) ;; for the second stream
                          0 2))
#<undef>
gosh> (stream->list (stream-take s0 10))
(0 2 4 6 8 10 12 14 16 18)
gosh> (stream->list (stream-take s1 10))
(1 3 5 7 9 11 13 15 17 19)
```

`stream-unfolds` *f seed* [Function]

[R7RS stream] {`util.stream`} Like `stream-unfoldn`, but the number of created streams is determined by the number fo return values from *f*. See `stream-unfoldn` above for the details.

`stream-constant` *obj ...* [Function]

[R7RS stream] {`util.stream`} Returns an infinite stream that repeats *obj* ....

```
(stream->list 10 (stream-constant 1 2))
  ⇒ (1 2 1 2 1 2 1 2 1 2)
```

`make-stream` *n :optional init* [Function]

{`util.stream`} Creates a new stream of *n* elements of *init*. If *init* is omitted, `#f` is used. Specifying a negative number to *n* creates an infinite stream.

`stream-tabulate` *n init-proc* [Function]

{`util.stream`} Creates a new stream of *n* elements. The *k*-th element is obtained by applying *init-proc* to *k*. Specifying a negative number to *n* creates an infinite stream.

`stream-iota` *:optional count start step* [Function]

{`util.stream`} Creates a new stream of numbers, starting from *start* and incrementing *step*. The length of stream is maximum integer not greater than nonnegative real number *count*. The default values of `count`, *start* and *step* are `+inf.0`, 0 and 1, respectively.

If *start* and *step* are exact, and *count* is exact or infinite, a sequence of exact numbers are created. Otherwise, a sequence of inexact numbers are created.

`stream-range` *start :optional end step* [Function]

[R7RS stream] {`util.stream`} Creates a new stream of real numbers, starting from *start* and stops before *end*, stepping by *step*. If *end* is omitted, positive infinity is assumed. If *step* is omitted, `1` is assumed if *end* is greater than *start*, and `-1` if *end* is less than *start*.

The generated numbers are exact if *start* and *step* are exact and *end* is either exact or infinite. Otherwise, inexact numbers are generated.

In R7RS `scheme.stream`, *end* argument is required.

```
(stream->list (stream-range 0 10))
 ⇒ (0 1 2 3 4 5 6 7 8 9)
```

`stream-from` *start :optional step* [Function]

[R7RS stream] {`util.stream`} This is yet another number sequence generator. Generates an infinite sequence whose *i*-th element is `(+ start (* i step))`. If *step* is omitted, 1 is assumed. If both *start* and *step* are exact, exact numbers are generated. Otherwise, inexact numbers are generated.

`stream-iterate` *f seed* [Function]

[R7RS stream] {`util.stream`} Returns a sequence starting from *seed*, and each successive element is calculated by `(f s)` where *s* is the previous element.

```
(stream->list 5 (stream-iterate (cut cons 'x <>) '()))
  ⇒ (() (x) (x x) (x x x) (x x x x))
```

`stream-xcons` *a b* [Function]

{`util.stream`} `(stream-cons b a)`. Just for convenience.

`stream-cons*` *elt . . . stream* [Function]

{`util.stream`} Creates a new stream which appends *elt . . .* before *stream*.

`list->stream` *list* [Function]

[R7RS stream] {`util.stream`} Returns a new stream whose elements are the elements in *list*.

`string->stream` *string :optional tail-stream* [Function]

{`util.stream`} Converts a string to a stream of characters. If an optional *tail-stream* is given, it becomes the tail of the resulting stream.

(stream->list (string->stream "abc" (list->stream '(1 2 3)))) ⇒ (#\a #\b #\c 1 2 3)

`stream-format` *fmt arg . . .* [Function]

{`util.stream`} Returns a stream which is a result of applying `string->stream` to `(format fmt arg ...)`.

`port->stream` *:optional iport reader closer* [Function]

[R7RS stream] {`util.stream`} Creates a stream, whose elements consist of the items read from the input port *iport*. The default *iport* is the current input port. The default *reader* is `read-char`.

The result stream terminates at the point where *reader* returns EOF (EOF itself is not included in the stream). The port won't be closed by default when it reaches EOF.

If *closer* is given, it is called with *iport* as an argument just after *reader* reads EOF. You can close the port with it.

The *reader* and *closer* arguments are Gauche's extension. R7RS `scheme.stream` only takes one optional argument, *iport*.

`iterator->stream` *iter* [Function]

{`util.stream`} A generic procedure to turn an internal iterator *iter* into a stream of iterated results.

The *iter* argument is a procedure that takes two arguments, *next* and *end*, where *next* is a procedure that takes one argument and *end* is a thunk. *Iter* is supposed to iterate over some set and call *next* for each argument, then call *end* to indicate the end of the iteration. Here's a contrived example:

```
(stream->list
 (iterator->stream
  (lambda (next end) (for-each next '(1 2 3 4 5)) (end))))
 ⇒ (1 2 3 4 5)
```

Internally `iterator->stream` uses the "inversion of iterator" technique, so that *iter* only iterates to the element that are needed by the stream. Thus *iter* can iterate over an infinite set. In the following example, *iter* is an infinite loop calling *next* with increasing integers, but only the first 10 elements are calculated because of `stream-take`:

```
(stream->list
 (stream-take
  (iterator->stream
   (lambda (next end)
     (let loop ((n 0)) (next n) (loop (+ n 1)))))
  10))
 ⇒ (0 1 2 3 4 5 6 7 8 9)
```

`stream-of` *elt-expr clause . . .* [Macro]

[R7RS stream] {`util.stream`} Stream comprehension. Returns a stream in which each element is computed by *elt-expr*. The *clause* creates scope of *elt-expr* and controls iterations. Each *clause* can be one of the following forms:

`(x in stream-expr)`
: Iterate over *stream-expr*, binding *x* to each element in each iteration. The variable *x* is visible in successive *clause*s and *elt-expr*

`(x is expr)`
: Bind a variable *x* with the value of *expr*. The variable *x* is visible in successive *clause*s and *elt-expr*.

`expr`
: If *expr* evaluates to `#f`, this iteration is skipped without generating a new element.

The following comprehension generates infinite sequence of pythagorean triples:

```
(define pythagorean-triples
  (stream-of (list a b c)
    (c in (stream-from 3))
    (b in (stream-range 2 c))
    (a in (stream-range 1 b))
    (= (square c) (+ (square b) (square a)))))

(stream->list 5 pythagorean-triples)
  ⇒ ((3 4 5) (6 8 10) (5 12 13) (9 12 15) (8 15 17))
```

12.71.3 Stream binding

`define-stream` (*name . formals*) *body body2 . . .* [Macro]

[R7RS stream] {`util.stream`} A convenient macro to define a procedure that yields a stream. Same as the following form:

```
(define (name . formals)
  (stream-delay
    (let () body body2 ...)))
```

`stream-let` *loop-var* ((*var init*) *. . .*) *body body2 . . .* [Macro]

[R7RS stream] {`util.stream`} A handy macro to write a lazy named-let loop. It is the same as the following:

```
(let loop-var ((var init) ...)
  (stream-delay
    (let () body body2 ...)))
```

`stream-match` *stream-expr clause . . .* [Macro]

[R7RS stream] {`util.stream`} This allows accessing streams via simple pattern matching. The *stream-expr* argument is evaluated and must yield a stream. Each *clause* must be either a form of `(pattern expr)` or `(pattern fender expr)`.

The content of the stream is matched to each *pattern*, which must have one of the following forms:

`()`
: Matches a null stream.

`(p0 p1 ...)`
: Matches a stream that has exactly the same number of elements as the number of pattern elements.

(*p0 p1 . . . . pRest*)
: Matches a stream that has at least the same number of elements as the number of pattern elements except *pRest*. The rest of stream matches with *pRest*.

pRest
: Matches an entire stream.

Each pattern element can be an identifier or a literal underscore. If it is an identifier, it is bound to the matched element while evaluating the corresponding *fender* and *expr*.

If *fender* is present in the *clause*, it is evaluated; if it yields `#f`, the match of the clause fails and next clauses will be tried.

Otherwise, *expr* is evaluated and the result(s) becomes the result(s) of `stream-match`.

Only the elements from the stream that is required to match are accessed.

The following example defines a procedure to count the number of true values in the stream:

```
(define (num-trues strm)
  (stream-match strm
    (() 0)
    ((head . tail) head (+ 1 (num-trues tail)))
    ((_ . tail) (num-trues tail))))

(num-trues (stream #f #f #t #f #t #f #t #t))
  ⇒ 4
```

12.71.4 Stream consumers

These procedures takes stream(s) and consumes its/their elements until one of the streams is exhausted.

`stream-for-each` *func . streams* [Function]

[R7RS stream] {`util.stream`} Applies *func* for each element of *streams*. Terminates if one of *streams* reaches the end.

`stream-fold` *f seed stream* [Function]

[R7RS stream] {`util.stream`} Apply *f* on the current seed value and an element from *stream* to obtain the next seed value, and repeat it until *stream* is exhausted, then returns the last seed value. The initial seed value is given by *seed*.

Note: The argument order of *f* differs from other `*-fold` procedures, e.g. `fold` (see Section 6.6.5 [Walking over lists], page 131) takes the element first, then the seed value.

```
(stream-fold - 0 (stream 1 2 3 4 5))
  ⇒ -15
```

12.71.5 Stream operations

`stream-append` *stream . . .* [Function]

[R7RS stream] {`util.stream`} Returns a new stream which is concatenation of given *streams*.

`stream-concat` *streams* [Function]
`stream-concatenate` *streams* [Function]

[R7RS stream] {`util.stream`} R7RS `scheme.stream` defines `stream-concat`. Gauche had `stream-concatenate`, and keeps it for the backward compatibility. Both are the same.

The *streams* argument is a stream of streams. Returns a new stream that is concatenation of them. Unlike `stream-append`, *streams* can generate infinite streams.

`stream-map` *func stream stream2 . . .* [Function]

[R7RS stream] {`util.stream`} Returns a new stream, whose elements are calculated by applying *func* to each element of *stream* . . . .

`stream-scan` *func seed stream* [Function]

[R7RS stream] {`util.stream`} Returns a stream of *seed*, (*func seed e0*), (*func* (*func seed e0*) *e1*), ..., where *e0*, *e1* ... are the elements from the input *stream*. If *stream* is finite, the result stream has one more elements than the number of elements in the original stream.

```
(stream->list
  (stream-scan xcons '() (stream 1 2 3 4 5)))
 ⇒ (() (1) (2 1) (3 2 1) (4 3 2 1) (5 4 3 2 1))
```

`stream-zip` *stream ...* [Function]

[R7RS stream] {`util.stream`} Returns a new stream whose elements are lists of corresponding elements from input *stream*s. The output stream ends when one of input streams is exhausted.

```
(stream->list
 (stream-zip (stream 1 2 3) (stream 'a 'b 'c 'd)))
 ⇒ ((1 a) (2 b) (3 c))
```

`stream-filter` *pred stream* [Function]

[R7RS stream] {`util.stream`} Returns a new stream including only elements passing *pred*.

`stream-remove` *pred stream* [Function]

{`util.stream`} Returns a new stream including only elements that doesn't satisfy *pred*.

`stream-partition` *pred stream* [Function]

{`util.stream`} Returns two streams, one consists of the elements in *stream* that satisfy *pred*, the other consists of the ones that doesn't satisfy *pred*.

`stream->list` *stream* [Function]
`stream->list` *n stream* [Function]

[R7RS stream] {`util.stream`} Converts a stream to a list. In the first form, all elements from *stream* are taken (thus it never returns if *stream* is infinite). In the second form, at most *n* elements are taken, where *n* must be a nonnegative exact integer.

Note: In usual Scheme conventions, the optional *n* comes after the main argument (*stream*).

`stream->string` *stream* [Function]

{`util.stream`} Converts a stream to a list. All elements of *stream* must be characters, or an error is signaled.

`stream-lines` *stream* [Function]

{`util.stream`} Splits *stream* where its element equals to `#\n`, and returns a stream of splitted streams.

```
(stream->list
 (stream-map stream->string
             (stream-lines (string->stream "abc\ndef\nghi"))))
 ⇒ ("abc" "def" "ghi")
```

`stream=` *elt= stream ...* [Function]

{`util.stream`} Returns true iff each corresponding element of *stream* ... are the same in terms of *elt=*. This procedure won't terminate if any of *stream*s is infinite.

`stream-prefix=` *stream prefix :optional elt=* [Function]

{`util.stream`} Compares initial elements of *stream* against a list *prefix* by *elt=*. Only as many elements of *stream* as *prefix* has are checked.

`stream-caar` *s* [Function]
`stream-cadr` *s* [Function]
...

`stream-cdddar` *s* [Function]
`stream-cddddr` *s* [Function]
{`util.stream`} `(stream-caar s)` = `(stream-car (stream-car s))` etc.

`stream-ref` *stream pos* [Function]
[R7RS stream] {`util.stream`} Returns the *pos*-th element in the stream. *Pos* must be a nonnegative exact integer.

`stream-first` *s* [Function]
`stream-second` *s* [Function]
`stream-third` *s* [Function]
`stream-fourth` *s* [Function]
`stream-fifth` *s* [Function]
`stream-sixth` *s* [Function]
`stream-seventh` *s* [Function]
`stream-eighth` *s* [Function]
`stream-ninth` *s* [Function]
`stream-tenth` *s* [Function]
{`util.stream`} `(stream-first s)` = `(stream-ref s 0)` etc.

`stream-take` *stream count* [Function]
`stream-take-safe` *stream count* [Function]
{`util.stream`} Returns a new stream that consists of the first *count* elements of the given stream. If the given stream has less than *count* elements, the stream returned by `stream-take` would raise an error when the elements beyond the original stream is accessed. On the other hand, the stream returned by `stream-take-safe` will return a shortened stream when the given stream has less than *count* elements.

```
(stream->list (stream-take (stream-iota -1) 10))
 ⇒ (0 1 2 3 4 5 6 7 8 9)

(stream-take (stream 1 2) 5)
 ⇒ stream

(stream->list (stream-take (stream 1 2) 5))
 ⇒ error

(stream->list (stream-take-safe (stream 1 2) 5))
 ⇒ (1 2)
```

Note: srfi-41 (`scheme.stream`) also defines `stream-take`, but the argument order is reversed, and also it allows *stream* to have less than `count` elements.

`stream-drop` *stream count* [Function]
`stream-drop-safe` *stream count* [Function]
{`util.stream`} Returns a new stream that consists of the elements in the given stream except the first *count* elements. If the given stream has less than *count* elements, `stream-drop` returns a stream that raises an error if its element is accessed, and `stream-drop-safe` returns an empty stream.

Note: srfi-41 (`scheme.stream`) also defines `stream-drop`, but the argument order is reversed, and also it allows *stream* to have less than `count` elements.

`stream-intersperse` *stream element* [Function]
{`util.stream`} Returns a new stream in which *element* is inserted between elements of *stream*.

`stream-split` *stream pred* [Function]
{`util.stream`}

`stream-last` *stream* [Function]
{`util.stream`}

`stream-last-n` *stream count* [Function]
{`util.stream`}

`stream-butlast` *stream* [Function]
{`util.stream`}

`stream-butlast-n` *stream count* [Function]
{`util.stream`}

`stream-length` *stream* [Function]
[R7RS stream] {`util.stream`} Returns the number of elements in *stream*. It diverges if *stream* is infinite.

`stream-length>=` *stream n* [Function]
`stream-length=` *stream n* [Function]
{`util.stream`}

`stream-reverse` *stream :optional tail-stream* [Function]
[R7RS stream] {`util.stream`} Returns a stream that returns the elements of *stream* in reverse order. If *tail-stream* is given, it is added after the reversed stream.

The optional argument is Gauche's extension. The description of `reverse` (see Section 6.6.6 [Other list procedures], page 134) for more details.

`stream-count` *pred stream . . .* [Function]
{`util.stream`}

`stream-find` *pred stream* [Function]
{`util.stream`}

`stream-find-tail` *pred stream* [Function]
{`util.stream`}

`stream-take-while` *pred stream* [Function]
[R7RS stream] {`util.stream`}

`stream-drop-while` *pred stream* [Function]
[R7RS stream] {`util.stream`}

`stream-span` *pred stream* [Function]
{`util.stream`}

`stream-break` *pred stream* [Function]
{`util.stream`}

`stream-any` *pred stream . . .* [Function]
{`util.stream`}

`stream-every` *pred stream . . .* [Function]
{`util.stream`}

`stream-index` *pred stream . . .* [Function]
{`util.stream`}

`stream-member` *obj stream :optional elt=* [Function]
`stream-memq` *obj stream* [Function]
`stream-memv` *obj stream* [Function]
{`util.stream`}

`stream-delete` *obj stream :optional elt=* [Function]
{`util.stream`}

`stream-delete-duplicates` *stream :optional elt=* [Function]
{`util.stream`}

`stream-grep` *re stream* [Function]
{`util.stream`}

`write-stream` *stream :optional oport writer* [Function]
{`util.stream`}

12.72 `util.toposort` - Topological sort

`util.toposort` [Module]
Implements topological sort algorithm.

`topological-sort` *graph :optional eqproc* [Function]
{`util.toposort`} *Graph* represents a directed acyclic graph (DAG) by a list of connections, where each connection is the form

```
(<node> <downstream> <downstream2> ...)
```

that means a node `<node>` is connected to other nodes `<downstream>` etc. `<node>` can be arbitrary object, as far as it can be compared by the procedure *eqproc*, which is *eqv?* by default (see Section 6.2.1 [Equality], page 97). Returns a list of `<node>`s sorted topologically.

If the graph contains circular reference, an error is signaled.

12.73 `util.unification` - Unification

`util.unification` [Module]
Implements unification algorithm.

The base API operates on abstract trees, while it is agnostic to the actual representation of the tree. The caller passes comparators and operators along the trees to unify.

We assume the abstract tree has the following structure:

```
Tree : Variable | Value | Tuple
Tuple : { Tree ... }
```

Here, `{...}` just represents a sequence of trees.

A variable can be bound to a tree. A value can only match itself.

To operate on this tree, we need the following comparators and procedures, which the API takes as arguments:

Variable comparator: `var-cmpr`
: A comparator to see if an item is a variable, and also check equality of two variables. It must be hashable. See Section 6.2.4 [Basic comparators], page 103, for the details of comparators.

Value comparator: `val-cmpr`

A comparator to see if an item is a value, and also check equality of two values. Note that if a tree satisfies neither *var-cmpr* nor *val-compr*, it is regarded as a tuple.

Tuple folder: `tuple-fold`

A procedure `(tuple-folde proc seed tuple1 [tuple2])`. This procedure should work like `fold` (see Section 6.6.5 [Walking over lists], page 131) over the elements in the tuple(s). It is only called with either one or two tuples.

Tuple constructor: `make-tuple`

A procedure `(make-tuple proto elements)`, where *proto* is a tuple and *elements* are a list of trees. It must return a new tuple with the given elements, while all other properties are the same as *proto*. This procedure isn't needed by `unify`.

`unify` *a b var-cmpr val-cmpr tuple-fold* [Function]

{`util.unification`} Unify two trees *a* and *b* and returns a substitution dictionary, which is a dictionary that maps variables to its bounded trees.

See the entry of `util.unification` above for the description of *var-cmpr*, *val-cmpr* and *tuple-fold*.

```
(dict->alist (unify '(a 3 (c b)) '(c b (2 e))
                    symbol-comparator
                    number-comparator
                    fold))
 ⇒ ((e . 3) (a . c) (b . 3) (c . 2))
```

As you can see in the example above, a variable may be mapped to another variable, or even to a tree that contains variables. If you apply the substitution to the original tree, you must do it recursively until all the variables in the dictionary is eliminated.

If two trees cannot be unified, `#f` is returned.

```
(unify '(a (a)) '(x x) symbol-comparator number-comparator fold)
 ⇒ #f
```

`unify-merge` *a b var-cmpr val-cmpr tuple-fold make-tuple* [Function]

{`util.unification`} Unify two trees *a* and *b*, and apply the result substitutions to create a new tree eliminating variables.

See the entry of `util.unification` above for the description of *var-cmpr*, *val-cmpr*, *tuple-fold* and *make-tuple*.

```
(unify-merge '(a 3 (c b)) '(c b (2 e))
             symbol-comparator
             number-comparator
             fold
             (^[_ elts] elts))
 ⇒ (2 3 (2 3))
```

If two trees can't be unified, `#f` is returned.

12.74 `www.cgi` - CGI utility

`www.cgi` [Module]

Provides a few basic functions useful to write a CGI script.

In order to write CGI script easily, you may want to use other modules, such as `rfc.uri` (see Section 12.43 [URI parsing and construction], page 738), `text.html-lite` (see Section 12.55

[Simple HTML document construction], page 784) and `text.tree` (see Section 12.61 [Lazy text construction], page 793).

Note: it seems that there is no active formal specification for CGI. See `http://w3c.org/CGI/` for more information.

Metavariables

`cgi-metavariables` *:optional metavariables* [Parameter]

{`www.cgi`} Normally, httpd passes a cgi program various information via environment variables. Most procedures in `www.cgi` refer to them (meta-variables). However, it is sometimes inconvenient to require environment variable access while you're developing cgi-related programs. With this parameter, you can overrides the information of meta-variables.

Metavariables should be a list of two-element lists. Car of each inner list names the variable, and its cadr gives the value of the variable by string.

For example, the following code overrides `REQUEST_METHOD` and `QUERY_STRING` meta-variables during execution of `my-cgi-procedure`. (See Section 9.22 [Parameters], page 411, for the details of `parameterize`).

```
(parameterize ((cgi-metavariables '(("REQUEST_METHOD" "GET")
                                     ("QUERY_STRING" "x=foo"))))
  (my-cgi-procedure))
```

`cgi-get-metavariable` *name* [Function]

{`www.cgi`} Returns a value of cgi metavariable *name*. This function first searches the parameter `cgi-metavariables`, and if the named variable is not found, calls `sys-getenv`.

CGI scripts may want to use `cgi-get-metavariable` instead of directly calling `sys-getenv`; doing so makes reuse of the script easier.

Parameter extraction

`cgi-parse-parameters` *:key :query-string :merge-cookies :part-handlers* [Function]

{`www.cgi`} Parses query string and returns associative list of parameters. When a keyword argument *query-string* is given, it is used as a source query string. Otherwise, the function checks the metavariable `REQUEST_METHOD` and obtain the query string depending on the value (either from stdin or from the metavariable `QUERY_STRING`). If such a metavariable is not defined and the current input port is a terminal, the function prompts the user to type parameters; it is useful for interactive debugging.

If `REQUEST_METHOD` is `POST`, this procedure can handle both `application/x-www-form-urlencoded` and `multipart/form-data` as the enctype. The latter is usually used if the form has file-uploading capability.

When the post data is sent by `multipart/form-data`, each content of the part is treated as a value of the parameter. That is, the content of uploaded file will be seen as one big chunk of a string. The other information, such as the original file name, is discarded. If it is not desirable to read entire file into a string, you can customize the behavior by the *part-handler* argument. The details are explained in the "Handling file uploads" section below.

When a true value is given to *merge-cookies*, the cookie values obtained from the metavariable `HTTP_COOKIE` are appended to the result.

Note that the query parameter may have multiple values, so `cdr` of each element in the result is a list, not an atom. If no value is given to the parameter, `#t` is placed as its value. See the following example:

```
(cgi-parse-parameters
  :query-string "foo=123&bar=%22%3f%3f%22&bar=zz&buzz")
 ⇒ (("foo" "123") ("bar "\"??\"" "zz") ("buzz" #t))
```

`cgi-get-parameter` *name params :key :default :list :convert* [Function]

{`www.cgi`} A convenient function to obtain a value of the parameter *name* from parsed query string *params*, which is the value `cgi-parse-parameters` returns. *Name* should be a string.

Unless true value is given to *list*, the returned value is a scalar value. If more than one value is associated to *name*, only the first value is returned. If *list* is true, the returned value is always a list, even *name* has only one value.

After the value is retrieved, you can apply a procedure to convert the string value to the appropriate type by giving a procedure to the *convert* argument. The procedure must take one string argument. If *list* is true, the convert procedure is applied to each values.

If the parameter *name* doesn't appear in the query, a value given to the keyword argument *default* is returned; the default value of *default* is `#f` if *list* is false, or `()` otherwise.

Output generation

`cgi-header` *:key status content-type location cookies* [Function]

{`www.cgi`} Creates a text tree (see Section 12.61 [Lazy text construction], page 793) for the HTTP header of the reply message. The most simple form is like this:

```
(tree->string (cgi-header))
  ⇒ "Content-type: text/html\r\n\r\n"
```

You can specify alternative content-type by the keyword argument *content-type*. If you want to set cookies to the client, specify a list of cookie strings to the keyword argument *cookies*. You can use `construct-cookie-string` (see Section 12.31 [HTTP cookie handling], page 717) to build such a list of cookie strings.

The keyword argument *location* may be used to generate a `Location:` header to redirect the client to the specified URI. You can also specify the `Status:` header by the keyword argument *status*. A typical way to redirect the client is as follows:

```
(cgi-header :status "302 Moved Temporarily"
            :location target-uri)
```

`cgi-output-character-encoding` *:optional encoding* [Parameter]

{`www.cgi`} The value of this parameter specifies the character encoding scheme (CES) used for CGI output by `cgi-main` defined below. The default value is Gauche's native encoding. If the parameter is set other than the native encoding, `cgi-main` converts the output encoding by `gauche.charconv` module (see Section 9.4 [Character code conversion], page 339).

Convenience procedures

`cgi-main` *proc :key on-error merge-cookies output-proc part-handlers* [Function]

{`www.cgi`} A convenient wrapper function for CGI script. This function calls `cgi-parse-parameters`, then calls *proc* with the result of `cgi-parse-parameters`. The keyword argument *merge-cookies* is passed to `cgi-parse-parameters`.

proc has to return a tree of strings (see Section 12.61 [Lazy text construction], page 793), including the HTTP header. `cgi-main` outputs the returned tree to the current output port by `write-tree`, then returns zero.

If an error is signaled in *proc*, it is caught and an HTML page reporting the error is generated. You can customize the error page by providing a procedure to the *on-error* keyword argument. The procedure takes an `<condition>` object (see Section 6.20.4 [Conditions], page 212), and has to return a tree of string for the error reporting HTML page, including an HTTP header.

When output the result, `cgi-main` refers to the value of the parameter `cgi-output-character-encoding`, and converts the character encoding if necessary.

The output behavior of `cgi-main` can be customized by a keyword argument *output-proc*; if it is given, the text tree (either the normal return value of *proc*, or an error page constructed by the error handler) is passed to the procedure given to *output-proc*. The procedure is responsible to format and output a text to the current output port, including character conversions, if necessary.

The keyword argument *part-handlers* are simply passed to `cgi-parse-parameters`, by which you can customize how the file uploads should be handled. See the "Handling file uploads" section below for the details.

If you specify to use temporary file(s) by it, `cgi-main` makes sure to clean up them whenever *proc* exits, even by error. See `cgi-add-temporary-file` below to utilize this feature for other purpose.

Before calling *proc*, `cgi-main` changes the buffering mode of the current error port to `:line` (See `port-buffering` in Section 6.22.3 [Common port operations], page 218, for the details about the buffering mode). This makes the error output easier for web servers to capture.

The following example shows the parameters given to the CGI program.

```
#!/usr/local/bin/gosh

(use text.html-lite)
(use www.cgi)

(define (main args)
  (cgi-main
    (lambda (params)
      `(,(cgi-header)
        ,(html-doctype)
        ,(html:html
          (html:head (html:title "Example"))
          (html:body
           (html:table
            :border 1
            (html:tr (html:th "Name") (html:th "Value"))
            (map (lambda (p)
                   (html:tr
                    (html:td (html-escape-string (car p)))
                    (html:td (html-escape-string (x->string (cdr p))))))
                 params))))
       ))))
```

`cgi-add-temporary-file` *filename* [Function]

{`www.cgi`} This is supposed to be called inside *proc* of `cgi-main`. It registers *filename* as a temporary file, which should be unlinked when *proc* exits. It is a convenient way to ensure that your cgi script won't leave garbages even if it throws an error. It is OK in *proc* to unlink or rename *filename* after calling this procedure.

`cgi-temporary-files` [Parameter]

{`www.cgi`} Keeps a list of filenames registered by `cgi-add-temporary-file`.

Handling file uploads

As explained in `cgi-parse-parameters` above, file uploads are handled transparently by default, taking the file content as the value of the parameter. Sometimes you might want to change this behavior, for the file might be quite big and you don't want to keep around a huge chunk of a string in memory. It is possible to customize handling of file uploads of `cgi-parse-parameters` and `cgi-main` by *part-handlers* argument. (The argument is only effective for the form data submitted by `multipart/form-data` enctype)

The *part-handlers* argument is, if given, a list of lists; each inner list is a form of `(name-pattern action kv-list ...)`. Each uploaded file with a matching parameter name with *name-pattern* is handled according to *action*. (Here, a parameter name is the 'name' attribute given to the `input` element in the submitted form, not the name of the uploaded file).

Name-pattern must be either a list of string (matches one of them), a regexp, or `#t` (matches anything).

Action must be either one of the followings:

`#f` Default action, i.e. the content of the uploaded file is turned into a string and becomes the value of the parameter.

`ignore` The uploaded content is discarded.

`file` The uploaded content is saved in a temporary file. The value of the parameter is the pathname of the temporary file.

For this action, you can write an entry like `(name-pattern file prefix)`, to specify the prefix of the pathname of the temporary file. For example, if you specify `("image" file "/var/mycgi/incoming/img")`, the file uploaded as `"image"` parameter will be stored as something like `/var/mycgi/incoming/img49g2Ua`.

The application should move the temporary file to appropriate location; if you're using `cgi-main`, the temporary files created by this action will be unlinked when `cgi-main` exits.

`file+name`

Like `file`, but the value of the parameter is a list of temporary filename and the filename passed by the client. It is useful if you want to use client's filename (but do not blindly assume the client sends a valid pathname; for example, you shouldn't use it to rename the uploaded file without validating it).

`procedure`

In this case, *procedure* is called to handle the uploaded contents. It is called with four arguments: `(procedure name filename part-info iport)`.

Name is the name of the parameter. *Filename* is the name of the original file (pathname in the client). *Part-info* is a `<mime-part>` object that keeps information about this mime part, and *iport* is where the body can be read from. For the details about these arguments, see Section 12.39 [MIME message handling], page 730; you might be able to use procedures provided by `rfc.mime`, such as `mime-retrieve-body`, to construct your own procedure.

If you create a temporary file in *procedure*, you can call `cgi-add-temporary-file` to make sure it is removed even if an error occurs during cgi processing.

If *kv-list* is given after *action*, it must be a keyword-value list and further modifies action. The following keywords are supported.

`:prefix` Valid only if *action* is either `file` or `file+name`. Specifies the prefix of the temporary file. If you give `:prefix "/tmp/foo"`, for example, the file is saved as something like `/tmp/fooxAgjeQ`.

`:mode` Valid only if *action* is either `file` or `file+name`. Specifies the mode of the temporary file in unix-style integer. By default it is `#o600`.

Note that the parameters that are not file uploads are not the subject of *part-handlers*; such parameter values are always turned into a string.

Here's a short example. Suppose you have a form like this:

```
<form enctype="multipart/form-data" method="POST" action="mycgi.cgi">
<input type="file" name="imagefile" />
<input type="text" name="description" />
<input type="hidden" name="mode" value="normal" />
</form>
```

If you use `cgi-parse-parameters` in `mycgi.cgi` without *part-handlers* argument, you'll get something like the following as the result. (The actual values depend on how the web client filled the form).

```
(("imagefile" #*".....(image file content as a string)....")
 ("description" "my image")
 ("mode" "normal"))
```

If you pass `'(("imagefile" file :prefix "/tmp/mycgi"))` to *part-handlers* instead, you might get something like the following, with the content of uploaded file saved in `/tmp/mycgi7gq0B`

```
(("imagefile" "/tmp/mycgi7gq0B")
 ("description" "my image")
 ("mode" "normal"))
```

If you use a symbol `file+name` instead of `file` above, you'll get something like `("/tmp/mycgi7gq0B" "logo.jpg")` as the value of `"imagefile"`, where `"logo.jpg"` is the client-side filename. (Note: the client can send any string as the name of the file, so *never* assume it is a valid pathname).

12.75 `www.cgi.test` - CGI testing

`www.cgi.test` [Module]

This module defines a useful procedures to test CGI script. The test actually runs the named script, with specified environment variable settings, and retrieve the output. Your test procedure then examine whether the output is as expected or not.

`cgi-test-environment-ref` *envvar-name* [Function]

`(setter cgi-test-environment-ref)` *envvar-name value* [Function]

{`www.cgi.test`} The module keeps a table of default values of environment variables with which the cgi script will be run. These procedures allow the programmer to get/set those default values.

Note that you can override these default values and/or pass additional environment variables for each call of cgi script. The following environment variables are set by default.

Name	Value
`SERVER_SOFTWARE`	`cgitest/1.0`
`SERVER_NAME`	`localhost`
`GATEWAY_INTERFACE`	`CGI/1.1`
`SERVER_PROTOCOL`	`HTTP/1.1`
`SERVER_PORT`	`80`
`REQUEST_METHOD`	`GET`
`REMOTE_HOST`	`remote`
`REMOTE_ADDR`	`127.0.0.1`

`call-with-cgi-script` *script proc :key (environment ()) (parameters #f)* [Function]

{`www.cgi.test`} Runs a script with given environment, and calls *proc* with one argument, an input port which is connected to the pipe of script's standard output. The argument *script* should be a list of program name and its arguments. Each element are passed to `x->string` first to stringify. The script is run under the environment given by *environment* variable and the default test environment described above. The *environment* argument must be an associative list, in which each key (`car`) is the name of the environment variable and its `cdr` is the value. Both are passed to `x->string` first. If the same environment variable appears in *environment* and the default test environment, the one in *environment* is used. Additionally, if an associative list is given to the *parameters* argument, a query string is built from it and passed the script. The actual method to pass the query string depends on the value of `REQUEST_METHOD` environment variable in the setting. If `REQUEST_METHOD` is either `GET` or `HEAD`, the query string is put in an environment variable `QUERY_STRING`. If it is `POST`, the query string is fed to the standard input of the script. In the latter case, `CONTENT_TYPE` is set to `application/x-www-form-urlencoded` and `CONTENT_LENGTH` are set to the length of `QUERY_STRING` automatically. If `REQUEST_METHOD` is other values, *parameters* is ignored. You can bypass this mechanism and set up environment variable `QUERY_STRING` directly, if you wish.

`run-cgi-script->header&body` *script reader :key environment parameters* [Function]

{`www.cgi.test`} A convenient wrapper of `call-with-cgi-script`. The *script*, *environment* and *parameters* are passed to `call-with-cgi-script` as they are. The output of the script is parsed by `run-cgi-script->header&body`. First, the RFC2822 header fields are parsed by `rfc822-read-headers` (see Section 12.29 [RFC822 message parsing], page 712). Then, the *reader* is called with an input port which is piped to the script's output. `Run-cgi-script->header&body` returns two values, the list of headers (as parsed by `rfc822-read-headers`), and the return value of *reader*.

`run-cgi-script->sxml` *script :key environment parameters* [Function]

{`www.cgi.test`} This is a procedure that uses `ssax:xml->sxml` (see Section 12.46 [Functional XML parser], page 747) as the *reader* in `run-cgi-script->header&body`. Useful when you're testing a cgi script that produces well-formed HTML and/or XML document.

`run-cgi-script->string` *script :key environment parameters* [Function]
`run-cgi-script->string-list` *script :key environment parameters* [Function]

{`www.cgi.test`} These procedures use `port->string` and `port->string-list` (see Section 6.22.7.4 [Input utility functions], page 231) as the *reader* in `run-cgi-script->header&body`, respectively.

An example:

```
(run-cgi-script->string-list "bbs.cgi"
                             :environment '((REMOTE_ADDR . "12.34.56.78"))
                             :parameters '((command . "view")
                                           (page . 1234)))
```

12.76 `www.css` - CSS parsing and construction

`www.css` [Module]

This module provides tools to convert between S-expression and CSS.

The S-expression CSS (SxCSS) is a convenient way to manipulate CSS in Scheme.

For example, the following CSS and SxCSS are equivalent, and can be converted back and forth:

CSS:

```
body { padding-left: 11em;
       font-family: Georgia, "Times New Roman", Times, serif;
       color: purple;
       background-color: #d8da3d }
ul.navbar li { background: white;
               margin: 0.5em 0;
               padding: 0.3em;
               border-right: 1em solid black }
ul#spec > a { text-decoration: none }
a:visited { color: purple !important }
```

SxCSS:

```
((style-rule body
   (padding-left (11 em))
   (font-family (:or Georgia "Times New Roman" Times serif))
   (color purple)
   (background-color (color "d8da3d")))
 (style-rule ((ul (class navbar)) li)
   (background white)
   (margin #((0.5 em) 0))
   (padding (0.3 em))
   (border-right #((1 em) solid black)))
 (style-rule ((ul (id spec)) > a) (text-decoration none))
 (style-rule (a (: visited)) (color purple !important)))
```

See the "CSS in S-expression" section below for the complete specification.

Constructing CSS

`construct-css` *sxcss :optional oport* [Function]

{`www.css`} Take SxCSS and writes out CSS to the given port, defaulted to the current output port.

Parsing CSS

`parse-css` *:optional iport* [Function]

{`www.css`} Read CSS from the given port, defaulted to the current input port, and returns SxCSS.

When it encounters unparsable CSS (either a malformed CSS, or unsupported syntax), it emits a warning message, ignore the unparsable part and tries to continue reading the rest.

NB: Currently we don't handle `@charset` directive; we assume the text is already in the port's encoding. We may support it in future versions.

`parse-css-file` *file :key encoding* [Function]

{`www.css`} Read the CSS text from the given file and parse it using `parse-css`. Again, we don't handle `@charset` directive yet, and you have to pass `encoding` argument if the CSS text isn't in the Gauche's native character encoding.

`parse-css-selector-string` *str* [Function]

{`www.css`} This parses the selector part of the CSS.

```
(parse-css-selector-string "ul li.item span#foo")
```

```
    ⇒ (ul (li (class item)) (span (id foo)))

  (parse-css-selector-string "h1,h2")
    ⇒ (:or h1 h2)
```

CSS in S-expression

The following is the complete rules of SxCSS syntax.

```
<sxcss>      : ({<style-rule> | <at-rule>} ...)

<style-rule> : (style-rule <pattern> <declaration> ...)
             | (style-decls <declaration> ...)

<pattern>   : <selector> | (:or <selector> ...)
<selector>  : <simple-selector>
            | <chained-selector>
<chained-selector> : (<simple-selector> . (<op>? . <chained-selector>))
<op>        : > | + | ~
<simple-selector> : <element-name>
            | (<element-name> <option> ...)
<option>    : (id <name>)                          ; E#id
            | (class <ident>)                      ; E.class
            | (has <ident>)                        ; E[attrib]
            | (= <ident> <attrib-value>)           ; E[attrib=val]
            | (~= <ident> <attrib-value>)          ; E[attrib~=val]
            | (:= <ident> <attrib-value>)          ; E[attrib|=val]
            | (*= <ident> <attrib-value>)          ; E[attrib*=val]
            | (^= <ident> <attrib-value>)          ; E[attrib^=val]
            | ($= <ident> <attrib-value>)          ; E[attrib$=val]
            | (:not <negation-arg>)                ; E:not(s)
            | (: <ident>)                          ; E:pseudo-class
            | (: (<fn> <ident> ...))               ; E:pseudo-class(arg)
            | (:: <ident>)                         ; E::pseudo-element
<element-name> : <ident> | *
<attrib-value> : <ident> | <string>
<negation-arg> | <element-name> | * | <option>  ; except <negation-arg>

<declaration>  : (<ident> <expr> <expr2> ... <important>?)
<important> : !important
<expr>      : <term>
            | (/ <term> <term> ...)
            | (:or <term> <term> ...)
            | #(<term> <term> ...)             ; juxtaposition
<term>      : <quantity> | (- <quantity>) | (+ <quantity>)
            | <string> | <ident> | <url> | <hexcolor> | <function>
<quantity>  : <number>
            | (<number> %)
            | (<number> <ident>)
<url>       | (url <string>)
<hexcolor>  | (color <string>)  ; <string> must be hexdigits
<function>  | (<fn> <arg> ...)
<arg>       | <term> | #(<term> ...) | (/ <term> <term> ...)
```

```
<at-rule>    : <at-media-rule> | <at-import-rule>
                   ; NB: Other at-rules are not supported yet
<at-media-rule>  : (@media (<symbol> ...) <style-rule> ...)
<at-import-rule> : (@import <string> (<symbol> ...))
```

NB: Negation op is `:not` instead of `not`, since `(not <negation-arg>)` would be ambiguous from the simple node named `"not"` with one option.

NB: `style-decls` selector rule is currently won't appear in the `parse-css` output; it can be used in SxCSS to make `construct-css` render declarations only, which can be used in the `style` attribute of the document, for example.

Appendix A References

[R5RS] R. Kelsey, W. Clinger, J. Rees (eds.), Revised^5 Report on the Algorithmic Language Scheme, Higher-Order and Symbolic Computation, 11(1), September, 1998 and ACM SIGPLAN Notices, 33(9), October, 1998.
`https://www.schemers.org/Documents/Standards/R5RS/`.

[R7RS] A. Shinn, J. Cowan, A. A. Greckler (eds.), Revised^7 Report on the Algorithmic Language Scheme
`https://bitbucket.org/cowan/r7rs/raw/tip/rnrs/r7rs.pdf`.

[1CC] Carl Bruggeman, Oscar Waddell and R. Kent Dybvig, Representing control in the presence of one-shot continuations, in *Proceedings of SIGPLAN '96*, pp. 99–107, 1996.

[Myers86] Eugene Myers, An O(ND) Difference Algorithm and Its Variations, *Algorithmica* Vol. 1 No. 2, pp. 251-266, 1986.

[MOP] Gregor Kiczales, Jim Des Rivieres, Daniel Bobrow, The Art of Metaobject Protocol, The MIT Press.

[Dylan] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, P. Tucker Withington, A Monotonic Superclass Linearization for Dylan, in *Proceedings of OOPSLA 96*, October 1996.
`https://dl.acm.org/citation.cfm?id=236343`.

[FairThreads]
F. Boussinot, Java Fair Threads,
`https://www-sop.inria.fr/mimosa/rp/FairThreads/html/FairThreads.html`.

[RFC1321]
R. Rivest, The MD5 Message-Digest Algorithm.
April 1992.
`https://www.ietf.org/rfc/rfc1321.txt`.

[RFC2045]
N. Freed, N. Borenstein: Multipurpose Internet Mail Extension (MIME) Part One: Format of Internet Message Bodies, November 1996.
`https://www.ietf.org/rfc/rfc2045.txt`.

[RFC2396]
T. Berners-Lee, R. Fielding, L. Masinter, Uniform Resource Identifiers (URI): Generic Syntax, August 1998.
`https://www.ietf.org/rfc/rfc2396.txt`.

[RFC2616]
R. Fielding et al, Hypertext Transfer Protocol – HTTP/1.1, June 1999.
`https://www.ietf.org/rfc/rfc2616.txt`.

[RFC2822]
P. Resnick (ed.), Internet Message Format, April 2001.
`https://www.ietf.org/rfc/rfc2822.txt`.

[RFC2965]
D. Kristol, L. Montulli, HTTP State Management Mechanism, October 2000.
`https://www.ietf.org/rfc/rfc2965.txt`.

[RFC3174]
D. Eastlake, 3rd and P. Jones, US Secure Hash Algorithm 1 (SHA1).
September 2001.
`https://www.ietf.org/rfc/rfc3174.txt`.

[RFC4648]
S. Josefsson, Ed.: The Base16, Base32, and Base64 Data Encodings October 2006. `https://www.ietf.org/rfc/rfc4648.txt`.

[SRFI-0] Marc Feeley, Feature-based conditional expansion construct, May 1999. `https://srfi.schemers.org/srfi-0/srfi-0.html`.

[SRFI-1] Olin Shivers, List Library, October 1999. `https://srfi.schemers.org/srfi-1/srfi-1.html`.

[SRFI-2] Oleg Kiselyov, `AND-LET*`: an `AND` with local bindings, a guarded `LET*` special form, March 1998. `https://srfi.schemers.org/srfi-2/srfi-2.html`.

[SRFI-6] William D Clinger, Basic String Ports, July 1999. `https://srfi.schemers.org/srfi-6/srfi-6.html`.

[SRFI-8] John David Stone, receive: Binding to multiple values, August 1999. `https://srfi.schemers.org/srfi-8/srfi-8.html`.

[SRFI-9] Richard Kelsey, Defining Record Types, September 1999. `https://srfi.schemers.org/srfi-9/srfi-9.html`.

[SRFI-10] Oleg Kiselyov, `#,` external form, January 2000. `https://srfi.schemers.org/srfi-10/srfi-10.html`.

[SRFI-11] Lars T Hansen, Syntax for receiving multiple values, March 2000. `https://srfi.schemers.org/srfi-11/srfi-11.html`.

[SRFI-13] Olin Shivers, String Libraries, December 2000. `https://srfi.schemers.org/srfi-13/srfi-13.html`.

[SRFI-14] Olin Shivers, Character-set Library, December 2000. `https://srfi.schemers.org/srfi-14/srfi-14.html`.

[SRFI-17] Per Bothner, Generalized `set!`, July 2000. `https://srfi.schemers.org/srfi-17/srfi-17.html`.

[SRFI-18] Marc Feeley, Multithreading Support, April 2000. `https://srfi.schemers.org/srfi-18/srfi-18.html`.

[SRFI-19] Will Fitzgerald, Time Data Types and Procedures, August 2000. `https://srfi.schemers.org/srfi-19/srfi-19.html`.

[SRFI-21] Marc Feeley, Readl-time Multithreading Support, April 2000. `https://srfi.schemers.org/srfi-21/srfi-21.html`.

[SRFI-22] Martin Gasbichler and Michael Sperber, Running Scheme Scripts on Unix, January 2002. `https://srfi.schemers.org/srfi-22/srfi-22.html`.

[SRFI-23] Stephan Housen, Error reporting mechanism, April 2001. `https://srfi.schemers.org/srfi-23/srfi-23.html`.

[SRFI-25] Jussi Piitulainen, Multi-dimensional Array Primitives, June 2002. `https://srfi.schemers.org/srfi-25/srfi-25.html`.

[SRFI-26] Sebastian Egner, Notation for Specializing Parameters without Currying, June 2002. `https://srfi.schemers.org/srfi-26/srfi-26.html`.

[SRFI-27] Sebastian Egner, Sources of Random Bits, June 2002. `https://srfi.schemers.org/srfi-27/srfi-27.html`.

[SRFI-28] Scott G. Miller, Basic Format Strings, June 2002.
`https://srfi.schemers.org/srfi-28/srfi-28.html`.

[SRFI-37] Anthony Carrico, Args-fold: a program argument processor, Jan. 2003.
`https://srfi.schemers.org/srfi-37/srfi-37.html`.

[SSAX] Oleg Kiselyov, XML and Scheme,
`http://okmij.org/ftp/Scheme/xml.html`.
The SSAX distribution is also available at sourceforge:
`http://ssax.sourceforge.net/`.

[MT] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 1998.
`https://dl.acm.org/citation.cfm?id=272995`

[OLEG1] Oleg Kiselyov, Making sense of an input stream,
`http://okmij.org/ftp/Scheme/parsing.html`.

[OLEG2] Oleg Kiselyov, General ways to traverse collections,,
`http://okmij.org/ftp/Scheme/enumerators-callcc.html`. 2000.

Appendix B C to Scheme mapping

For the convenience of the programmers familiar to C, I composed a simple table of C operators and library functions with the corresponding Scheme functions.

`+` R7RS arithmetic procedure `+`. See Section 6.3.4 [Arithmetics], page 112.

`+=` Gauche `inc!` macro. See Section 4.4 [Assignments], page 47.

`-` R7RS arithmetic procedure `-`. See Section 6.3.4 [Arithmetics], page 112.

`-=` Gauche `dec!` macro. See Section 4.4 [Assignments], page 47.

`->` Gauche `slot-ref` is something close to this. See Section 7.3.2 [Accessing instance], page 296.

`* (binary)`
R7RS arithmetic procedure `*`. See Section 6.3.4 [Arithmetics], page 112.

`* (unary)` No equivalent procedure. Scheme doesn't have explicit notation of pointers.

`*=` No equivalent procedure.

`/` In C, it has two different meanings depending on the types of operands. For real division, use `/`. For integer quotient, use `quotient`. See Section 6.3.4 [Arithmetics], page 112.

`/=` No equivalent procedure.

`& (binary)`
Gauche `logand`. See Section 11.34 [Bitwise operations], page 633.

`& (unary)` No equivalent procedure. Scheme doesn't have explicit notation of pointers.

`&&` R7RS syntax `and`. See Section 4.5 [Conditionals], page 49.

`&=` No equivalent procedure.

`|` Gauche `logior`. See Section 11.34 [Bitwise operations], page 633.

`||` R7RS syntax `or`. See Section 4.5 [Conditionals], page 49.

`|=` No equivalent procedure.

`^` Gauche `logxor`. See Section 11.34 [Bitwise operations], page 633.

`=` R7RS syntax `set!`. See Section 4.4 [Assignments], page 47.

`==` R7RS equivalence procedure, `eq?`, `eqv?` and `equal?`. See Section 6.2.1 [Equality], page 97.

`<`
`<=` R7RS arithmetic procedure `<` and `<=`. See Section 6.3.3 [Numerical comparison], page 111. Unlike C operator, Scheme version is transitive.

`<<` Gauche `ash`. See Section 11.34 [Bitwise operations], page 633.

`<<=` No equivalent procedure.

`>`
`>=` R7RS arithmetic procedure `>` and `>=`. See Section 6.3.3 [Numerical comparison], page 111. Unlike C operator, Scheme version is transitive.

`>>` Gauche `ash`. See Section 11.34 [Bitwise operations], page 633.

`>>=` No equivalent procedure.

`%` R7RS operator `modulo` and `remainder`. See Section 6.3.4 [Arithmetics], page 112.

`%=` No equivalent procedure.

`[]` R7RS `vector-ref` (see Section 6.14 [Vectors], page 174) is something close. Or you can use Gauche's generic function `ref` (see Section 9.29 [Sequence framework], page 441) for arbitrary sequences.

`.` Gauche `slot-ref` is something close to this. See Section 7.3.2 [Accessing instance], page 296.

`~` Gauche `lognot`. See Section 11.34 [Bitwise operations], page 633.

`~=` No equivalent procedure.

`!` R7RS procedure `not`. See Section 6.4 [Booleans], page 124.

`!=` No equivalent procedure.

`abort` Gauche `sys-abort`. See Section 6.25.1 [Program termination], page 246.

`abs` R7RS `abs`. See Section 6.3.4 [Arithmetics], page 112.

`access` Gauche `sys-access`. See Section 6.25.4.4 [File stats], page 255.

`acos` R7RS `acos`. See Section 6.3.4 [Arithmetics], page 112.

`alarm` Gauche `sys-alarm`. See Section 6.25.13 [Miscellaneous system calls], page 275.

`asctime` Gauche `sys-asctime`. See Section 6.25.9 [Time], page 269.

`asin` R7RS `asin`. See Section 6.3.4 [Arithmetics], page 112.

`assert` No equivalent function in Gauche.

`atan`
`atan2` R7RS `atan`. See Section 6.3.4 [Arithmetics], page 112.

`atexit` No equivalent function in Gauche, but the "after" thunk of active dynamic handlers are called when `exit` is called. See Section 6.25.1 [Program termination], page 246, and See Section 6.18.7 [Continuations], page 195.

`atof`
`atoi`
`atol` You can use `string->number`. See Section 6.3.5 [Numerical conversions], page 118.

`bsearch` You can use SRFI-133's `vector-binary-search`. See Section 11.30 [Vector library], page 632.

`calloc` Allocation is handled automatically in Scheme.

`ceil` R7RS `ceiling`. See Section 6.3.4 [Arithmetics], page 112.

`cfgetispeed`
`cfgetospeed`
`cfsetispeed`
`cfsetospeed`
Gauche's `sys-cfgetispeed`, `sys-cfgetospeed`, `sys-cfsetispeed`, `sys-cfsetospeed`. See Section 9.31 [Terminal control], page 449.

`chdir` Gauche's `sys-chdir`. See Section 6.25.4.5 [Other file operations], page 257.

`chmod` Gauche's `sys-chmod`. See Section 6.25.4.4 [File stats], page 255.

`chown` Gauche's `sys-chown`. See Section 6.25.4.4 [File stats], page 255.

`clearerr` Not supported yet.

`clock` No equivalent function in Gauche. You can use `sys-times` to get information about CPU time.

`close` You can't directly close the file descriptor, but when you use `close-input-port` or `close-output-port`, underlying file is closed. Some port-related functions, such as `call-with-output-file`, automatically closes the file when operation is finished. The file is also closed when its governing port is garbage collected. See Section 6.22.3 [Common port operations], page 218.

`closedir` No equivalent function in Gauche. You can use `sys-readdir` to read the directory entries at once. See Section 6.25.4.1 [Directories], page 250.

`cos`
`cosh` `cos` and `cosh`. See Section 6.3.4 [Arithmetics], page 112.

`creat` A file is implicitly created by default when you open it for writing. See Section 6.22.4 [File ports], page 221, for more control over the creation of files.

`ctermid` Gauche `sys-ctermid`. See Section 6.25.8 [System inquiry], page 266.

`ctime` Gauche `sys-ctime`. See Section 6.25.9 [Time], page 269.

`cuserid` No equivalent function. This is removed from the newer POSIX. You can use alternative functions, such as `sys-getlogin` or `sys-getpwuid` with `sys-getuid`.

`difftime` Gauche `sys-difftime`. See Section 6.25.9 [Time], page 269.

`div` You can use R7RS `quotient` and `remainder`. See Section 6.3.4 [Arithmetics], page 112.

`dup`
`dup2` Not directly supported, but you can use `port-fd-dup!`.

`execl`
`execle`
`execlp`
`execv`
`execve`
`execvp` Gauche `sys-exec`. See Section 6.25.10 [Process management], page 271. For higher level interface, Section 9.25 [High-level process interface], page 421.

`exit`

`_exit` Use `exit` or `sys-exit`, depends on what you need. See Section 6.25.1 [Program termination], page 246.

`exp` R7RS `exp`. See Section 6.3.4 [Arithmetics], page 112.

`fabs` R7RS `abs`. See Section 6.3.4 [Arithmetics], page 112.

`fclose` You can't directly close the file stream, but when you use `close-input-port` or `close-output-port`, underlying file is closed. Some port-related functions, such as `call-with-output-file`, automatically closes the file when operation is finished. The file is also closed when its governing port is garbage collected.

`fcntl` Implemented as `sys-fcntl` in `gauche.fcntl` module. See Section 9.10 [Low-level file operations], page 370.

`fdopen` Gauche's `open-input-fd-port` or `open-output-fd-port`. See Section 6.22.4 [File ports], page 221.

feof No equivalent operation, but you can check if an input port have reached to the end by `peek-char` or `peek-byte`. See Section 6.22.7.1 [Reading data], page 227.

ferror Not supported yet.

fflush Gauche's `flush`. See Section 6.22.8 [Output], page 231.

fgetc Use `read-char` or `read-byte`. See Section 6.22.7 [Input], page 227.

fgetpos Use Gauche's `port-tell` (see Section 6.22.3 [Common port operations], page 218)

fgets Use `read-line` or `read-string`. See Section 6.22.7 [Input], page 227.

fileno `port-file-number`. See Section 6.22.3 [Common port operations], page 218.

floor R7RS `floor`. See Section 6.3.4 [Arithmetics], page 112.

fmod Gauche's `fmod`.

fopen R7RS `open-input-file` or `open-output-file` corresponds to this operation. See Section 6.22.4 [File ports], page 221.

fork Gauche's `sys-fork`. See Section 6.25.10 [Process management], page 271.

forkpty Use `sys-forkpty`. See Section 9.31 [Terminal control], page 449.

fpathconf
Not supported.

fprintf Not directly supported, but Gauche's `format` provides similar functionality. See Section 6.22.8 [Output], page 231. SLIB has `printf` implementation.

fputc Use `write-char` or `write-byte`. See Section 6.22.8 [Output], page 231.

fputs Use `display`. See Section 6.22.8 [Output], page 231.

fread Not directly supported. To read binary numbers, see Section 12.1 [Binary I/O], page 638. If you want to read a chunk of bytes, you may be able to use `read-uvector!`. See Section 9.36.4 [Uvector block I/O], page 488.

free You don't need this in Scheme.

freopen Not supported.

frexp Gauche's `frexp`

fscanf Not supported. For general case, you have to write a parser. If you can keep the data in S-exp, you can use `read`. If the syntax is very simple, you may be able to utilize `string-tokenize` in `srfi-14` (Section 11.5 [String library], page 591), and/or regular expression stuff (Section 6.13 [Regular expressions], page 162).

fseek Use Gauche's `port-seek` (see Section 6.22.3 [Common port operations], page 218)

fsetpos Use Gauche's `port-seek` (see Section 6.22.3 [Common port operations], page 218)

fstat Gauche's `sys-stat`. See Section 6.25.4.4 [File stats], page 255.

ftell Use Gauche's `port-tell` (see Section 6.22.3 [Common port operations], page 218)

fwrite Not directly supported. To write binary numbers, see Section 12.1 [Binary I/O], page 638. If you want to write a chunk of bytes, you can simply use `display` or `write-uvector` (see Section 9.36.4 [Uvector block I/O], page 488).

getc
getchar Use `read-char` or `read-byte`. See Section 6.22.7 [Input], page 227.

getcwd Gauche's `sys-getcwd`. See Section 6.25.8 [System inquiry], page 266.

`getdomainname`
: Gauche's `sys-getdomainname`. See Section 6.25.8 [System inquiry], page 266.

`getegid`
: Gauche's `sys-getegid`. See Section 6.25.8 [System inquiry], page 266.

`getenv`
: Gauche's `sys-getenv`. See Section 6.25.3 [Environment inquiry], page 248.

`geteuid`
: Gauche's `sys-geteuid`. See Section 6.25.8 [System inquiry], page 266.

`gethostname`
: Gauche's `sys-gethostname`. See Section 6.25.8 [System inquiry], page 266.

`getgid`
: Gauche's `sys-getgid`. See Section 6.25.8 [System inquiry], page 266.

`getgrgid`
`getgrnam`
: Gauche's `sys-getgrgid` and `sys-getgrnam`. See Section 6.25.5 [Unix groups and users], page 257.

`getgroups`
: Gauche's `sys-getgroups`. See Section 6.25.8 [System inquiry], page 266.

`getlogin`
: Gauche's `sys-getlogin`. See Section 6.25.8 [System inquiry], page 266.

`getpgrp`
: Gauche's `sys-getpgrp`. See Section 6.25.8 [System inquiry], page 266.

`getpid`
`getppid`
: Gauche's `sys-getpid`. See Section 6.25.8 [System inquiry], page 266.

`getpwnam`
`getpwuid`
: Gauche's `sys-getpwnam` and `sys-getpwuid`. See Section 6.25.5 [Unix groups and users], page 257.

`gets`
: Use `read-line` or `read-string`. See Section 6.22.7 [Input], page 227.

`gettimeofday`
: Gauche's `sys-gettimeofday`. See Section 6.25.9 [Time], page 269.

`getuid`
: Gauche's `sys-getuid`. See Section 6.25.8 [System inquiry], page 266.

`gmtime`
: Gauche's `sys-gmtime`. See Section 6.25.9 [Time], page 269.

`isalnum`
: Not directly supported, but you can use R7RS `char-alphabetic?` and `char-numeric?`. See Section 6.10 [Characters], page 143. You can also use character set. See Section 6.11 [Character set], page 147, also Section 11.6 [Character-set library], page 600.

`isalpha`
: R7RS `char-alphabetic?`. See Section 6.10 [Characters], page 143. See also Section 6.11 [Character set], page 147, and Section 11.6 [Character-set library], page 600.

`isatty`
: Gauche's `sys-isatty`. See Section 6.25.4.5 [Other file operations], page 257.

`iscntrl`
: Not directly supported, but you can use `(char-set-contains? char-set:iso-control c)` with `srfi-14`. See Section 11.6 [Character-set library], page 600.

`isdigit`
: R7RS `char-numeric?`. See Section 6.10 [Characters], page 143. You can also use `(char-set-contains? char-set:digit c)` with `srfi-14`. See Section 11.6 [Character-set library], page 600.

`isgraph`
: Not directly supported, but you can use `(char-set-contains? char-set:graphic c)` with `srfi-14`. See Section 11.6 [Character-set library], page 600.

`islower`
: R7RS `char-lower-case?`. See Section 6.10 [Characters], page 143. You can also use `(char-set-contains? char-set:lower-case c)` with `srfi-14`. See Section 11.6 [Character-set library], page 600.

`isprint` Not directly supported, but you can use `(char-set-contains? char-set:printing c)` with `srfi-14`. See Section 11.6 [Character-set library], page 600.

`ispunct` Not directly supported, but you can use `(char-set-contains? char-set:punctuation c)` with `srfi-14`. See Section 11.6 [Character-set library], page 600.

`isspace` R7RS `char-whitespace?`. See Section 6.10 [Characters], page 143. You can also use `(char-set-contains? char-set:whitespace c)` with `srfi-14`. See Section 11.6 [Character-set library], page 600.

`isupper` R7RS `char-upper-case?`. See Section 6.10 [Characters], page 143. You can also use `(char-set-contains? char-set:upper-case c)` with `srfi-14`. See Section 11.6 [Character-set library], page 600.

`isxdigit` Not directly supported, but you can use `(char-set-contains? char-set:hex-digit c)` with `srfi-14`. See Section 11.6 [Character-set library], page 600.

`kill` Gauche's `sys-kill`. See Section 6.25.7 [Signal], page 260.

`labs` R7RS `abs`. See Section 6.3.4 [Arithmetics], page 112.

`ldexp` Gauche's `ldexp`.

`ldiv` Use R7RS `quotient` and `remainder`. See Section 6.3.4 [Arithmetics], page 112.

`link` Gauche's `sys-link`. See Section 6.25.4.2 [Directory manipulation], page 252.

`localeconv`
Gauche's `sys-localeconv`. See Section 6.25.6 [Locale], page 259.

`localtime`
Gauche's `sys-localtime`. See Section 6.25.9 [Time], page 269.

`log` R7RS `log`. See Section 6.3.4 [Arithmetics], page 112.

`log10` Not directly supported. `log10(z)` ≡ `(/ (log z) (log 10))`.

`longjmp` R7RS `call/cc` provides similar (superior) mechanism. See Section 6.18.7 [Continuations], page 195.

`lseek` Use Gauche's `port-seek` (see Section 6.22.3 [Common port operations], page 218)

`malloc` Not necessary in Scheme.

`mblen`
`mbstowcs`
`mbtowc` Gauche handles multibyte strings internally, so generally you don't need to care about multibyte-ness of the string. `string-length` always returns a number of characters for a string in supported encoding. If you want to convert the character encoding, see Section 9.4 [Character code conversion], page 339.

`memcmp`
`memcpy`
`memmove`
`memset` No equivalent functions.

`mkdir` Gauche's `sys-mkdir`. See Section 6.25.4.2 [Directory manipulation], page 252.

`mkfifo` Gauche's `sys-mkfifo`.

`mkstemp` Gauche's `sys-mkstemp`. See Section 6.25.4.2 [Directory manipulation], page 252. Use this instead of tmpnam.

`mktime` Gauche's `sys-mktime`. See Section 6.25.9 [Time], page 269.

`modf` Gauche's `modf`.

`open` Not directly supported. R7RS `open-input-file` or `open-output-file` corresponds to this operation. See Section 6.22.4 [File ports], page 221.

`opendir` Not directly supported. You can use `sys-readdir` to read the directory entries at once. See Section 6.25.4.1 [Directories], page 250.

`openpty` Use `sys-openpty`. See Section 9.31 [Terminal control], page 449.

`pathconf` Not supported.

`pause` Gauche's `sys-pause`. See Section 6.25.13 [Miscellaneous system calls], page 275.

`perror` No equivalent function in Gauche. System calls generally throws an error (`<system-error>`), including the description of the reason of failure.

`pipe` Gauche's `sys-pipe`. See Section 6.25.4.5 [Other file operations], page 257.

`pow` R7RS `expt`. See Section 6.3.4 [Arithmetics], page 112.

`printf` Not directly supported, but Gauche's `format` provides similar functionality. See Section 6.22.8 [Output], page 231. SLIB has `printf` implementation.

`putc`
`putchar` Use `write-char` or `write-byte`. See Section 6.22.8 [Output], page 231.

`puts` Use `display`. See Section 6.22.8 [Output], page 231.

`qsort` Gauche's `sort` and `sort!` provides a convenient way to sort list of items. See Section 6.24 [Sorting and merging], page 244.

`raise` No equivalent function in Gauche. Scheme function `raise` (SRFI-18) is to raise an exception. You can use `(sys-kill (sys-getpid) SIG)` to send a signal `SIG` to the current process.

`rand` Not supported directly, but on most platforms a better RNG is available as `sys-random`. See Section 6.25.13 [Miscellaneous system calls], page 275.

`read` Not supported directly, but you may be able to use `read-uvector` or `read-uvector!` (see Section 9.36.4 [Uvector block I/O], page 488).

`readdir` Not supported directly. Gauche's `sys-readdir` reads the directory at once. See Section 6.25.4.1 [Directories], page 250.

`readlink` Gauche's `sys-readlink`. See Section 6.25.4.2 [Directory manipulation], page 252. This function is available on systems that support symbolic links.

`realloc` Not necessary in Scheme.

`realpath` Gauche's `sys-normalize-pathname` or `sys-realpath`. See Section 6.25.4.3 [Pathnames], page 253.

`remove` Gauche's `sys-remove`. See Section 6.25.4.2 [Directory manipulation], page 252.

`rename` Gauche's `sys-rename`. See Section 6.25.4.2 [Directory manipulation], page 252.

`rewind` Not directly supported, but you can use `port-seek` instead. See Section 6.22.3 [Common port operations], page 218.

`rewinddir`
Not supported directly. You can use `sys-readdir` to read the directory entries at once. See Section 6.25.4.1 [Directories], page 250.

rmdir Gauche's sys-rmdir. See Section 6.25.4.2 [Directory manipulation], page 252.

scanf Not supported. For general case, you have to write a parser. If you can keep the data in S-exp, you can use read. If the syntax is very simple, you may be able to utilize string-tokenize in srfi-14 (Section 11.5 [String library], page 591), and/or regular expression stuff (Section 6.13 [Regular expressions], page 162).

select Gauche's sys-select. See Section 6.25.11 [I/O multiplexing], page 274.

setbuf Not necessary.

setgid Gauche's sys-setgid.

setjmp R7RS call/cc provides similar (superior) mechanism. See Section 6.18.7 [Continuations], page 195.

setlocale
Gauche's sys-setlocale. See Section 6.25.6 [Locale], page 259.

setpgid Gauche's sys-setpgid. See Section 6.25.8 [System inquiry], page 266.

setsid Gauche's sys-setsid. See Section 6.25.8 [System inquiry], page 266.

setuid Gauche's sys-setuid. See Section 6.25.8 [System inquiry], page 266.

setvbuf Not necessary.

sigaction
You can use set-signal-handler! to install signal handlers. See Section 6.25.7.3 [Handling signals], page 262.

sigaddset
sigdelset
sigemptyset
sigfillset
Gauche's sys-sigset-add! and sys-sigset-delete!. See Section 6.25.7.1 [Signals and signal sets], page 260.

sigismember
Not supported yet.

siglongjmp
R7RS call/cc provides similar (superior) mechanism. See Section 6.18.7 [Continuations], page 195.

signal You can use with-signal-handlers to install signal handlers. See Section 6.25.7.3 [Handling signals], page 262.

sigpending
Not supported yet.

sigprocmask
Signal mask is handled internally. See Section 6.25.7.3 [Handling signals], page 262.

sigsetjmp
R7RS call/cc provides similar (superior) mechanism. See Section 6.18.7 [Continuations], page 195.

sigsuspend
Gauche's sys-sigsuspend. See Section 6.25.7.4 [Masking and waiting signals], page 264.

sigwait Gauche's sys-sigwait. See Section 6.25.7.4 [Masking and waiting signals], page 264.

`sin`
`sinh` Use `sin` and `sinh`. See Section 6.3.4 [Arithmetics], page 112.

`sleep` Gauche's `sys-sleep`. See Section 6.25.13 [Miscellaneous system calls], page 275.

`sprintf` Not directly supported, but Gauche's `format` provides similar functionality. See Section 6.22.8 [Output], page 231. SLIB has `printf` implementation.

`sqrt` R7RS `sqrt`. See Section 6.3.4 [Arithmetics], page 112.

`srand` Not supported directly, but on most platforms a better RNG is available as `sys-srandom` (see Section 6.25.13 [Miscellaneous system calls], page 275). The `math.mt-random` module provides much superior RNG (see Section 12.26 [Mersenne-Twister random number generator], page 704).

`sscanf` Not supported. For general case, you have to write a parser. If you can keep the data in S-exp, you can use `read`. If the syntax is very simple, you may be able to utilize `string-tokenize` in `srfi-14` (Section 11.5 [String library], page 591), and/or regular expression stuff (Section 6.13 [Regular expressions], page 162).

`stat` Gauche's `sys-stat`. See Section 6.25.4.4 [File stats], page 255.

`strcasecmp`
R7RS `string-ci=?` and other comparison functions. See Section 6.12.6 [String Comparison], page 158.

`strcat` R7RS `string-append`. See Section 6.12.7 [String utilities], page 158.

`strchr` SRFI-13 `string-index`. See Section 11.5.7 [SRFI-13 String searching], page 596.

`strcmp` R7RS `string=?` and other comparison functions. See Section 6.12.6 [String Comparison], page 158.

`strcoll` Not supported yet.

`strcpy` R7RS `string-copy`. See Section 6.12.7 [String utilities], page 158.

`strcspn` Not directly supported, but you can use SRFI-13 `string-skip` with a character set. See Section 11.5.7 [SRFI-13 String searching], page 596.

`strerror` Gauche's `sys-strerror`. See Section 6.25.8 [System inquiry], page 266.

`strftime` Gauche's `sys-strftime`. See Section 6.25.9 [Time], page 269.

`strlen` R7RS `string-length`. See Section 6.12.5 [String Accessors & Modifiers], page 157.

`strncat` Not directly supported, but you can use `string-append` and `substring`.

`strncasecmp`
SRFI-13 `string-compare-ci` provides the most flexible (but a bit difficult to use) functionality. See Section 11.5.5 [SRFI-13 String Comparison], page 594. If what you want is just to check the fixed-length prefixes of two string matches, you can use SRFI-13 `string-prefix-ci?`.

`strncmp` SRFI-13 `string-compare` provides the most flexible (but a bit difficult to use) functionality. See Section 11.5.5 [SRFI-13 String Comparison], page 594. If what you want is just to check the fixed-length prefixes of two string matches, you can use SRFI-13 `string-prefix?`. See Section 11.5.6 [SRFI-13 String Prefixes & Suffixes], page 595.

`strncpy` SRFI-13 `substring`. See Section 6.12.7 [String utilities], page 158.

`strpbrk` Not directly supported, but you can use SRFI-13 `string-skip` with a character set. See Section 11.5.7 [SRFI-13 String searching], page 596.

`strrchr` SRFI-13 `string-index-right`. See Section 11.5.7 [SRFI-13 String searching], page 596.

`strspn` Not directly supported, but you can use SRFI-13 `string-index` with a character set. See Section 11.5.7 [SRFI-13 String searching], page 596.

`strstr` SRFI-13 `string-contains`. See Section 11.5.7 [SRFI-13 String searching], page 596.

`strtod` You can use R7RS `string->number`. See Section 6.3.5 [Numerical conversions], page 118.

`strtok` SRFI-13 `string-tokenize`. See Section 11.5.12 [SRFI-13 other string operations], page 599.

`strtol`
`strtoul` You can use R7RS `string->number`. See Section 6.3.5 [Numerical conversions], page 118.

`strxfrm` Not supported yet.

`symlink` Gauche's `sys-symlink`. See Section 6.25.4.2 [Directory manipulation], page 252. This function is available on systems that support symbolic links.

`sysconf` Not supported yet.

`system` Gauche's `sys-system`. See Section 6.25.10 [Process management], page 271. It is generally recommended to use the process library (Section 9.25 [High-level process interface], page 421).

`tan`
`tanh` R7RS `tan` and Gauche `tanh`. See Section 6.3.4 [Arithmetics], page 112.

`tcdrain`

`tcflow`

`tcflush`

`tcgetattr`
`tcgetpgrp`
`tcsendbreak`
`tcsetattr`
`tcsetpgrp`
Corresponding functions are: `sys-tcdrain`, `sys-tcflow`, `sys-tcflush`, `sys-tcgetattr`, `sys-tcgetpgrp`, `sys-tcsendbreak`, `sys-tcsetattr`, `sys-tcsetpgrp`. See Section 9.31 [Terminal control], page 449.

`time` Gauche's `sys-time`. See Section 6.25.9 [Time], page 269.

`times` Gauche's `sys-times`. See Section 6.25.8 [System inquiry], page 266.

`tmpfile` Not exactly supported. See `sys-mkstemp` instead. See Section 6.25.4.2 [Directory manipulation], page 252.

`tmpnam` Gauche's `sys-tmpnam`. This function is provided since it is in POSIX, but its use is discouraged for the potential security risk. Use `sys-mkstemp` instead. See Section 6.25.4.2 [Directory manipulation], page 252.

`tolower`
`toupper` R7RS `char-upcase` and `char-downcase`. See Section 6.10 [Characters], page 143.

`ttyname` Gauche's `sys-ttyname`. See Section 6.25.4.5 [Other file operations], page 257.

`tzset` Not supported yet.

`umask` Gauche's `sys-umask`. See Section 6.25.4.2 [Directory manipulation], page 252.

`uname` Gauche's `sys-uname`. See Section 6.25.8 [System inquiry], page 266.

`ungetc` Not directly supported. You can use `peek-char` to look one character ahead, instead of pushing back.

`unlink` Gauche's `sys-unlink`. See Section 6.25.4.2 [Directory manipulation], page 252.

`utime` Gauche's `sys-utime`. See Section 6.25.4.4 [File stats], page 255.

`va_arg`
`va_end`
`va_start` Not necessary, for Scheme handles variable number of arguments naturally.

`vfprintf`
`vprintf`
`vsprintf` Not directly supported, but Gauche's `format` provides similar functionality. See Section 6.22.8 [Output], page 231. SLIB has `printf` implementation.

`wait` Gauche's `sys-wait`. See Section 6.25.10 [Process management], page 271.

`waitpid` Gauche's `sys-waitpid`. See Section 6.25.10 [Process management], page 271.

`wcstombs`
`wctomb` Gauche handles multibyte strings internally, so generally you don't need to care about multibyte-ness of the string. `string-length` always returns a number of characters for a string in supported encoding. If you want to convert the character encoding, see Section 9.4 [Character code conversion], page 339.

`write` R7RS `display` (see Section 6.22.8 [Output], page 231). Or `write-uvector` (see Section 9.36.4 [Uvector block I/O], page 488).

Appendix C Function and Syntax Index

!

!= 335

$

$ 45

%

% 334
%= 335
%macroexpand 95
%macroexpand-1 95

&

& 334

(

(setter ~) 189
(setter cgi-test-environment-ref) 824
(setter dict-get) 367
(setter object-apply) 195
(setter port-buffering) 220
(setter random-data-seed) 660
(setter ref) 179, 190, 296, 441
(setter subseq) 442

*

* 112, 334
*. 113
*= 335

+

+ 112, 334
+. 113
+= 335

–

- 112, 334
-. 113
-= 335
-> 334
->char-set 534

.

.$ 191
.array 332
.cond 333
.define 333, 339
.function 332
.if 333, 339
.include 333, 339
.raw-c-code 334
.static-decls 334
.struct 332
.type 335
.undef 333, 339
.union 332
.unless 333, 339
.when 333, 339

/

/ 112, 334
/. 113
/= 335

:

: 612
:char-range 613
:dispatched 614
:do 614
:generator 614
:integers 613
:let 614
:list 613
:parallel 614
:port 613
:range 613
:real-range 613
:string 613
:until 614
:uvector 613
:vector 613
:while 614

<

< 111, 335
<< 334
<<= 335
<= 111, 335
<=? 105
<? 105
<gauche-package-description> 411

=

= 111, 335
== 335
=? 105

>

> 111, 335
>= 111, 335
>=? 105
>> 334
>>= 335
>? 105

?

?: 335

@

@? 477
@vector 478
@vector->list 484
@vector->vector 484
@vector-add 487
@vector-add! 487
@vector-and 487
@vector-and! 487
@vector-append 482
@vector-append-subvectors 482
@vector-clamp 488
@vector-clamp! 488
@vector-compare 480
@vector-concatenate 482
@vector-copy 480
@vector-copy! 480
@vector-div 487
@vector-div! 487
@vector-dot 487
@vector-empty? 478
@vector-fill! 480
@vector-ior 487
@vector-ior! 487
@vector-length 478
@vector-mul 487
@vector-mul! 487
@vector-multi-copy! 481
@vector-range-check 487
@vector-ref 479
@vector-reverse-copy 480
@vector-set! 479
@vector-sub 487
@vector-sub! 487
@vector-swap! 480
@vector-unfold 478
@vector-unfold-right 478
@vector-xor 487
@vector-xor! 487
@vector= 480
@vector=? 480
@vector? 477

^

^ 42
^_ 44
^a 44
^b 44
^c 44
^d 44
^e 44
^f 44
^g 44
^h 44
^i 44
^j 44
^k 44
^l 44
^m 44
^n 44
^o 44
^p 44
^q 44
^r 44
^s 44
^t 44
^u 44
^v 44
^w 44
^x 44
^y 44
^z 44

~

~ 189

A

abandoned-mutex-exception? 468
abs 113
absolute-path? 696
acons 135
acos 117
acosh 117
add-duration 602
add-duration! 602
add-hook! 384
add-job! 647
add-load-path 240
address-family 624
address-info 624
adler32 746
alist->bag 530
alist->hash-table 179, 537, 619
alist->hashmap 573
alist->hashmap! 573
alist->imap 655
alist->mapping 568
alist->mapping! 568
alist->mapping/ordered 568
alist->mapping/ordered! 568
alist->rbtree 313
alist->tree-map 185
alist-cons 516
alist-copy 135
alist-delete 136

alist-delete! 136
all-modules 77
allocate-instance 295
and 51, 334, 614
and-let* 53
and-let1 53
angle 118
any 133
any$ 191
any-bit-set? 576
any-bits-set? 617
any-in-queue 659
any-pred 191
any?-ec 612
append 134
append! 134
append-ec 611
append-map 131
append-map! 131
append-reverse 135
append-reverse! 135
applicable? 188
apply 188
apply$ 190
apply-generic 307
apply-method 307
apply-methods 307
approx=? 112
apropos 384
aref 334
args-fold 608
arithmetic-shift 576
arity 194
arity-at-least-value 194
arity-at-least? 194
array 316
array->list 319
array->vector 319
array-add-elements 321
array-add-elements! 321
array-concatenate 319
array-copy 316
array-div-elements 321
array-div-elements! 321
array-div-left 321
array-div-right 321
array-end 317
array-expt 321
array-flip 320
array-flip! 320
array-for-each-index 318
array-inverse 321
array-length 317
array-map 319
array-map! 319
array-mul 321
array-mul-elements 321
array-mul-elements! 321
array-negate-elements 322
array-negate-elements! 322
array-rank 317
array-reciprocate-elements 322
array-reciprocate-elements! 322
array-ref 317
array-retabulate! 319
array-rotate-90 320
array-set! 317
array-shape 317
array-size 317
array-start 317
array-sub-elements 321
array-sub-elements! 321
array-transpose 320
array? 316
as-nodeset 758
ash 121
asin 117
asinh 117
assert-curr-char 787
assoc 135
assoc$ 191
assoc-adjoin 136
assoc-ref 136
assoc-set! 136
assoc-update-in 137
assq 135
assq-ref 136
assq-set! 136
assume 51
assume-type 51
assv 135
assv-ref 136
assv-set! 136
atan 117
atanh 117
atom 466
atom-ref 466
atom? 466
atomic 466
atomic-update! 466
attlist->alist 749
attlist-add 749
attlist-fold 749
attlist-null? 749
attlist-remove-top 749
autoload 242

B

bag 525
bag->alist 530
bag->list 530
bag->set 530
bag-adjoin 527
bag-adjoin! 527
bag-any? 528
bag-contains? 526
bag-copy 530
bag-count 528
bag-decrement! 532
bag-delete 527
bag-delete! 527
bag-delete-all 527
bag-delete-all! 527
bag-difference 531
bag-difference! 531
bag-disjoint? 526
bag-element-comparator 527
bag-element-count 532
bag-empty? 526

bag-every? 528
bag-filter 529
bag-filter! 529
bag-find 528
bag-fold 529
bag-fold-unique 532
bag-for-each 529
bag-for-each-unique 532
bag-increment! 532
bag-intersection 531
bag-intersection! 531
bag-map 528
bag-member 526
bag-partition 529
bag-partition! 529
bag-product 532
bag-product! 532
bag-remove 529
bag-remove! 529
bag-replace 527
bag-replace! 527
bag-search! 527
bag-size 528
bag-sum 532
bag-sum! 532
bag-unfold 526
bag-union 531
bag-union! 531
bag-unique-size 532
bag-xor 531
bag-xor! 531
bag<=? 531
bag<? 531
bag=? 530
bag>=? 531
bag>? 531
balanced-quotient 575
balanced-remainder 575
balanced/ 575
base64-decode 716
base64-decode-string 716
base64-encode 716
base64-encode-string 716
bcrypt-gensalt 648
bcrypt-hashpw 648
beep 774
begin 55, 332, 339, 614
begin0 56
bignum? 111
bimap-left 369
bimap-left-delete! 370
bimap-left-exists? 370
bimap-left-get 370
bimap-put! 370
bimap-right 369
bimap-right-delete! 370
bimap-right-exists? 370
bimap-right-get 370
binary-heap-clear! 653
binary-heap-copy 653
binary-heap-delete! 654
binary-heap-empty? 653
binary-heap-find 654
binary-heap-find-max 653
binary-heap-find-min 653
binary-heap-num-entries 653
binary-heap-pop-max! 653
binary-heap-pop-min! 653
binary-heap-push! 653
binary-heap-remove! 654
binary-heap-swap-max! 653
binary-heap-swap-min! 653
binary-port? 506
bindtextdomain 784
bit-count 576
bit-field 122
bit-field-any? 577
bit-field-clear 577
bit-field-every? 577
bit-field-replace 577
bit-field-replace-same 577
bit-field-reverse 577
bit-field-rotate 577
bit-field-set 577
bit-set? 576
bit-swap 576
bits 578
bits->generator 374
bits->list 577
bits->vector 577
bitwise-and 575
bitwise-andc1 575
bitwise-andc2 575
bitwise-eqv 575
bitwise-fold 578
bitwise-for-each 578
bitwise-if 576
bitwise-ior 575
bitwise-merge 617
bitwise-nand 575
bitwise-nor 575
bitwise-not 575
bitwise-orc1 575
bitwise-orc2 575
bitwise-unfold 578
bitwise-xor 575
blob->sint-list 622
blob->u8-list 622
blob->uint-list 622
blob-copy 622
blob-copy! 622
blob-length 620
blob-s16-native-ref 621
blob-s16-native-set! 621
blob-s16-ref 621
blob-s16-set! 621
blob-s32-native-ref 621
blob-s32-native-set! 621
blob-s32-ref 621
blob-s32-set! 621
blob-s64-native-ref 621
blob-s64-native-set! 621
blob-s64-ref 621
blob-s64-set! 621
blob-s8-ref 621
blob-s8-set! 621
blob-sint-ref 620
blob-sint-set! 621
blob-u16-native-ref 621
blob-u16-native-set! 621

blob-u16-ref 621
blob-u16-set! 621
blob-u32-native-ref 621
blob-u32-native-set! 621
blob-u32-ref 621
blob-u32-set! 621
blob-u64-native-ref 621
blob-u64-native-set! 621
blob-u64-ref 621
blob-u64-set! 621
blob-u8-ref 621
blob-u8-set! 621
blob-uint-ref 620
blob-uint-set! 621
blob=? 621
blob? 620
boolean 124
boolean-hash 102
boolean=? 124
boolean? 124
booleans 661
booleans->integer 617
box 547
box? 547
bpsw-prime? 707
break 333, 516
break! 516
break-list-by-sequence 444
break-list-by-sequence! 444
build-binary-heap 653
build-path 696
build-transliterator 793
byte-ready? 228
bytevector 490
bytevector->generator 373
bytevector-accumulator 543
bytevector-accumulator! 544
bytevector-append 490
bytevector-copy 490
bytevector-copy! 490
bytevector-length 490
bytevector-u8-ref 490
bytevector-u8-set! 490
bytevector=? 490
bytevector? 490

C

c128? 477
c128vector 478
c128vector->list 484
c128vector->vector 484
c128vector-add 487
c128vector-add! 487
c128vector-append 482
c128vector-append-subvectors 482
c128vector-compare 480
c128vector-concatenate 482
c128vector-copy 480
c128vector-copy! 480
c128vector-div 487
c128vector-div! 487
c128vector-dot 487
c128vector-empty? 478
c128vector-fill! 480
c128vector-length 478
c128vector-mul 487
c128vector-mul! 487
c128vector-multi-copy! 481
c128vector-ref 479
c128vector-reverse-copy 480
c128vector-set! 479
c128vector-sub 487
c128vector-sub! 487
c128vector-swap! 480
c128vector-unfold 478
c128vector-unfold-right 478
c128vector= 480
c128vector=? 480
c128vector? 477
c32? 477
c32vector 478
c32vector->list 484
c32vector->vector 484
c32vector-add 487
c32vector-add! 487
c32vector-append 482
c32vector-append-subvectors 482
c32vector-compare 480
c32vector-concatenate 482
c32vector-copy 480
c32vector-copy! 480
c32vector-div 487
c32vector-div! 487
c32vector-dot 487
c32vector-empty? 478
c32vector-fill! 480
c32vector-length 478
c32vector-mul 487
c32vector-mul! 487
c32vector-multi-copy! 481
c32vector-ref 479
c32vector-reverse-copy 480
c32vector-set! 479
c32vector-sub 487
c32vector-sub! 487
c32vector-swap! 480
c32vector-unfold 478
c32vector-unfold-right 478
c32vector= 480
c32vector=? 480
c32vector? 477
c64? 477
c64vector 478
c64vector->list 484
c64vector->vector 484
c64vector-add 487
c64vector-add! 487
c64vector-append 482
c64vector-append-subvectors 482
c64vector-compare 480
c64vector-concatenate 482
c64vector-copy 480
c64vector-copy! 480
c64vector-div 487
c64vector-div! 487
c64vector-dot 487
c64vector-empty? 478
c64vector-fill! 480

c64vector-length 478
c64vector-mul 487
c64vector-mul! 487
c64vector-multi-copy! 481
c64vector-ref 479
c64vector-reverse-copy 480
c64vector-set! 479
c64vector-sub 487
c64vector-sub! 487
c64vector-swap! 480
c64vector-unfold 478
c64vector-unfold-right 478
c64vector= 480
c64vector=? 480
c64vector? 477
caaaar 127
caaadr 127
caaar 127
caadar 127
caaddr 127
caadr 127
caar 127
cache-check! 651
cache-clear! 651
cache-compact-queue! 652
cache-comparator 651
cache-evict! 651
cache-lookup! 650
cache-populate-queue! 651
cache-register! 651
cache-renumber-entries! 652
cache-storage 651
cache-through! 650
cache-write! 651
cadaar 127
cadadr 127
cadar 127
caddar 127
cadddr 127
caddr 127
cadr 127
calculate-dominators 796
call-with-builder 350
call-with-cgi-script 825
call-with-client-socket 403
call-with-console 774
call-with-current-continuation 195
call-with-ftp-connection 718
call-with-input-conversion 343
call-with-input-file 223
call-with-input-process 430
call-with-input-string 225
call-with-iterator 349
call-with-iterators 350
call-with-output-conversion 343
call-with-output-file 223
call-with-output-process 431
call-with-output-string 225
call-with-port 219
call-with-process-io 431
call-with-string-io 226
call-with-temporary-directory 702
call-with-temporary-file 701
call-with-values 197
call/cc 195
call/pc 418
car 127
car+cdr 514
car-sxpath 763
cartesian-product 795
cartesian-product-right 795
case 50, 332
case-lambda 46
case/fallthrough 332
cast 335
cdaaar 127
cdaadr 127
cdaar 127
cdadar 127
cdaddr 127
cdadr 127
cdar 127
cddaar 127
cddadr 127
cddar 127
cdddar 127
cddddr 127
cdddr 127
cddr 127
cdr 127
ceiling 116
ceiling->exact 117
ceiling-quotient 574
ceiling-remainder 574
ceiling/ 574
ces-conversion-supported? 340
ces-convert 342
ces-convert-to 342
ces-equivalent? 340
ces-guess-from-string 341
ces-upper-compatible? 340
cf$ 358
cf-arg-enable 355
cf-arg-var 358
cf-arg-with 355
cf-check-decl 361
cf-check-decls 361
cf-check-func 362
cf-check-funcs 362
cf-check-header 360
cf-check-headers 360
cf-check-lib 362
cf-check-member 361
cf-check-members 361
cf-check-prog 359
cf-check-type 360
cf-check-types 361
cf-config-headers 364
cf-define 358
cf-echo 357
cf-feature-ref 356
cf-have-subst? 358
cf-help-string 356
cf-includes-default 360
cf-init 355
cf-init-gauche-extension 354
cf-lang 363
cf-lang-call 363
cf-lang-io-program 363
cf-lang-program 363

cf-make-gpd 365
cf-msg-checking 356
cf-msg-error 357
cf-msg-notice 357
cf-msg-result 357
cf-msg-warn 357
cf-output 364
cf-output-default 363
cf-package-ref 356
cf-path-prog 359
cf-prog-cxx 359
cf-ref 358
cf-search-libs 362
cf-show-substs 365
cf-subst 358
cf-subst-append 358
cf-subst-prepend 358
cf-try-compile 363
cf-try-compile-and-link 363
cgen-add! 327
cgen-body 325
cgen-box-expr 331
cgen-cexpr 328
cgen-current-unit 324
cgen-decl 325
cgen-emit-body 327
cgen-emit-c 324
cgen-emit-decl 327
cgen-emit-h 324
cgen-emit-init 327
cgen-emit-xtrn 327
cgen-extern 325
cgen-init 325
cgen-literal 328
cgen-pred-expr 331
cgen-safe-comment 325
cgen-safe-name 325
cgen-safe-name-friendly 325
cgen-safe-string 325
cgen-type-from-name 331
cgen-unbox-expr 331
cgen-unit-c-file 324
cgen-unit-h-file 324
cgen-unit-init-name 325
cgen-with-cpp-condition 326
cgi-add-temporary-file 822
cgi-get-metavariable 820
cgi-get-parameter 821
cgi-header 821
cgi-main 821
cgi-metavariables 820
cgi-output-character-encoding 821
cgi-parse-parameters 820
cgi-temporary-files 822
cgi-test-environment-ref 824
change-class 297
change-object-class 297
char->integer 146
char->ucs 146
char-alphabetic? 144
char-ci-hash 102
char-ci<=? 144
char-ci<? 144
char-ci=? 144
char-ci>=? 144
char-ci>? 144
char-downcase 146
char-east-asian-width 476
char-foldcase 146
char-general-category 144
char-hash 102
char-lower-case? 144
char-numeric? 144
char-ready? 228
char-set 152
char-set->list 535
char-set->sre 560
char-set->string 535
char-set-adjoin 535
char-set-adjoin! 535
char-set-any 535
char-set-complement 152
char-set-complement! 152
char-set-contains? 152
char-set-copy 152
char-set-count 535
char-set-cursor 534
char-set-cursor-next 534
char-set-delete 536
char-set-delete! 536
char-set-diff+intersection 536
char-set-diff+intersection! 536
char-set-difference 536
char-set-difference! 536
char-set-every 535
char-set-filter 533
char-set-filter! 533
char-set-fold 535
char-set-for-each 535
char-set-hash 534
char-set-immutable? 152
char-set-intersection 536
char-set-intersection! 536
char-set-map 535
char-set-ref 534
char-set-size 152
char-set-unfold 535
char-set-unfold! 535
char-set-union 536
char-set-union! 536
char-set-xor 536
char-set-xor! 536
char-set<= 534
char-set= 534
char-set? 152
char-titlecase 146
char-upcase 146
char-upper-case? 144
char-whitespace? 144
char<=? 144
char<? 144
char=? 144
char>=? 144
char>? 144
char? 144
chars$ 661
check-directory-tree 695
check-substring-spec 600
chibi-test 644
chready? 775

circular-generator ... 373
circular-list ... 513
circular-list? ... 126
cise-ambient-copy ... 335
cise-ambient-decl-strings ... 335
cise-default-ambient ... 335
cise-lookup-macro ... 336
cise-register-macro! ... 336
cise-render ... 335
cise-render-rec ... 335
cise-render-to-string ... 335
cise-translate ... 336
clamp ... 117
class-direct-methods ... 291
class-direct-slots ... 291
class-direct-subclasses ... 291
class-direct-supers ... 291
class-name ... 290
class-of ... 96
class-post-initialize ... 303
class-precedence-list ... 290
class-slot-accessor ... 291
class-slot-bound? ... 297
class-slot-definition ... 291
class-slot-ref ... 297
class-slot-set! ... 297
class-slots ... 291
clear-screen ... 775
clear-to-eol ... 775
clear-to-eos ... 775
close-input-port ... 219
close-output-port ... 219
close-port ... 219
code ... 591
codepoints->grapheme-clusters ... 474
codepoints->words ... 474
codepoints-downcase ... 475
codepoints-foldcase ... 475
codepoints-titlecase ... 475
codepoints-upcase ... 475
coerce-to ... 349
combinations ... 794
combinations* ... 794
combinations*-for-each ... 794
combinations-for-each ... 794
combinations-of ... 663
combine-hash-value ... 102
command-line ... 248
common-prefix ... 447
common-prefix-to ... 447
comparator-check-type ... 105
comparator-compare ... 106
comparator-comparison-procedure ... 105
comparator-comparison-procedure? ... 627
comparator-equal? ... 627
comparator-equality-predicate ... 105
comparator-flavor ... 105
comparator-hash ... 106
comparator-hash-function ... 105
comparator-hash-function? ... 627
comparator-hashable? ... 105
comparator-max ... 637
comparator-min ... 637
comparator-min-in-list ... 637
comparator-ordered? ... 105
comparator-ordering-predicate ... 105
comparator-register-default! ... 106
comparator-test-type ... 105
comparator-type-test-procedure ... 105, 627
comparator? ... 104
compare ... 99
complement ... 191
complete-sexp? ... 391
complex? ... 109
compose ... 191
compute-cpl ... 303
compute-get-n-set ... 303, 305
compute-slot-accessor ... 305
compute-slots ... 303, 305
concatenate ... 134
concatenate! ... 134
cond ... 49, 332
cond-expand ... 68
cond-list ... 126
condition ... 216
condition-has-type? ... 215
condition-ref ... 215
condition-type? ... 215
condition-variable-broadcast! ... 465
condition-variable-name ... 465
condition-variable-signal! ... 465
condition-variable-specific ... 465
condition-variable-specific-set! ... 465
condition-variable? ... 465
condition? ... 215
connection-address-name ... 366
connection-close ... 366
connection-input-port ... 366
connection-output-port ... 366
connection-peer-address ... 365
connection-self-address ... 365
connection-shutdown ... 366
cons ... 126
cons* ... 126
console-device ... 698
construct-cookie-string ... 717
construct-css ... 826
construct-edn ... 781
construct-edn-string ... 781
construct-json ... 729
construct-json-string ... 729
continue ... 333
continued-fraction ... 116
copy-bit ... 122
copy-bit-field ... 122
copy-directory* ... 695
copy-file ... 699
copy-port ... 220
copy-queue ... 658
copy-time ... 601
cos ... 117
cosh ... 117
count ... 133
count$ ... 190
count-accumulator ... 544
cpu-architecture ... 625
crc32 ... 746
create-directory* ... 694
create-directory-tree ... 695
csv-rows->tuples ... 777

current-class-of ... 297
current-country ... 607
current-date ... 601
current-directory ... 693
current-dynamic-extent ... 636
current-error-port ... 218
current-exception-handler ... 211
current-input-port ... 218
current-jiffy ... 511
current-julian-day ... 601
current-language ... 607
current-load-history ... 240
current-load-next ... 240
current-load-path ... 240
current-load-port ... 240
current-locale-details ... 607
current-modified-julian-day ... 601
current-module ... 73
current-output-port ... 218
current-second ... 511
current-thread ... 460
current-time ... 271, 601
cursor-down/scroll-up ... 775
cursor-up/scroll-down ... 775
cut ... 44
cute ... 44

D

d ... 385
date->julian-day ... 603
date->modified-julian-day ... 603
date->rfc822-date ... 716
date->string ... 604
date->time-monotonic ... 603
date->time-tai ... 603
date->time-utc ... 603
date-day ... 602
date-hour ... 602
date-minute ... 602
date-month ... 603
date-nanosecond ... 602
date-second ... 602
date-week-day ... 603
date-week-number ... 603
date-year ... 603
date-year-day ... 603
date-zone-offset ... 603
date? ... 602
dbi-close ... 678, 679, 680, 682
dbi-connect ... 678
dbi-do ... 680, 682
dbi-escape-sql ... 680, 682
dbi-execute ... 679
dbi-execute-using-connection ... 681
dbi-list-drivers ... 678
dbi-make-connection ... 681
dbi-make-driver ... 678
dbi-open? ... 678, 679, 680, 682
dbi-parse-dsn ... 682
dbi-prepare ... 678, 681
dbi-prepare-sql ... 682
dbm-close ... 684
dbm-closed? ... 685
dbm-db-copy ... 686
dbm-db-exists? ... 686
dbm-db-move ... 686
dbm-db-remove ... 686
dbm-delete! ... 685
dbm-exists? ... 685
dbm-fold ... 685
dbm-for-each ... 685
dbm-get ... 685
dbm-map ... 686
dbm-open ... 684
dbm-put! ... 685
dbm-type->class ... 685
dcgettext ... 784
debug-funcall ... 277
debug-label ... 278
debug-print ... 277
debug-print-width ... 277
debug-source-info ... 277
dec! ... 49
declare-bundle! ... 607
declare-cfn ... 334, 339
declare-cvar ... 334, 339
declcode ... 339
decode-float ... 119
decompose-path ... 696
default-endian ... 123
default-hash ... 100
default-sizer ... 664
default-tls-class ... 737
define ... 61
define-cclass ... 337
define-cfn ... 334, 339
define-cgeneric ... 337
define-cise-expr ... 336, 339
define-cise-macro ... 336
define-cise-stmt ... 336, 339
define-cise-toplevel ... 336
define-class ... 286
define-cmethod ... 337
define-condition-type ... 215
define-constant ... 63, 339
define-cproc ... 336
define-cptr ... 338
define-ctype ... 334, 339
define-cvar ... 334, 339
define-dict-interface ... 368
define-enum ... 339
define-enum-conditionally ... 339
define-generic ... 299
define-in-module ... 65
define-inline ... 63
define-library ... 503
define-macro ... 90
define-method ... 299
define-module ... 73
define-reader-ctor ... 230
define-record-type ... 434
define-stream ... 813
define-symbol ... 338
define-syntax ... 83
define-type ... 336
define-values ... 63
define-variable ... 339
deflate-string ... 746
deflating-port-full-flush ... 746

delay 199
delay-force 509
delete 134
delete! 134
delete$ 191
delete-directory* 694
delete-duplicates 134
delete-duplicates! 134
delete-file 700
delete-files 701
delete-hook! 384
delete-keyword 140
delete-keyword! 140
delete-keywords 140
delete-keywords! 140
delete-neighbor-dups 446
delete-neighbor-dups! 446
delete-neighbor-dups-squeeze! 447
denominator 116
dequeue! 658
dequeue-all! 658
dequeue/wait! 659
describe 385
determinant 321
determinant! 321
dgettext 784
dict->alist 368
dict-clear! 367
dict-comparator 367
dict-delete! 367
dict-exists? 367
dict-fold 367
dict-fold-right 367
dict-for-each 368
dict-get 367
dict-keys 368
dict-map 368
dict-pop! 368
dict-push! 368
dict-put! 367
dict-update! 368
dict-values 368
diff 779
diff-report 779
digest 796
digest-final! 796
digest-hexify 796
digest-string 796
digest-update! 796
digit->integer 146
digit-value 507
directory-fold 694
directory-list 693
directory-list2 693
disasm 277
display 234
div 114
div-and-mod 114
div0 114
div0-and-mod0 114
dl-distance 800
dl-distances 800
do 56
do-ec 611
do-generator 383
do-pipeline 426
do-process 421
do-process! 421
dolist 57, 333
dopairs 333
dotimes 57, 333
dotted-list? 126
drop 128
drop* 129
drop-right 129
drop-right! 129
drop-right* 129
drop-while 516
dynamic-extent? 636
dynamic-lambda 636
dynamic-load 241
dynamic-wind 196

E

eager 199
ecase 50
ed 386
edn-equal? 781
edn-map 781
edn-object-handler 782
edn-object-payload 782
edn-object-tag 782
edn-object? 782
edn-set 781
edn-symbol-basename 782
edn-symbol-prefix 782
edn-valid-symbol-name? 782
edn-write 782
eighth 513
emergency-exit 510
encode-float 119
end-of-char-set? 535
endianness 620
enqueue! 658
enqueue-unique! 658
enqueue/wait! 659
environment 508
eof-object 228
eof-object? 228
ephemeron-broken? 551
ephemeron-datum 551
ephemeron-key 551
ephemeron? 551
eq-compare 100
eq-hash 100
eq? 97
equal? 98
eqv-hash 100
eqv? 97
er-macro-transformer 86
error 207
error-object-irritants 506
error-object-message 506
error-object? 506
errorf 208
euclidean-quotient 574
euclidean-remainder 574
euclidean/ 574
eval 216, 508

even? 111
every 133
every$ 191
every-bit-set? 576
every-in-queue 659
every-pred 192
every?-ec 612
exact 119
exact->inexact 120
exact-integer-sqrt 118
exact-integer? 110
exact? 110
exit 246
exit-handler 247
exp 117
expand-path 696
expand-template-file 791
expand-template-string 791
export 73
export-all 74
expt 118
expt-mod 118
extend 76
extract-condition 216

F

f16? 477
f16array 317
f16vector 478
f16vector->list 484
f16vector->vector 484
f16vector-add 487
f16vector-add! 487
f16vector-append 482
f16vector-append-subvectors 482
f16vector-clamp! 488
f16vector-compare 480
f16vector-concatenate 482
f16vector-copy 480
f16vector-copy! 480
f16vector-div 487
f16vector-div! 487
f16vector-dot 487
f16vector-empty? 478
f16vector-fill! 480
f16vector-length 478
f16vector-mul 487
f16vector-mul! 487
f16vector-multi-copy! 481
f16vector-range-check 487
f16vector-ref 479
f16vector-reverse-copy 480
f16vector-set! 479
f16vector-sub 487
f16vector-sub! 487
f16vector-swap! 480
f16vector-unfold 478
f16vector-unfold-right 478
f16vector= 480
f16vector=? 480
f16vector? 477
f32? 477
f32array 317
f32vector 478
f32vector->list 484
f32vector->vector 484
f32vector-add 487
f32vector-add! 487
f32vector-append 482
f32vector-append-subvectors 482
f32vector-clamp 488
f32vector-clamp! 488
f32vector-compare 480
f32vector-concatenate 482
f32vector-copy 480
f32vector-copy! 480
f32vector-div 487
f32vector-div! 487
f32vector-dot 487
f32vector-empty? 478
f32vector-fill! 480
f32vector-length 478
f32vector-mul 487
f32vector-mul! 487
f32vector-multi-copy! 481
f32vector-range-check 487
f32vector-ref 479
f32vector-reverse-copy 480
f32vector-set! 479
f32vector-sub 487
f32vector-sub! 487
f32vector-swap! 480
f32vector-unfold 478
f32vector-unfold-right 478
f32vector= 480
f32vector=? 480
f32vector? 477
f64? 477
f64array 317
f64vector 478
f64vector->list 484
f64vector->vector 484
f64vector-add 487
f64vector-add! 487
f64vector-append 482
f64vector-append-subvectors 482
f64vector-clamp 488
f64vector-clamp! 488
f64vector-compare 480
f64vector-concatenate 482
f64vector-copy 480
f64vector-copy! 480
f64vector-div 487
f64vector-div! 487
f64vector-dot 487
f64vector-empty? 478
f64vector-fill! 480
f64vector-length 478
f64vector-mul 487
f64vector-mul! 487
f64vector-multi-copy! 481
f64vector-range-check 487
f64vector-ref 479
f64vector-reverse-copy 480
f64vector-set! 479
f64vector-sub 487
f64vector-sub! 487
f64vector-swap! 480
f64vector-unfold 478

f64vector-unfold-right 478
f64vector= 480
f64vector=? 480
f64vector? 477
feature-cond 591
features 507
fifth 513
file->byte-generator 375
file->char-generator 375
file->generator 375
file->line-generator 375
file->list 701
file->sexp-generator 375
file->sexp-list 701
file->string 701
file->string-list 701
file-atime 698
file-atime<=? 699
file-atime<? 699
file-atime=? 699
file-atime>=? 699
file-atime>? 699
file-ctime 698
file-ctime<=? 699
file-ctime<? 699
file-ctime=? 699
file-ctime>=? 699
file-ctime>? 699
file-dev 698
file-eq? 698
file-equal? 698
file-eqv? 698
file-error? 506
file-exists? 255
file-filter 691
file-filter-fold 692
file-filter-for-each 692
file-filter-map 692
file-gid 698
file-ino 698
file-is-directory? 255
file-is-executable? 698
file-is-readable? 698
file-is-regular? 255
file-is-symlink? 698
file-is-writable? 698
file-mode 698
file-mtime 698
file-mtime<=? 699
file-mtime<? 699
file-mtime=? 699
file-mtime>=? 699
file-mtime>? 699
file-nlink 698
file-perm 698
file-rdev 698
file-size 698
file-type 698
file-uid 698
files 591
filter 133, 347
filter! 133
filter$ 190
filter-map 133
filter-to 348
find 133, 347
find$ 191
find-file-in-paths 697
find-gauche-package-description 411
find-in-queue 659
find-index 443
find-max 347
find-min 347
find-min&max 347
find-module 76
find-string-from-port? 786
find-tail 133
find-tail$ 191
find-with-index 443
finite? 110
first 513
first-ec 612
first-set-bit 576
fixnum-width 118
fixnum? 111
fixnums 661
fl* 586
fl+ 586
fl+* 586
fl- 586
fl<=? 585
fl<? 585
fl=? 585
fl>=? 585
fl>? 585
flabs 586
flabsdiff 586
flacos 588
flacosh 588
fladjacent 584
flasin 588
flasinh 588
flatan 588
flatanh 588
flcbrt 587
flceiling 587
flcopysign 584
flcos 588
flcosh 588
fldenominator 587
fldenormalized? 586
flerf 589
flerfc 589
fleven? 585
flexp 587
flexp-1 587
flexp2 587
flexponent 584
flexpt 588
flfinite? 586
flfirst-bessel 589
flfloor 587
flgamma 589
flhypot 587
flinfinite? 586
flinteger-exponent 584
flinteger-fraction 584
flinteger? 585
fllog 588
fllog1+ 588

fllog10 588
fllog2 588
flloggamma 589
flmax 586
flmin 586
flnan? 586
flnevative? 585
flnormalized-fraction-exponent 584
flnormalized? 586
flnumerator 587
flodd? 585
flonum 584
flonum-epsilon 112
flonum-min-denormalized 112
flonum-min-normalized 112
flonum? 111
floor 116
floor->exact 117
floor-quotient 115
floor-remainder 115
floor/ 115
flposdiff 587
flpositive? 585
flquotient 588
flremainder 588
flremquo 589
flround 587
flsecond-bessel 589
flsgn 587
flsign-bit 585
flsin 588
flsinh 588
flsqrt 587
flsquare 587
fltan 588
fltanh 588
fltruncate 587
fluid-let 54
flunordered? 585
flush 239
flush-all-ports 239
flush-output-port 507
flzero? 585
fmod 119
fold 131, 345
fold$ 190, 347
fold-ec 612
fold-left 132
fold-right 132, 443
fold-right$ 190
fold-with-index 442
fold2 345
fold3 345
fold3-ec 612
for 333
for-each 131, 333, 346
for-each$ 190, 347
for-each-with-index 443
force 199
format 235, 607
fourth 513
frexp 119
ftp-chdir 719
ftp-current-directory 719
ftp-get 720
ftp-help 719
ftp-list 720
ftp-login 719
ftp-ls 720
ftp-mdtm 720
ftp-mkdir 719
ftp-mtime 720
ftp-name-list 720
ftp-noop 720
ftp-passive? 719
ftp-put 720
ftp-put-unique 720
ftp-quit 719
ftp-remove 719
ftp-rename 721
ftp-rmdir 719
ftp-site 719
ftp-size 720
ftp-stat 719
ftp-system 720
ftp-transfer-type 719
fx* 580
fx*/carry 580
fx+ 580
fx+/carry 580
fx- 580
fx-/carry 580
fx<=? 579
fx<? 579
fx=? 579
fx>=? 579
fx>? 579
fxabs 580
fxand 580
fxarithmetic-shift 580
fxbit-count 580
fxbit-field 580
fxbit-field-rotate 581
fxbit-set? 580
fxcopy-bit 580
fxeven? 579
fxfirst-set-bit 580
fxif 581
fxior 580
fxlength 580
fxmax 579
fxmin 579
fxneg 580
fxnegative? 579
fxnot 580
fxodd? 579
fxpositive? 579
fxquotient 580
fxremainder 580
fxsqrt 580
fxsquare 580
fxxor 580
fxzero? 579

G

gamma ... 118
gappend ... 376
gauche-architecture ... 249
gauche-architecture-directory ... 250
gauche-character-encoding ... 147
gauche-config ... 352
gauche-library-directory ... 249
gauche-package-description-paths ... 411
gauche-site-architecture-directory ... 250
gauche-site-library-directory ... 250
gauche-thread-type ... 457
gauche-version ... 249
gbuffer-filter ... 379
gc ... 275
gc-stat ... 275
gcd ... 115
gcombine ... 378
gconcatenate ... 376
gcons* ... 376
gdbm-close ... 688
gdbm-closed? ... 689
gdbm-delete ... 689
gdbm-errno ... 689
gdbm-exists? ... 689
gdbm-fetch ... 689
gdbm-firstkey ... 689
gdbm-nextkey ... 689
gdbm-open ... 688
gdbm-reorganize ... 689
gdbm-setopt ... 689
gdbm-store ... 689
gdbm-strerror ... 689
gdbm-sync ... 689
gdbm-version ... 689
gdelete ... 378
gdelete-neighbor-dups ... 378
gdrop ... 379
gdrop-while ... 380
generate ... 373
generator ... 376
generator->bytevector ... 382
generator->bytevector! ... 382
generator->ideque ... 542
generator->list ... 381
generator->lseq ... 200
generator->reverse-list ... 381
generator->string ... 381
generator->uvector ... 382
generator->uvector! ... 382
generator->vector ... 381
generator->vector! ... 382
generator-any ... 383
generator-count ... 383
generator-every ... 383
generator-find ... 198
generator-fold ... 197
generator-fold-right ... 198
generator-for-each ... 198
generator-map ... 198
generator-map->list ... 381
generator-unfold ... 383
gensym ... 138
get-environment-variable ... 622
get-environment-variables ... 622
get-f16 ... 639
get-f16be ... 640
get-f16le ... 640
get-f32 ... 639
get-f32be ... 640
get-f32le ... 640
get-f64 ... 640
get-f64be ... 640
get-f64le ... 640
get-keyword ... 140
get-keyword* ... 140
get-optional ... 193
get-output-bytevector ... 506
get-output-string ... 225
get-output-uvector ... 497
get-remaining-input-generator ... 498
get-remaining-input-list ... 497
get-remaining-input-string ... 224
get-s16 ... 639
get-s16be ... 640
get-s16le ... 640
get-s32 ... 639
get-s32be ... 640
get-s32le ... 640
get-s64 ... 639
get-s64be ... 640
get-s64le ... 640
get-s8 ... 639
get-signal-handler ... 264
get-signal-handler-mask ... 264
get-signal-handlers ... 264
get-signal-pending-limit ... 264
get-sint ... 640
get-u16 ... 639
get-u16be ... 640
get-u16le ... 640
get-u32 ... 639
get-u32be ... 640
get-u32le ... 640
get-u64 ... 639
get-u64be ... 640
get-u64le ... 640
get-u8 ... 639
get-uint ... 640
getch ... 774
getter-with-setter ... 48
gettext ... 783
gfilter ... 378
gfilter-map ... 378
gflatten ... 377
ggroup ... 380
gindex ... 381
giota ... 373
glet* ... 382
glet1 ... 383
glob ... 250
glob-fold ... 252
global-variable-bound? ... 77
global-variable-ref ... 77
gmap ... 377
gmap-accum ... 378
gmerge ... 377
goto ... 333
grange ... 373

greatest-fixnum ... 118
gremove ... 378
group-collection ... 348
group-contiguous-sequence ... 445
group-sequence ... 445
grxmatch ... 380
gselect ... 381
gslices ... 380
gstate-filter ... 378
gtake ... 379
gtake* ... 379
gtake-while ... 380
guard ... 208
gunfold ... 375
gzip-decode-string ... 746
gzip-encode-string ... 746

H

has-setter? ... 48
has-windows-console? ... 452
hash ... 102, 539, 620
hash-bound ... 102
hash-by-identity ... 539, 620
hash-salt ... 102
hash-table ... 178, 537
hash-table->alist ... 179
hash-table-clear! ... 180
hash-table-comparator ... 177
hash-table-compare-as-sets ... 181
hash-table-contains? ... 180
hash-table-copy ... 178
hash-table-count ... 538
hash-table-delete! ... 180, 537
hash-table-delete!-r7 ... 180
hash-table-difference! ... 182
hash-table-empty-copy ... 178
hash-table-equivalence-function ... 540, 619
hash-table-exists? ... 180
hash-table-find ... 181, 538
hash-table-fold ... 181, 539
hash-table-for-each ... 181, 539
hash-table-from-pairs ... 177
hash-table-get ... 179
hash-table-hash-function ... 540, 619
hash-table-intern! ... 538
hash-table-intern!-r7 ... 179
hash-table-intersection! ... 182
hash-table-keys ... 181
hash-table-map ... 181, 538
hash-table-map! ... 538
hash-table-map->list ... 538
hash-table-merge! ... 539, 620
hash-table-mutable? ... 177
hash-table-num-entries ... 177
hash-table-pop! ... 180, 538
hash-table-pop!-r7 ... 180
hash-table-prune! ... 539
hash-table-push! ... 180
hash-table-put! ... 179
hash-table-r7 ... 178
hash-table-ref ... 179, 619
hash-table-ref/default ... 179, 619
hash-table-set! ... 179, 619
hash-table-size ... 177, 619
hash-table-type ... 177
hash-table-unfold ... 178
hash-table-union! ... 182
hash-table-update! ... 180, 538, 619
hash-table-update!-r7 ... 181
hash-table-update!/default ... 181, 619
hash-table-values ... 181
hash-table-walk ... 619
hash-table-xor! ... 182
hash-table=? ... 182
hash-table? ... 177
hashmap ... 570
hashmap->alist ... 573
hashmap-adjoin ... 571
hashmap-adjoin! ... 571
hashmap-any? ... 572
hashmap-contains? ... 571
hashmap-copy ... 573
hashmap-count ... 572
hashmap-delete ... 571
hashmap-delete! ... 571
hashmap-delete-all ... 571
hashmap-delete-all! ... 571
hashmap-difference ... 573
hashmap-difference! ... 573
hashmap-disjoint? ... 571
hashmap-empty? ... 570
hashmap-entries ... 572
hashmap-every? ... 572
hashmap-filter ... 573
hashmap-filter! ... 573
hashmap-find ... 572
hashmap-fold ... 572
hashmap-for-each ... 572
hashmap-intern ... 572
hashmap-intern! ... 572
hashmap-intersection ... 573
hashmap-intersection! ... 573
hashmap-key-comparator ... 571
hashmap-keys ... 572
hashmap-map ... 572
hashmap-map->list ... 572
hashmap-partition ... 573
hashmap-partition! ... 573
hashmap-pop ... 572
hashmap-pop! ... 572
hashmap-ref ... 571
hashmap-ref/default ... 571
hashmap-remove ... 573
hashmap-remove! ... 573
hashmap-replace ... 571
hashmap-replace! ... 571
hashmap-search ... 572
hashmap-search! ... 572
hashmap-set ... 571
hashmap-set! ... 571
hashmap-size ... 572
hashmap-unfold ... 570
hashmap-union ... 573
hashmap-union! ... 573
hashmap-update ... 572
hashmap-update! ... 572
hashmap-update!/default ... 572
hashmap-update/default ... 572
hashmap-values ... 572

hashmap-xor ... 573
hashmap-xor! ... 573
hashmap<=? ... 573
hashmap<? ... 573
hashmap=? ... 573
hashmap>=? ... 573
hashmap>? ... 573
hashmap? ... 570
hide-cursor ... 775
hmac-digest ... 721
hmac-digest-string ... 721
hmac-final! ... 721
hmac-update! ... 721
home-directory ... 693
hook->list ... 384
hook-add! ... 637
hook-delete! ... 637
hook-empty? ... 384
hook-reset! ... 637
hook-run ... 637
hook? ... 384
html-doctype ... 785
html-escape ... 784
html-escape-string ... 784
html:a ... 785
html:abbr ... 785
html:acronym ... 785
html:address ... 785
html:area ... 785
html:b ... 785
html:base ... 785
html:bdo ... 785
html:big ... 785
html:blockquote ... 785
html:body ... 785
html:br ... 785
html:button ... 785
html:caption ... 785
html:cite ... 785
html:code ... 785
html:col ... 785
html:colgroup ... 785
html:dd ... 785
html:del ... 785
html:dfn ... 785
html:div ... 785
html:dl ... 785
html:dt ... 785
html:em ... 785
html:fieldset ... 785
html:form ... 785
html:frame ... 785
html:frameset ... 785
html:h1 ... 785
html:h2 ... 785
html:h3 ... 785
html:h4 ... 785
html:h5 ... 785
html:h6 ... 785
html:head ... 785
html:hr ... 785
html:html ... 785
html:i ... 785
html:iframe ... 785
html:img ... 785
html:input ... 785
html:ins ... 785
html:kbd ... 785
html:label ... 785
html:legend ... 785
html:li ... 785
html:link ... 785
html:map ... 785
html:meta ... 785
html:noframes ... 785
html:noscript ... 785
html:object ... 785
html:ol ... 785
html:optgroup ... 785
html:option ... 785
html:p ... 785
html:param ... 785
html:pre ... 785
html:q ... 785
html:samp ... 785
html:script ... 785
html:select ... 785
html:small ... 785
html:span ... 785
html:strong ... 785
html:style ... 785
html:sub ... 785
html:sup ... 785
html:table ... 785
html:tbody ... 785
html:td ... 785
html:textarea ... 785
html:tfoot ... 785
html:th ... 785
html:thead ... 785
html:title ... 785
html:tr ... 785
html:tt ... 785
html:ul ... 785
html:var ... 785
http-compose-form-data ... 726
http-compose-query ... 725
http-default-redirect-handler ... 725
http-delete ... 722
http-get ... 722
http-head ... 722
http-post ... 722
http-proxy ... 725
http-put ... 722
http-secure-connection-available? ... 726
http-status-code->description ... 726
http-user-agent ... 725

I

icmp-packet-code 727
icmp-packet-ident 727
icmp-packet-sequence 727
icmp-packet-type 727
icmp4-describe-packet 727
icmp4-exceeded-code->string 727
icmp4-fill-checksum! 727
icmp4-fill-echo! 727
icmp4-message-type->string 727
icmp4-parameter-code->string 727
icmp4-redirect-code->string 727
icmp4-router-code->string 727
icmp4-security-code->string 727
icmp4-unreach-code->string 727
icmp6-describe-packet 727
icmp6-exceeded-code->string 727
icmp6-fill-echo! 727
icmp6-message-type->string 727
icmp6-parameter-code->string 727
icmp6-unreach-code->string 727
identifier->symbol 143
identifier? 143
identity-array 320
ideque 540
ideque->generator 542
ideque->list 542
ideque-add-back 540
ideque-add-front 540
ideque-any 542
ideque-append 541
ideque-append-map 541
ideque-back 540
ideque-break 542
ideque-drop 541
ideque-drop-right 541
ideque-drop-while 542
ideque-drop-while-right 542
ideque-empty? 540
ideque-every 542
ideque-filter 541
ideque-filter-map 541
ideque-find 541
ideque-find-right 541
ideque-fold 541
ideque-fold-right 541
ideque-for-each 541
ideque-for-each-right 541
ideque-front 540
ideque-length 541
ideque-map 541
ideque-partition 541
ideque-ref 541
ideque-remove 541
ideque-remove-back 540
ideque-remove-front 540
ideque-reverse 541
ideque-span 542
ideque-split-at 541
ideque-tabulate 540
ideque-take 541
ideque-take-right 541
ideque-take-while 542
ideque-take-while-right 542
ideque-unfold 540
ideque-unfold-right 540
ideque-zip 541
ideque= 541
ideque? 540
if 49, 332, 339, 614
if-car-sxpath 763
if-let1 53
if-not=? 630
if-sxpath 763
if<=? 630
if<? 630
if=? 630
if>=? 630
if>? 630
if3 630
imag-part 118
imap-delete 656
imap-empty? 655
imap-exists? 655
imap-get 656
imap-max 656
imap-min 656
imap-put 656
imap? 655
implementation-name 625
implementation-version 625
import 74, 502
in-closed-interval? 631
in-closed-open-interval? 631
in-open-closed-interval? 631
in-open-interval? 631
inc! 49
include 67, 339
include-ci 67
inet-address->string 400
inet-checksum 408
inet-string->address 400
inet-string->address! 400
inexact 119
inexact->exact 120
inexact? 110
infinite? 110
inflate-string 746
inflate-sync 746
info 385
info-search 386
initcode 339
initialize 295, 303
input-port-open? 506
input-port? 218
instance-of 396
int16s 661
int32s 661
int64s 661
int8s 661
integer->char 146
integer->digit 147
integer->list 617
integer-length 122
integer-range->char-set 534
integer-range->char-set! 534
integer-valued? 110
integer? 109
integers$ 661

integers-between$ 661
integers-geometric$ 662
integers-poisson$ 662
interaction-environment 216
intersperse 130
iota 126
ip-destination-address 728
ip-header-length 728
ip-protocol 625, 728
ip-source-address 728
ip-version 728
is-a? 96
isomorphic? 797
iterator->stream 812

J

jacobi 708
jiffies-per-second 511
job-acknowledge-time 646
job-finish-time 646
job-result 646
job-start-time 646
job-status 645
job-wait 646
job? 645
join-timeout-exception? 468
json-array-handler 729
json-object-handler 729
json-special-handler 729
julian-day->date 603
julian-day->time-monotonic 603
julian-day->time-tai 603
julian-day->time-utc 603

K

keyword->string 140
keyword? 139
kmp-step 600

L

l-distance 800
l-distances 800
label 333
lambda 42
lappend 387
lappend-map 388
last 130
last-ec 612
last-pair 130
lazy 199
lazy-size-of 349
lcm 115
lconcatenate 388
lcons 201
lcons* 202
lcs 798
lcs-edit-list 799
lcs-fold 799
lcs-with-positions 798
ldexp 119
least-fixnum 118
legacy-hash 101
length 128
length+ 128
length<=? 128
length<? 128
length=? 128
length>=? 128
length>? 128
let 52, 57, 590
let* 52, 332
let*-values 55
let-args 413
let-keywords 193
let-keywords* 194
let-optionals* 192
let-string-start+end 600
let-syntax 83
let-values 55
let/cc 196
let1 52
letrec 52
letrec* 52
letrec-syntax 83
lfilter 389
lfilter-map 389
lgamma 118
library-exists? 244
library-fold 243
library-for-each 244
library-has-module? 244
library-map 244
linterweave 388
liota 202
list 126
list* 126
list->@vector 484
list->bag 530
list->bag! 530
list->bits 578
list->c128vector 484
list->c32vector 484
list->c64vector 484
list->char-set 533
list->char-set! 533
list->f16vector 484
list->f32vector 484
list->f64vector 484
list->file 701
list->generator 373
list->hook 637
list->hook! 637
list->ideque 542
list->integer 617
list->queue 658
list->s16vector 484
list->s32vector 484
list->s64vector 484
list->s8vector 484
list->set 530
list->set! 530
list->stream 812
list->string 158
list->sys-fdset 274
list->u16vector 484
list->u32vector 484

list->u64vector 484
list->u8vector 484
list->vector 174
list-accumulator 543
list-copy 126
list-delete-neighbor-dups 523
list-delete-neighbor-dups! 523
list-ec 611
list-index 516
list-merge 522
list-merge! 522
list-queue 548
list-queue-add-back! 549
list-queue-add-front! 549
list-queue-append 549
list-queue-append! 550
list-queue-back 548
list-queue-concatenate 550
list-queue-copy 548
list-queue-empty? 548
list-queue-fist-last 549
list-queue-for-each 550
list-queue-front 548
list-queue-list 549
list-queue-map 550
list-queue-map! 550
list-queue-remove-all! 549
list-queue-remove-back! 549
list-queue-remove-front! 549
list-queue-set-list! 549
list-queue-unfold 548
list-queue-unfold-right 548
list-queue? 548
list-ref 130
list-set! 130
list-sort 522
list-sort! 522
list-sorted? 522
list-stable-sort 522
list-stable-sort! 522
list-tabulate 513
list-tail 129
list= 513
list? 125
listener-read-handler 391
listener-show-prompt 391
lists-of 664
llist* 202
lmap 387
lmap-accum 387
load 239, 510
load-bundle! 607
load-from-port 240
localized-template 607
log 117
log-default-drain 394
log-format 394
log-open 394
log2-binary-factors 617
logand 121, 334
logand= 335
logbit? 122
logcount 122
logior 121, 334
logior= 335
lognot 121, 334
logtest 121
logxor 121, 334
logxor= 335
loop 333
lrange 202
lrxmatch 389
lseq->generator 545
lseq-any 546
lseq-append 545
lseq-car 545
lseq-cdr 545
lseq-drop 545
lseq-drop-while 546
lseq-every 546
lseq-filter 546
lseq-find 546
lseq-find-tail 546
lseq-first 545
lseq-for-each 545
lseq-index 546
lseq-length 545
lseq-map 545
lseq-member 546
lseq-memq 546
lseq-memv 546
lseq-realize 545
lseq-remove 546
lseq-rest 545
lseq-take 545
lseq-take-while 546
lseq-zip 545
lseq=? 545
lseq? 545
lset-adjoin 516
lset-diff+intersection 517
lset-diff+intersection! 517
lset-difference 516
lset-difference! 517
lset-intersection 516
lset-intersection! 517
lset-union 516
lset-union! 517
lset-xor 517
lset-xor! 517
lset<= 516
lset= 516
lslices 389
lstate-filter 389
ltake 389
ltake-while 389
lunfold 387

M

machine-name 626
macroexpand 93
macroexpand-1 93
macroexpand-all 94
magnitude 118
make 295, 307
make-@vector 478
make-accumulator 543
make-array 316
make-bimap 369
make-binary-heap 652
make-bitwise-generator 578
make-blob 620
make-byte-string 155
make-bytevector 490
make-bytevector-comparator 628
make-c128vector 478
make-c32vector 478
make-c64vector 478
make-car-comparator 628
make-cdr-comparator 628
make-client-socket 401, 623
make-comparator 103, 627
make-comparator/compare 104
make-comparison< 630
make-comparison<= 630
make-comparison=/< 630
make-comparison=/> 630
make-comparison> 630
make-comparison>= 630
make-compound-condition 216
make-condition 215
make-condition-type 215
make-condition-variable 465
make-coroutine-generator 376
make-csv-header-parser 778
make-csv-reader 776
make-csv-record-parser 779
make-csv-writer 776
make-date 602
make-debug-comparator 630
make-default-comparator 108
make-default-console 774
make-directory* 694
make-edn-object 782
make-empty-attlist 749
make-ephemeron 551
make-eq-comparator 108
make-eqv-comparator 108
make-f16array 316
make-f16vector 478
make-f32array 316
make-f32vector 478
make-f64array 316
make-f64vector 478
make-fifo-cache 650
make-fllog-base 588
make-flonum 584
make-for-each-generator 376
make-gauche-package-description 411
make-gettext 784
make-glob-fs-fold 252
make-grapheme-cluster-breaker 474
make-grapheme-cluster-reader 475
make-hash-table 177, 537, 619
make-hashmap-comparator 574
make-hook 384
make-ideque 654
make-imap 655
make-improper-list-comparator 629
make-inexact-real-comparator 627
make-iota-generator 376
make-key-comparator 108
make-keyword 140
make-kmp-restart-vector 600
make-list 126
make-list-comparator 628
make-list-queue 548
make-listwise-comparator 629
make-lru-cache 650
make-mapping-comparator 570
make-module 76
make-mtqueue 657
make-mutex 463
make-option-parser 417
make-overflow-doubler 665
make-packer 643
make-pair-comparator 629
make-parameter 412
make-polar 118
make-process-connection 433
make-promise 509
make-queue 657
make-random-source 605
make-range-generator 376
make-rbtree 313
make-record-type 806
make-rectangular 118
make-refining-comparator 630
make-reverse-comparator 108, 630
make-ring-buffer 665
make-rtd 437
make-s16array 316
make-s16vector 478
make-s32array 316
make-s32vector 478
make-s64array 316
make-s64vector 478
make-s8array 316
make-s8vector 478
make-selecting-comparator 629
make-server-socket 401, 623
make-server-sockets 402
make-sockaddrs 399
make-socket 403
make-sparse-matrix 670
make-sparse-table 671
make-sparse-vector 668
make-stream 811
make-string 155
make-template-environment 791
make-text-progress-bar 788
make-thread 460
make-thread-pool 646
make-time 601
make-time-result 469
make-tree-map 182
make-trie 673

make-ttl-cache ... 650
make-ttlr-cache ... 650
make-tuple-comparator ... 108
make-u16array ... 316
make-u16vector ... 478
make-u32array ... 316
make-u32vector ... 478
make-u64array ... 316
make-u64vector ... 478
make-u8array ... 316
make-u8vector ... 478
make-unfold-generator ... 376
make-uvector ... 478
make-vector ... 174
make-vector-comparator ... 628
make-vectorwise-comparator ... 629
make-weak-vector ... 187
make-word-breaker ... 474
make-word-reader ... 475
make-write-controls ... 233
make-xml-token ... 749
make<=? ... 630
make<? ... 630
make=? ... 630
make>=? ... 630
make>? ... 630
map ... 131, 345
map! ... 515
map$ ... 190, 347
map* ... 131
map-accum ... 346
map-in-order ... 515
map-to ... 346
map-to-with-index ... 443
map-union ... 759
map-with-index ... 443
mapping ... 563
mapping->alist ... 568
mapping-adjoin ... 565
mapping-adjoin! ... 565
mapping-any? ... 567
mapping-catenate ... 569
mapping-catenate! ... 569
mapping-contains? ... 564
mapping-copy ... 568
mapping-count ... 567
mapping-delete ... 566
mapping-delete! ... 566
mapping-delete-all ... 566
mapping-delete-all! ... 566
mapping-difference ... 569
mapping-difference! ... 569
mapping-disjoint? ... 564
mapping-empty? ... 564
mapping-entries ... 567
mapping-every? ... 567
mapping-filter ... 568
mapping-filter! ... 568
mapping-find ... 567
mapping-fold ... 568
mapping-fold/reverse ... 568
mapping-for-each ... 567
mapping-intern ... 566
mapping-intern! ... 566
mapping-intersection ... 569
mapping-intersection! ... 569
mapping-key-comparator ... 565
mapping-key-predecessor ... 569
mapping-key-successor ... 569
mapping-keys ... 567
mapping-map ... 567
mapping-map->list ... 568
mapping-map/monotone ... 567
mapping-map/monotone! ... 567
mapping-max-entry ... 569
mapping-max-key ... 569
mapping-max-value ... 569
mapping-min-entry ... 569
mapping-min-key ... 569
mapping-min-value ... 569
mapping-partition ... 568
mapping-partition! ... 568
mapping-pop ... 567
mapping-pop! ... 567
mapping-range< ... 569
mapping-range<! ... 569
mapping-range<= ... 569
mapping-range<=! ... 569
mapping-range= ... 569
mapping-range=! ... 569
mapping-range> ... 569
mapping-range>! ... 569
mapping-range>= ... 569
mapping-range>=! ... 569
mapping-ref ... 564
mapping-ref/default ... 565
mapping-remove ... 568
mapping-remove! ... 568
mapping-replace ... 565
mapping-replace! ... 565
mapping-search ... 567
mapping-search! ... 567
mapping-set ... 565
mapping-set! ... 565
mapping-size ... 567
mapping-split ... 569
mapping-split! ... 569
mapping-unfold ... 563
mapping-unfold/ordered ... 564
mapping-union ... 568
mapping-union! ... 568
mapping-update ... 566
mapping-update! ... 566
mapping-update!/default ... 567
mapping-update/default ... 567
mapping-values ... 567
mapping-xor ... 569
mapping-xor! ... 569
mapping/ordered ... 564
mapping<=? ... 568
mapping<? ... 568
mapping=? ... 568
mapping>=? ... 568
mapping>? ... 568
mapping? ... 564
match ... 801
match-define ... 802
match-lambda ... 801
match-lambda* ... 802
match-let ... 802

match-let* 802
match-let1 802
match-letrec 802
max 111
max-ec 611
mc-factorize 707
md5-digest 730
md5-digest-string 730
member 135
member$ 191
memq 135
memv 135
merge 245
merge! 245
message-type 625
method-more-specific? 307
miller-rabin-prime? 706
mime-body->file 734
mime-body->string 734
mime-compose-message 735
mime-compose-message-string 735
mime-compose-parameters 731
mime-decode-text 732
mime-decode-word 731
mime-encode-text 732
mime-encode-word 732
mime-make-boundary 735
mime-parse-content-disposition 731
mime-parse-content-type 731
mime-parse-message 733
mime-parse-parameters 731
mime-parse-version 730
mime-retrieve-body 734
min 111
min&max 111
min-ec 611
mod 114
mod0 114
modf 119
modified-julian-day->date 603
modified-julian-day->time-monotonic 603
modified-julian-day->time-tai 603
modified-julian-day->time-utc 603
modifier 442
module-exports 77
module-imports 77
module-name 77
module-name->path 78
module-parents 77
module-precedence-list 77
module-reload-rules 439
module-table 77
module? 76
modulo 113
move-cursor-to 775
move-file 700
mt-random-fill-f32vector! 705
mt-random-fill-f64vector! 705
mt-random-fill-u32vector! 705
mt-random-get-state 705
mt-random-integer 705
mt-random-real 705
mt-random-real0 705
mt-random-set-seed! 705
mt-random-set-state! 705
mtqueue-max-length 657
mtqueue-num-waiting-readers 657
mtqueue-room 657
mtqueue? 657
mutex-lock! 463
mutex-locker 464
mutex-name 463
mutex-specific 463
mutex-specific-set! 463
mutex-state 463
mutex-unlock! 464
mutex-unlocker 464
mutex? 463

N

naive-factorize 707
nan? 110
native-endian 123
ndbm-clear-error 690
ndbm-close 690
ndbm-closed? 690
ndbm-delete 690
ndbm-error 690
ndbm-fetch 690
ndbm-firstkey 690
ndbm-nextkey 690
ndbm-open 690
ndbm-store 690
negative? 110
nested 614
newline 235
next-method 301
next-token 787
next-token-of 787
ngettext 784
ninth 513
node-closure 761
node-eq? 759
node-equal? 759
node-join 760
node-or 761
node-pos 759
node-reduce 760
node-reverse 760
node-self 760
node-trace 760
nodeset? 758
not 124, 334, 614
not-pair? 513
ntype-names?? 758
ntype-namespace-id?? 758
ntype?? 758
null-device 698
null-environment 216
null-generator 373
null-list? 125
null? 125
number->string 121
number-hash 102
number? 109
numerator 116

O

object-apply ... 195
object-compare ... 99, 100
object-equal? ... 98, 104
object-hash ... 101
object-isomorphic? ... 798
odbm-close ... 691
odbm-delete ... 691
odbm-fetch ... 691
odbm-firstkey ... 691
odbm-init ... 691
odbm-nextkey ... 691
odbm-store ... 691
odd? ... 111
open-binary-input-file ... 509
open-binary-output-file ... 509
open-coding-aware-port ... 226
open-deflating-port ... 744
open-inflating-port ... 745
open-input-byte-generator ... 498
open-input-byte-list ... 497
open-input-bytevector ... 506
open-input-char-generator ... 498
open-input-char-list ... 497
open-input-conversion-port ... 342
open-input-fd-port ... 224
open-input-file ... 221
open-input-process-port ... 429
open-input-string ... 224
open-input-uvector ... 496
open-output-bytevector ... 506
open-output-conversion-port ... 342
open-output-fd-port ... 224
open-output-file ... 221
open-output-process-port ... 431
open-output-string ... 225
open-output-uvector ... 497
option ... 608
option-names ... 609
option-optional-arg? ... 609
option-processor ... 609
option-required-arg? ... 609
option? ... 609
or ... 51, 334, 614
os-name ... 626
os-version ... 626
output-port-open? ... 506
output-port? ... 218

P

pa$ ... 190
pack ... 641
pair-fold ... 514
pair-fold-right ... 514
pair-for-each ... 333, 515
pair? ... 125
pairs-of ... 663
parameter-observer-add! ... 413
parameter-observer-delete! ... 413
parameter-post-observers ... 413
parameter-pre-observers ... 413
parameterize ... 412
parse-cookie-string ... 717
parse-css ... 826
parse-css-file ... 826
parse-css-selector-string ... 826
parse-edn ... 780
parse-edn* ... 781
parse-edn-string ... 781
parse-json ... 728
parse-json* ... 729
parse-json-string ... 729
parse-options ... 417
partition ... 348, 515
partition! ... 515
partition$ ... 190
partition-to ... 348
path->gauche-package-description ... 411
path->module-name ... 78
path-extension ... 697
path-sans-extension ... 697
path-swap-extension ... 697
peek-byte ... 227
peek-char ... 227
peek-next-char ... 787
peek-u8 ... 506
permutations ... 794
permutations* ... 794
permutations*-for-each ... 794
permutations-for-each ... 794
permutations-of ... 663
permute ... 447
permute! ... 448
permute-to ... 448
pop! ... 48
port->byte-generator ... 375
port->byte-lseq ... 202
port->char-generator ... 375
port->char-lseq ... 202
port->line-generator ... 375
port->list ... 231
port->sexp-generator ... 375
port->sexp-list ... 231
port->sexp-lseq ... 202
port->stream ... 812
port->string ... 231
port->string-list ... 231
port->string-lseq ... 202
port->uvector ... 489
port-buffering ... 220
port-closed? ... 218
port-current-line ... 220
port-fd-dup! ... 224
port-file-number ... 220
port-fold ... 231
port-fold-right ... 231
port-for-each ... 231
port-map ... 231
port-name ... 219
port-seek ... 220
port-tell ... 220
port-type ... 219
port? ... 218
portable-hash ... 100
positive? ... 110
post++ ... 334
post-- ... 334
power-set ... 795

power-set* 795
power-set*-for-each 795
power-set-binary 795
power-set-for-each 795
pprint 234
pre++ 334
pre-- 334
primes 706
print 234
procedure-arity-includes? 194
procedure? 188
process-alive? 428
process-command 427
process-continue 429
process-error 428
process-exit-status 429
process-input 427
process-kill 429
process-list 428
process-output 427
process-output->string 431
process-output->string-list 431
process-pid 427
process-send-signal 429
process-stop 429
process-upstreams 428
process-wait 429
process-wait-any 429
process? 427
product-accumulator 544
product-ec 611
profiler-reset 278
profiler-show 278
profiler-start 278
profiler-stop 278
program 591
promise? 199
proper-list? 126
provide 242
provided? 242
pseudo-rtd 438
push! 48
put-f16! 640
put-f16be! 641
put-f16le! 641
put-f32! 640
put-f32be! 641
put-f32le! 641
put-f64! 640
put-f64be! 641
put-f64le! 641
put-s16! 640
put-s16be! 641
put-s16le! 641
put-s32! 640
put-s32be! 641
put-s32le! 641
put-s64! 640
put-s64be! 641
put-s64le! 641
put-s8! 640
put-sint! 641
put-u16! 640
put-u16be! 641
put-u16le! 641
put-u32! 640
put-u32be! 641
put-u32le! 641
put-u64! 640
put-u64be! 641
put-u64le! 641
put-u8! 640
put-uint! 641
putch 774
putstr 774

Q

quasiquote 59
quasirename 89
query-cursor-position 775
query-screen-size 775
queue->list 659
queue-empty? 657
queue-front 658
queue-internal-list 659
queue-length 657
queue-pop! 658
queue-pop/wait! 659
queue-push! 658
queue-push-unique! 658
queue-push/wait! 659
queue-rear 658
queue? 657
quote 41
quoted-printable-decode 736
quoted-printable-decode-string 736
quoted-printable-encode 736
quoted-printable-encode-string 736
quotient 113
quotient&remainder 114

R

raise 208, 506
raise-continuable 506
random-data-seed 660
random-integer 605
random-real 605
random-source-make-integers 606
random-source-make-reals 606
random-source-pseudo-randomize! 606
random-source-randomize! 606
random-source-state-ref 606
random-source-state-set! 606
random-source? 606
rassoc 136
rassoc-ref 136
rassq 136
rassq-ref 136
rassv 136
rassv-ref 136
rational-valued? 110
rational? 109
rationalize 116
rbtree->alist 313
rbtree-copy 313
rbtree-delete! 313
rbtree-empty? 313

rbtree-exists? 313
rbtree-extract-max! 313
rbtree-extract-min! 313
rbtree-fold 313
rbtree-fold-right 313
rbtree-get 313
rbtree-keys 313
rbtree-max 313
rbtree-min 313
rbtree-num-entries 313
rbtree-pop! 313
rbtree-push! 313
rbtree-put! 313
rbtree-update! 313
rbtree-values 313
rbtree? 313
re-distance 800
re-distances 800
read 227
read-ber-integer 638
read-block 228
read-block! 489
read-byte 227
read-bytevector 507
read-bytevector! 507
read-char 227
read-error? 506
read-eval-print-loop 216
read-f16 639
read-f32 639
read-f64 639
read-from-string 226
read-line 227
read-s16 638
read-s32 638
read-s64 638
read-s8 638
read-sint 638
read-string 228, 787
read-u16 638
read-u32 638
read-u64 638
read-u8 506, 638
read-uint 638
read-uvector 488
read-uvector! 489
read-with-shared-structure 227
read/ss 227
reader-lexical-mode 228
real->rational 120
real-part 118
real-valued? 110
real? 109
reals$ 661
reals-between$ 661
reals-exponential$ 662
reals-normal$ 662
rec 55
receive 54
record-accessor 806
record-constructor 806
record-modifier 806
record-predicate 806
record-rtd 436
record? 436
reduce 133
reduce$ 190
reduce-right 133
reduce-right$ 190
ref 179, 190, 296, 334, 441
reference-barrier 551
referencer 442
regexp 166, 560
regexp->sre 560
regexp->string 166
regexp-ast 173
regexp-compile 172
regexp-compile-sre 173
regexp-extract 561
regexp-fold 560
regexp-match->list 563
regexp-match-count 562
regexp-match-submatch 562
regexp-match-submatch-end 563
regexp-match-submatch-start 562
regexp-match? 562
regexp-matches 560
regexp-matches? 560
regexp-named-groups 166
regexp-num-groups 166
regexp-optimize 172
regexp-parse 172
regexp-parse-sre 172
regexp-partition 561
regexp-quote 170
regexp-replace 169, 561
regexp-replace* 169
regexp-replace-all 169, 562
regexp-replace-all* 169
regexp-search 560
regexp-split 561
regexp-unparse 173
regexp? 166, 560
register-edn-object-handler! 782
regmatch 168
regular-string$ 662
relation-accessor 807
relation-coercer 808
relation-column-getter 808
relation-column-getters 808
relation-column-name? 808
relation-column-names 807
relation-column-setter 808
relation-column-setters 808
relation-deletable? 808
relation-delete! 808
relation-fold 808
relation-insert! 808
relation-insertable? 808
relation-modifier 807
relation-ref 808
relation-rows 808
relation-set! 808
relative-path? 696
relnum-compare 492
reload 439
reload-modified-modules 439
reload-verbose 439
remainder 113
remove 133, 348

remove! 133
remove$ 190
remove-directory* 694
remove-file 700
remove-files 701
remove-from-queue! 659
remove-hook! 384
remove-to 348
report-error 211
report-time-results 470
require 241
require-extension 616
requires 591
reset 418
reset-character-attribute 776
reset-hook! 384
reset-primes 706
reset-terminal 775
resolve-path 696
return 333
reverse 134
reverse! 134
reverse-bit-field 617
reverse-bits->generator 374
reverse-list->@vector 484
reverse-list->c128vector 484
reverse-list->c32vector 484
reverse-list->c64vector 484
reverse-list->f16vector 484
reverse-list->f32vector 484
reverse-list->f64vector 484
reverse-list->s16vector 484
reverse-list->s32vector 484
reverse-list->s64vector 484
reverse-list->s8vector 484
reverse-list->string 592
reverse-list->u16vector 484
reverse-list->u32vector 484
reverse-list->u64vector 484
reverse-list->u8vector 484
reverse-list->vector 175
reverse-list-accumulator 543
reverse-vector->generator 373
reverse-vector->list 522
reverse-vector-accumulator 543
rfc822-atom 714
rfc822-date->date 715
rfc822-dot-atom 714
rfc822-field->tokens 714
rfc822-header->list 712
rfc822-header-ref 713
rfc822-next-token 714
rfc822-parse-date 715
rfc822-quoted-string 714
rfc822-read-headers 712
rfc822-skip-cfws 714
rfc822-write-headers 715
ring-buffer-add-back! 666
ring-buffer-add-front! 666
ring-buffer-back 666
ring-buffer-capacity 666
ring-buffer-empty? 665
ring-buffer-front 666
ring-buffer-full? 665
ring-buffer-num-entries 665
ring-buffer-ref 666
ring-buffer-remove-back! 666
ring-buffer-remove-front! 666
ring-buffer-set! 666
rlet1 53
rotate-bit-field 617
round 116
round->exact 117
round-quotient 574
round-remainder 574
round/ 574
rtd-accessor 438
rtd-all-field-names 437
rtd-constructor 437
rtd-field-mutable? 437
rtd-field-names 437
rtd-mutator 438
rtd-name 437
rtd-parent 437
rtd-predicate 437
rtd? 437
run-cgi-script->header&body 825
run-cgi-script->string 825
run-cgi-script->string-list 825
run-cgi-script->sxml 825
run-hook 384
run-pipeline 426
run-process 421
rx 560
rxmatch 166
rxmatch->string 167
rxmatch-after 167
rxmatch-before 167
rxmatch-case 171
rxmatch-cond 170
rxmatch-end 167
rxmatch-if 170
rxmatch-let 170
rxmatch-named-groups 168
rxmatch-num-matches 168
rxmatch-positions 167
rxmatch-start 167
rxmatch-substring 167
rxmatch-substrings 167

S

s16? 477
s16array 317
s16vector 478
s16vector->list 484
s16vector->vector 484
s16vector-add 487
s16vector-add! 487
s16vector-and 487
s16vector-and! 487
s16vector-append 482
s16vector-append-subvectors 482
s16vector-clamp 488
s16vector-clamp! 488
s16vector-compare 480
s16vector-concatenate 482
s16vector-copy 480
s16vector-copy! 480
s16vector-dot 487

s16vector-empty? 478
s16vector-fill! 480
s16vector-ior 487
s16vector-ior! 487
s16vector-length 478
s16vector-mul 487
s16vector-mul! 487
s16vector-multi-copy! 481
s16vector-range-check 487
s16vector-ref 479
s16vector-reverse-copy 480
s16vector-set! 479
s16vector-sub 487
s16vector-sub! 487
s16vector-swap! 480
s16vector-unfold 478
s16vector-unfold-right 478
s16vector-xor 487
s16vector-xor! 487
s16vector= 480
s16vector=? 480
s16vector? 477
s32? 477
s32array 317
s32vector 478
s32vector->list 484
s32vector->string 486
s32vector->vector 484
s32vector-add 487
s32vector-add! 487
s32vector-and 487
s32vector-and! 487
s32vector-append 482
s32vector-append-subvectors 482
s32vector-clamp 488
s32vector-clamp! 488
s32vector-compare 480
s32vector-concatenate 482
s32vector-copy 480
s32vector-copy! 480
s32vector-dot 487
s32vector-empty? 478
s32vector-fill! 480
s32vector-ior 487
s32vector-ior! 487
s32vector-length 478
s32vector-mul 487
s32vector-mul! 487
s32vector-multi-copy! 481
s32vector-range-check 487
s32vector-ref 479
s32vector-reverse-copy 480
s32vector-set! 479
s32vector-sub 487
s32vector-sub! 487
s32vector-swap! 480
s32vector-unfold 478
s32vector-unfold-right 478
s32vector-xor 487
s32vector-xor! 487
s32vector= 480
s32vector=? 480
s32vector? 477
s64? 477
s64array 317
s64vector 478
s64vector->list 484
s64vector->vector 484
s64vector-add 487
s64vector-add! 487
s64vector-and 487
s64vector-and! 487
s64vector-append 482
s64vector-append-subvectors 482
s64vector-clamp 488
s64vector-clamp! 488
s64vector-compare 480
s64vector-concatenate 482
s64vector-copy 480
s64vector-copy! 480
s64vector-dot 487
s64vector-empty? 478
s64vector-fill! 480
s64vector-ior 487
s64vector-ior! 487
s64vector-length 478
s64vector-mul 487
s64vector-mul! 487
s64vector-multi-copy! 481
s64vector-range-check 487
s64vector-ref 479
s64vector-reverse-copy 480
s64vector-set! 479
s64vector-sub 487
s64vector-sub! 487
s64vector-swap! 480
s64vector-unfold 478
s64vector-unfold-right 478
s64vector-xor 487
s64vector-xor! 487
s64vector= 480
s64vector=? 480
s64vector? 477
s8? 477
s8array 317
s8vector 478
s8vector->list 484
s8vector->string 485
s8vector->vector 484
s8vector-add 487
s8vector-add! 487
s8vector-and 487
s8vector-and! 487
s8vector-append 482
s8vector-append-subvectors 482
s8vector-clamp 488
s8vector-clamp! 488
s8vector-compare 480
s8vector-concatenate 482
s8vector-copy 480
s8vector-copy! 480
s8vector-dot 487
s8vector-empty? 478
s8vector-fill! 480
s8vector-ior 487
s8vector-ior! 487
s8vector-length 478
s8vector-mul 487
s8vector-mul! 487
s8vector-multi-copy! 481

s8vector-range-check 487
s8vector-ref 479
s8vector-reverse-copy 480
s8vector-set! 479
s8vector-sub 487
s8vector-sub! 487
s8vector-swap! 480
s8vector-unfold 478
s8vector-unfold-right 478
s8vector-xor 487
s8vector-xor! 487
s8vector= 480
s8vector=? 480
s8vector? 477
samples$ 662
samples-from 662
save-bundle! 607
scheme-report-environment 216
second 513
seconds->time 271
select-kids 760
select-module 73
selector-add! 440
selector-delete! 440
selector-select 440
sequence->kmp-stepper 444
sequence-contains 443
sequences-of 664
set 525
set! 47, 335
set!-values 47
set->bag 530
set->bag! 530
set->list 530
set-adjoin 527
set-adjoin! 527
set-any? 528
set-box! 547
set-car! 127
set-cdr! 127
set-character-attribute 775
set-contains? 526
set-copy 530
set-count 528
set-delete 527
set-delete! 527
set-delete-all 527
set-delete-all! 527
set-difference 531
set-difference! 531
set-disjoint? 526
set-element-comparator 527
set-empty? 526
set-every? 528
set-filter 529
set-filter! 529
set-find 528
set-fold 529
set-for-each 529
set-intersection 531
set-intersection! 531
set-map 528
set-member 526
set-partition 529
set-partition! 529
set-remove 529
set-remove! 529
set-replace 527
set-replace! 527
set-search! 527
set-signal-handler! 263
set-signal-pending-limit 264
set-size 528
set-time-nanosecond! 601
set-time-second! 601
set-time-type! 601
set-unfold 526
set-union 531
set-union! 531
set-xor 531
set-xor! 531
set<=? 531
set<? 531
set=? 530
set>=? 531
set>? 531
setter 47
seventh 513
sexp-list->file 701
sha1-digest 737
sha1-digest-string 737
sha224-digest 737
sha224-digest-string 737
sha256-digest 737
sha256-digest-string 737
sha384-digest 737
sha384-digest-string 737
sha512-digest 737
sha512-digest-string 737
shape 316
shape-for-each 318
share-array 318
shell-escape-string 432
shell-tokenize-string 432
shift 418
show-cursor 775
shuffle 448
shuffle! 448
shuffle-to 448
shutdown-method 625
simplify-path 696
sin 117
sinh 117
sint-list->blob 622
sixth 513
size-of 349
skip-until 787
skip-while 787
slices 130
slot-bound-using-accessor? 305
slot-bound-using-class? 297
slot-bound? 296
slot-definition-accessor 292
slot-definition-allocation 291
slot-definition-getter 292
slot-definition-name 291
slot-definition-option 292
slot-definition-options 291
slot-definition-setter 292
slot-exists? 296

slot-initialize-using-accessor! 305
slot-missing 297
slot-pop! 296
slot-push! 296
slot-ref 296
slot-ref-using-accessor 305
slot-ref-using-class 297
slot-set! 296
slot-set-using-accessor! 305
slot-set-using-class! 297
slot-unbound 296
small-prime? 706
sockaddr-addr 399
sockaddr-family 398, 399
sockaddr-name 398, 399
sockaddr-port 399
socket-accept 404
socket-address 402
socket-bind 404
socket-buildmsg 405
socket-close 403
socket-connect 404
socket-domain 624
socket-fd 404
socket-getpeername 405
socket-getsockname 405
socket-getsockopt 407
socket-input-port 402
socket-listen 404
socket-merge-flags 624
socket-output-port 402
socket-purge-flags 624
socket-recv 406, 624
socket-recv! 406
socket-recvfrom 406
socket-recvfrom! 406
socket-send 405, 623
socket-sendmsg 405
socket-sendto 405
socket-setsockopt 407
socket-shutdown 405
socket-status 404
socket? 623
sort 245
sort! 245
sort-applicable-methods 307
sort-by 246
sort-by! 246
sorted? 245
source-code 277
source-location 277
span 516
span! 516
sparse-matrix-clear! 670
sparse-matrix-copy 670
sparse-matrix-delete! 670
sparse-matrix-exists? 670
sparse-matrix-fold 671
sparse-matrix-for-each 671
sparse-matrix-inc! 671
sparse-matrix-keys 671
sparse-matrix-map 671
sparse-matrix-num-entries 670
sparse-matrix-pop! 671
sparse-matrix-push! 671
sparse-matrix-ref 670
sparse-matrix-set! 670
sparse-matrix-update! 670
sparse-matrix-values 671
sparse-table-clear! 672
sparse-table-comparator 671
sparse-table-copy 672
sparse-table-delete! 672
sparse-table-exists? 672
sparse-table-fold 672
sparse-table-for-each 672
sparse-table-keys 672
sparse-table-map 672
sparse-table-num-entries 672
sparse-table-pop! 672
sparse-table-push! 672
sparse-table-ref 672
sparse-table-set! 672
sparse-table-update! 672
sparse-table-values 672
sparse-vector-clear! 669
sparse-vector-copy 668
sparse-vector-delete! 668
sparse-vector-exists? 668
sparse-vector-fold 669
sparse-vector-for-each 669
sparse-vector-inc! 669
sparse-vector-keys 669
sparse-vector-map 669
sparse-vector-max-index-bits 668
sparse-vector-num-entries 668
sparse-vector-pop! 669
sparse-vector-push! 669
sparse-vector-ref 668
sparse-vector-set! 668
sparse-vector-update! 669
sparse-vector-values 669
split-at 130
split-at! 130
split-at* 130
sql-tokenize 790
sqrt 117
square 118
srl:display-sxml 773
srl:parameterizable 771
srl:sxml->html 771
srl:sxml->html-noindent 771
srl:sxml->string 773
srl:sxml->xml 771
srl:sxml->xml-noindent 771
ssax:assert-token 754
ssax:complete-start-tag 753
ssax:handle-parsed-entity 752
ssax:make-elem-parser 755
ssax:make-parser 756
ssax:make-pi-parser 755
ssax:ncname-starting-char? 750
ssax:read-attributes 752
ssax:read-cdata-body 751
ssax:read-char-data 754
ssax:read-char-ref 751
ssax:read-external-id 753
ssax:read-markup-token 750
ssax:read-NCName 750
ssax:read-pi-body-as-string 751

ssax:read-QName 750
ssax:resolve-name 752
ssax:reverse-collect-str 757
ssax:reverse-collect-str-drop-ws 757
ssax:scan-Misc 754
ssax:skip-internal-dtd 751
ssax:skip-pi 751
ssax:skip-S 750
ssax:uri-string->symbol 753
ssax:xml->sxml 757
stable-sort 246
stable-sort! 246
stable-sort-by 246
stable-sort-by! 246
standard-error-port 219
standard-input-port 219
standard-output-port 219
stream 546, 810
stream+ 810
stream->list 815
stream->string 815
stream-any 817
stream-append 814
stream-break 817
stream-butlast 817
stream-butlast-n 817
stream-caaaar 816
stream-caaadr 816
stream-caaar 816
stream-caadar 816
stream-caaddr 816
stream-caadr 816
stream-caar 816
stream-cadaar 816
stream-cadadr 816
stream-cadar 816
stream-caddar 816
stream-cadddr 816
stream-caddr 816
stream-cadr 816
stream-car 809
stream-cdaaar 816
stream-cdaadr 816
stream-cdaar 816
stream-cdadar 816
stream-cdaddr 816
stream-cdadr 816
stream-cdar 816
stream-cddaar 816
stream-cddadr 816
stream-cddar 816
stream-cdddar 816
stream-cddddr 816
stream-cdddr 816
stream-cddr 816
stream-cdr 809
stream-concat 814
stream-concatenate 814
stream-cons 809
stream-cons* 812
stream-constant 811
stream-count 817
stream-delay 809
stream-delete 818
stream-delete-duplicates 818
stream-drop 547, 816
stream-drop-safe 816
stream-drop-while 817
stream-eighth 816
stream-every 818
stream-fifth 816
stream-filter 815
stream-find 817
stream-find-tail 817
stream-first 816
stream-fold 814
stream-for-each 814
stream-format 812
stream-fourth 816
stream-from 811
stream-grep 818
stream-index 818
stream-intersperse 817
stream-iota 811
stream-iterate 811
stream-lambda 809
stream-last 817
stream-last-n 817
stream-length 817
stream-length= 817
stream-length>= 817
stream-let 813
stream-lines 815
stream-map 814
stream-match 813
stream-member 818
stream-memq 818
stream-memv 818
stream-ninth 816
stream-null? 809
stream-of 813
stream-pair? 809
stream-partition 815
stream-prefix= 815
stream-range 811
stream-ref 816
stream-remove 815
stream-reverse 817
stream-scan 815
stream-second 816
stream-seventh 816
stream-sixth 816
stream-span 817
stream-split 817
stream-tabulate 811
stream-take 547, 816
stream-take-safe 816
stream-take-while 817
stream-tenth 816
stream-third 816
stream-unfold 810
stream-unfoldn 810
stream-unfolds 811
stream-xcons 812
stream-zip 815
stream= 815
stream? 809
string 155
string->char-set 533
string->char-set! 533

string->date 605
string->file 701
string->generator 373
string->grapheme-clusters 474
string->list 158
string->number 121
string->regexp 165
string->s32vector 485
string->s32vector! 486
string->s8vector 485
string->s8vector! 485
string->stream 812
string->symbol 138
string->u32vector 485
string->u32vector! 486
string->u8vector 485
string->u8vector! 485
string->uninterned-symbol 138
string->utf8 473
string->vector 175
string->words 474
string-accumulator 544
string-any 592
string-append 158
string-append! 632
string-append-ec 611
string-append/shared 597
string-break 635
string-byte-ref 157
string-byte-set! 157
string-ci-hash 102, 539, 620
string-ci< 595
string-ci<= 595
string-ci<=? 158, 475
string-ci<> 595
string-ci<? 158, 475
string-ci= 595
string-ci=? 158, 475
string-ci> 595
string-ci>= 595
string-ci>=? 158, 476
string-ci>? 158, 475
string-compare 594
string-compare-ci 594
string-concatenate 597
string-concatenate-reverse 597
string-concatenate-reverse/shared 597
string-concatenate/shared 597
string-contains 596
string-contains-ci 596
string-contains-right 634
string-copy 159
string-copy! 593
string-count 596
string-delete 599
string-downcase 475, 596
string-downcase! 596
string-drop 593
string-drop-right 593
string-drop-while 635
string-drop-while-right 635
string-ec 611
string-every 592
string-fill! 159
string-filter 599
string-fold 598
string-fold-right 598
string-foldcase 475
string-for-each 505, 598
string-for-each-index 598
string-hash 102, 539, 595, 620
string-hash-ci 595
string-immutable? 154
string-incomplete->complete 161
string-incomplete? 154
string-index 596
string-index-right 596
string-join 159
string-kmp-partial-search 600
string-length 157
string-list->file 701
string-map 505, 597
string-map! 597
string-null? 592
string-pad 593
string-pad-right 593
string-parse-final-start+end 600
string-parse-start+end 600
string-prefix-ci? 595
string-prefix-length 595
string-prefix-length-ci 595
string-prefix? 595
string-ref 157
string-remove 633
string-replace 599
string-replace! 632
string-replicate 634
string-reverse 597
string-reverse! 597
string-scan 160
string-scan-right 160
string-segment 634
string-set! 157
string-size 157
string-skip 596
string-skip-right 596
string-span 635
string-split 160
string-suffix-ci? 595
string-suffix-length 595
string-suffix-length-ci 595
string-suffix? 595
string-tabulate 592
string-take 593
string-take-right 593
string-take-while 634
string-take-while-right 634
string-titlecase 475, 596
string-titlecase! 596
string-tokenize 599
string-tr 793
string-trim 593
string-trim-both 593
string-trim-right 593
string-unfold 598
string-unfold-right 598
string-upcase 475, 596
string-upcase! 596
string-xcopy! 599
string< 594

string<= 594
string<=? 158
string<> 594
string<? 158
string= 594
string=? 158
string> 594
string>= 594
string>=? 158
string>? 158
string? 154
strings-of 664
subseq 442
substring 158
substring-spec-ok? 600
substring/shared 593
subtract-duration 602
subtract-duration! 602
subtype? 97
sum-accumulator 544
sum-ec 611
supported-character-encodings 147
sxml:add-attr 769
sxml:add-attr! 769
sxml:add-aux 769
sxml:add-aux! 769
sxml:add-parents 770
sxml:ancestor 765
sxml:ancestor-or-self 765
sxml:attr 768
sxml:attr->html 770
sxml:attr->xml 770
sxml:attr-as-list 767
sxml:attr-list 764
sxml:attr-list-node 767
sxml:attr-list-u 767
sxml:attr-u 768
sxml:attribute 765
sxml:aux-as-list 767
sxml:aux-list 767
sxml:aux-list-node 767
sxml:aux-list-u 767
sxml:aux-node 767
sxml:aux-nodes 767
sxml:boolean 764
sxml:change-attr 769
sxml:change-attr! 769
sxml:change-attrlist 769
sxml:change-attrlist! 769
sxml:change-content 769
sxml:change-content! 769
sxml:change-name 769
sxml:change-name! 769
sxml:child 765
sxml:child-elements 765
sxml:child-nodes 765
sxml:clean 769
sxml:clean-feed 770
sxml:content 767
sxml:content-raw 767
sxml:descendant 765
sxml:descendant-or-self 765
sxml:element-name 766
sxml:element? 758
sxml:empty-element? 766
sxml:equal? 764
sxml:equality-cmp 764
sxml:filter 759
sxml:following 765
sxml:following-sibling 765
sxml:id 764
sxml:id-alist 763
sxml:invert 758
sxml:lookup 770
sxml:minimized? 766
sxml:name 766
sxml:name->ns-id 766
sxml:namespace 765
sxml:ncname 766
sxml:node-name 766
sxml:node-parent 770
sxml:node? 764
sxml:non-terminated-html-tag? 770
sxml:normalized? 766
sxml:not-equal? 764
sxml:ns-id 768
sxml:ns-id->nodes 768
sxml:ns-id->uri 768
sxml:ns-list 768
sxml:ns-prefix 768
sxml:ns-uri 768
sxml:ns-uri->id 768
sxml:num-attr 768
sxml:number 764
sxml:parent 765
sxml:preceding 765
sxml:preceding-sibling 765
sxml:relational-cmp 764
sxml:set-attr 769
sxml:set-attr! 769
sxml:shallow-minimized? 766
sxml:shallow-normalized? 766
sxml:squeeze 769
sxml:squeeze! 769
sxml:string 764
sxml:string->html 770
sxml:string->xml 770
sxml:string-value 764
sxml:sxml->html 770
sxml:sxml->xml 770
sxpath 761
symbol->string 138
symbol-append 139
symbol-hash 102
symbol-interned? 138
symbol-sans-prefix 139
symbol=? 138
symbol? 138
syntax-error 95
syntax-errorf 95
syntax-rules 84
sys-abort 247
sys-access 256
sys-alarm 275
sys-alloc-console 709
sys-asctime 270
sys-available-processors 250
sys-basename 254
sys-cfgetispeed 451
sys-cfgetospeed 451

sys-cfsetispeed 451
sys-cfsetospeed 451
sys-chdir 257
sys-chmod 257
sys-chown 257
sys-clearenv 249
sys-closelog 449
sys-create-console-screen-buffer 709
sys-crypt 259
sys-ctermid 267
sys-ctime 270
sys-difftime 270
sys-dirname 254
sys-environ 248
sys-environ->alist 249
sys-errno->symbol 269
sys-exec 271
sys-exit 247
sys-fchmod 257
sys-fcntl 370
sys-fdset 274
sys-fdset->list 274
sys-fdset-clear! 274
sys-fdset-copy! 274
sys-fdset-max-fd 275
sys-fdset-ref 274
sys-fdset-set! 274
sys-fill-console-output-attribute 712
sys-fill-console-output-character 711
sys-flush-console-input-buffer 712
sys-fork 271
sys-fork-and-exec 272
sys-forkpty 451
sys-forkpty-and-exec 451
sys-free-console 709
sys-fstat 255
sys-ftruncate 257
sys-generate-console-ctrl-event 709
sys-get-console-cp 709
sys-get-console-cursor-info 709
sys-get-console-mode 709
sys-get-console-output-cp 709
sys-get-console-screen-buffer-info 710
sys-get-console-title 712
sys-get-largest-console-window-size 710
sys-get-number-of-console-input-events 711
sys-get-number-of-console-mouse-buttons 711
sys-get-osfhandle 277
sys-get-std-handle 712
sys-getaddrinfo 410
sys-getcwd 266
sys-getdomainname 266
sys-getegid 266
sys-getenv 248
sys-geteuid 266
sys-getgid 266
sys-getgrgid 258
sys-getgrnam 258
sys-getgroups 267
sys-gethostbyaddr 409
sys-gethostbyname 408
sys-gethostname 266
sys-getlogin 267
sys-getpgid 267
sys-getpgrp 267
sys-getpid 267
sys-getppid 267
sys-getprotobyname 410
sys-getprotobynumber 410
sys-getpwnam 259
sys-getpwuid 259
sys-getrlimit 267
sys-getservbyname 409
sys-getservbyport 409
sys-gettimeofday 269
sys-getuid 266
sys-gid->group-name 258
sys-glob 250
sys-gmtime 270
sys-group-name->gid 258
sys-htonl 410
sys-htons 410
sys-isatty 257
sys-kill 262
sys-link 253
sys-localeconv 259
sys-localtime 270
sys-logmask 449
sys-lstat 256
sys-message-box 708
sys-mkdir 253
sys-mkdtemp 253
sys-mkfifo 257
sys-mkstemp 252
sys-mktime 270
sys-nanosleep 276
sys-normalize-pathname 253
sys-ntohl 410
sys-ntohs 410
sys-openlog 449
sys-openpty 451
sys-pause 275
sys-peek-console-input 711
sys-pipe 257
sys-putenv 249
sys-random 276
sys-read-console 711
sys-read-console-input 711
sys-read-console-output 711
sys-read-console-output-attribute 711
sys-read-console-output-character 711
sys-readdir 250
sys-readlink 253
sys-realpath 254
sys-remove 252
sys-rename 252
sys-rmdir 253
sys-scroll-console-screen-buffer 710
sys-select 275
sys-select! 275
sys-set-console-active-screen-buffer 710
sys-set-console-cp 709
sys-set-console-cursor-info 709
sys-set-console-cursor-position 709
sys-set-console-mode 709
sys-set-console-output-cp 709
sys-set-console-text-attribute 711
sys-set-console-title 712
sys-set-console-window-info 711
sys-set-screen-buffer-size 710

sys-set-std-handle ... 712
sys-setenv ... 249
sys-setgid ... 266
sys-setgroups ... 267
sys-setlocale ... 259
sys-setlogmask ... 449
sys-setpgid ... 267
sys-setrlimit ... 267
sys-setsid ... 267
sys-setuid ... 266
sys-sigmask ... 265
sys-signal-name ... 261
sys-sigset ... 261
sys-sigset-add! ... 261
sys-sigset-delete! ... 261
sys-sigset-empty! ... 261
sys-sigset-fill! ... 261
sys-sigsuspend ... 265
sys-sigwait ... 265
sys-sleep ... 275
sys-srandom ... 276
sys-stat ... 255
sys-stat->atime ... 256
sys-stat->ctime ... 256
sys-stat->dev ... 256
sys-stat->file-type ... 256
sys-stat->gid ... 256
sys-stat->ino ... 256
sys-stat->mode ... 256
sys-stat->mtime ... 256
sys-stat->nlink ... 256
sys-stat->rdev ... 256
sys-stat->size ... 256
sys-stat->uid ... 256
sys-strerror ... 268
sys-strftime ... 270
sys-symbol->errno ... 269
sys-symlink ... 253
sys-syslog ... 449
sys-system ... 271
sys-tcdrain ... 450
sys-tcflow ... 451
sys-tcflush ... 450
sys-tcgetattr ... 450
sys-tcgetpgrp ... 451
sys-tcsendbreak ... 450
sys-tcsetattr ... 450
sys-tcsetpgrp ... 451
sys-time ... 269
sys-times ... 267
sys-tm->alist ... 270
sys-tmpdir ... 254
sys-tmpnam ... 252
sys-truncate ... 257
sys-ttyname ... 257
sys-uid->user-name ... 259
sys-umask ... 253
sys-uname ... 266
sys-unlink ... 253
sys-unsetenv ... 249
sys-user-name->uid ... 259
sys-utime ... 257
sys-wait ... 273
sys-wait-exit-status ... 273
sys-wait-exited? ... 273
sys-wait-signaled? ... 273
sys-wait-stopped? ... 273
sys-wait-stopsig ... 273
sys-wait-termsig ... 273
sys-waitpid ... 273
sys-win-process-pid ... 274
sys-win-process? ... 274
sys-write-console ... 711
sys-write-console-output-character ... 711

T

tabulate-array ... 319
take ... 128
take! ... 129
take* ... 129
take-after ... 759
take-right ... 129
take-right* ... 129
take-until ... 759
take-while ... 515
take-while! ... 515
tan ... 117
tanh ... 117
temporary-directory ... 701
tenth ... 513
terminate-all! ... 647
terminated-thread-exception? ... 468
test ... 454
test* ... 454
test-check ... 455
test-end ... 453
test-error ... 455
test-log ... 453
test-module ... 456
test-none-of ... 455
test-one-of ... 455
test-record-file ... 453
test-script ... 457
test-section ... 453
test-start ... 453
test-summary-check ... 454
textdomain ... 783
textual-port? ... 506
third ... 513
thread-cont! ... 461
thread-join! ... 462
thread-name ... 461
thread-pool-results ... 647
thread-pool-shut-down? ... 647
thread-sleep! ... 461
thread-specific ... 461
thread-specific-set! ... 461
thread-start! ... 461
thread-state ... 461
thread-stop! ... 461
thread-terminate! ... 462
thread-yield! ... 461
thread? ... 460
time ... 468
time->seconds ... 271
time-counter-reset! ... 471
time-counter-start! ... 470
time-counter-stop! ... 470
time-counter-value ... 471

time-difference ... 602
time-difference! ... 602
time-monotonic->date ... 603
time-monotonic->julian-day ... 603
time-monotonic->modified-julian-day ... 603
time-monotonic->time-tai ... 603
time-monotonic->time-tai! ... 603
time-monotonic->time-utc ... 603
time-monotonic->time-utc! ... 603
time-nanosecond ... 601
time-resolution ... 601
time-result+ ... 469
time-result- ... 469
time-result-count ... 469
time-result-real ... 469
time-result-sys ... 469
time-result-user ... 469
time-result? ... 469
time-second ... 601
time-tai->date ... 603
time-tai->julian-day ... 603
time-tai->modified-julian-day ... 603
time-tai->time-monotonic ... 603
time-tai->time-monotonic! ... 603
time-tai->time-utc ... 603
time-tai->time-utc! ... 603
time-these ... 469
time-these/report ... 469
time-this ... 468
time-type ... 601
time-utc->date ... 603
time-utc->julian-day ... 603
time-utc->modified-julian-day ... 603
time-utc->time-monotonic ... 604
time-utc->time-monotonic! ... 604
time-utc->time-tai ... 604
time-utc->time-tai! ... 604
time<=? ... 601
time<? ... 601
time=? ... 601
time>=? ... 601
time>? ... 601
time? ... 271
tls-ca-bundle-path ... 738
topological-sort ... 818
totient ... 708
touch-file ... 699
touch-files ... 699
tr ... 792
trace-macro ... 93
tree->string ... 794
tree-map->alist ... 185
tree-map->imap ... 655
tree-map-ceiling ... 184
tree-map-ceiling-key ... 185
tree-map-ceiling-value ... 185
tree-map-clear! ... 183
tree-map-comparator ... 183
tree-map-compare-as-sequences ... 186
tree-map-compare-as-sets ... 185
tree-map-copy ... 183
tree-map-delete! ... 183
tree-map-empty? ... 183
tree-map-exists? ... 183
tree-map-floor ... 184
tree-map-floor-key ... 185
tree-map-floor-value ... 185
tree-map-fold ... 184
tree-map-fold-right ... 184
tree-map-for-each ... 184
tree-map-get ... 183
tree-map-keys ... 185
tree-map-map ... 184
tree-map-max ... 184
tree-map-min ... 184
tree-map-num-entries ... 183
tree-map-pop! ... 183
tree-map-pop-max! ... 184
tree-map-pop-min! ... 184
tree-map-predecessor ... 184
tree-map-predecessor-key ... 185
tree-map-predecessor-value ... 185
tree-map-push! ... 183
tree-map-put! ... 183
tree-map-successor ... 184
tree-map-successor-key ... 185
tree-map-successor-value ... 185
tree-map-update! ... 183
tree-map-values ... 185
trie ... 674
trie->hash-table ... 675
trie->list ... 675
trie-common-prefix ... 675
trie-common-prefix-fold ... 676
trie-common-prefix-for-each ... 676
trie-common-prefix-keys ... 675
trie-common-prefix-map ... 676
trie-common-prefix-values ... 675
trie-delete! ... 675
trie-exists? ... 674
trie-fold ... 676
trie-for-each ... 676
trie-get ... 675
trie-keys ... 675
trie-longest-match ... 675
trie-map ... 676
trie-num-entries ... 674
trie-partial-key? ... 674
trie-put! ... 675
trie-update! ... 675
trie-values ... 675
trie-with-keys ... 674
trie? ... 674
truncate ... 116
truncate->exact ... 117
truncate-quotient ... 115
truncate-remainder ... 115
truncate/ ... 115
tuples-of ... 663
twos-exponent ... 123
twos-exponent-factor ... 123

U

u16? . . . 477
u16array . . . 317
u16vector . . . 478
u16vector->list . . . 484
u16vector->vector . . . 484
u16vector-add . . . 487
u16vector-add! . . . 487
u16vector-and . . . 487
u16vector-and! . . . 487
u16vector-append . . . 482
u16vector-append-subvectors . . . 482
u16vector-clamp . . . 488
u16vector-clamp! . . . 488
u16vector-compare . . . 480
u16vector-concatenate . . . 482
u16vector-copy . . . 480
u16vector-copy! . . . 480
u16vector-dot . . . 487
u16vector-empty? . . . 478
u16vector-fill! . . . 480
u16vector-ior . . . 487
u16vector-ior! . . . 487
u16vector-length . . . 478
u16vector-mul . . . 487
u16vector-mul! . . . 487
u16vector-multi-copy! . . . 481
u16vector-range-check . . . 487
u16vector-ref . . . 479
u16vector-reverse-copy . . . 480
u16vector-set! . . . 479
u16vector-sub . . . 487
u16vector-sub! . . . 487
u16vector-swap! . . . 480
u16vector-unfold . . . 478
u16vector-unfold-right . . . 478
u16vector-xor . . . 487
u16vector-xor! . . . 487
u16vector= . . . 480
u16vector=? . . . 480
u16vector? . . . 477
u32? . . . 477
u32array . . . 317
u32vector . . . 478
u32vector->list . . . 484
u32vector->string . . . 486
u32vector->vector . . . 484
u32vector-add . . . 487
u32vector-add! . . . 487
u32vector-and . . . 487
u32vector-and! . . . 487
u32vector-append . . . 482
u32vector-append-subvectors . . . 482
u32vector-clamp . . . 488
u32vector-clamp! . . . 488
u32vector-compare . . . 480
u32vector-concatenate . . . 482
u32vector-copy . . . 480
u32vector-copy! . . . 480
u32vector-dot . . . 487
u32vector-empty? . . . 478
u32vector-fill! . . . 480
u32vector-ior . . . 487
u32vector-ior! . . . 487
u32vector-length . . . 478
u32vector-mul . . . 487
u32vector-mul! . . . 487
u32vector-multi-copy! . . . 481
u32vector-range-check . . . 487
u32vector-ref . . . 479
u32vector-reverse-copy . . . 480
u32vector-set! . . . 479
u32vector-sub . . . 487
u32vector-sub! . . . 487
u32vector-swap! . . . 480
u32vector-unfold . . . 478
u32vector-unfold-right . . . 478
u32vector-xor . . . 487
u32vector-xor! . . . 487
u32vector= . . . 480
u32vector=? . . . 480
u32vector? . . . 477
u64? . . . 477
u64array . . . 317
u64vector . . . 478
u64vector->list . . . 484
u64vector->vector . . . 484
u64vector-add . . . 487
u64vector-add! . . . 487
u64vector-and . . . 487
u64vector-and! . . . 487
u64vector-append . . . 482
u64vector-append-subvectors . . . 482
u64vector-clamp . . . 488
u64vector-clamp! . . . 488
u64vector-compare . . . 480
u64vector-concatenate . . . 482
u64vector-copy . . . 480
u64vector-copy! . . . 480
u64vector-dot . . . 487
u64vector-empty? . . . 478
u64vector-fill! . . . 480
u64vector-ior . . . 487
u64vector-ior! . . . 487
u64vector-length . . . 478
u64vector-mul . . . 487
u64vector-mul! . . . 487
u64vector-multi-copy! . . . 481
u64vector-range-check . . . 487
u64vector-ref . . . 479
u64vector-reverse-copy . . . 480
u64vector-set! . . . 479
u64vector-sub . . . 487
u64vector-sub! . . . 487
u64vector-swap! . . . 480
u64vector-unfold . . . 478
u64vector-unfold-right . . . 478
u64vector-xor . . . 487
u64vector-xor! . . . 487
u64vector= . . . 480
u64vector=? . . . 480
u64vector? . . . 477
u8-list->blob . . . 622
u8-ready? . . . 506
u8? . . . 477
u8array . . . 317
u8vector . . . 478
u8vector->list . . . 484
u8vector->string . . . 485

u8vector->vector 484
u8vector-add 487
u8vector-add! 487
u8vector-and 487
u8vector-and! 487
u8vector-append 482
u8vector-append-subvectors 482
u8vector-clamp 488
u8vector-clamp! 488
u8vector-compare 480
u8vector-concatenate 482
u8vector-copy 480
u8vector-copy! 480, 618
u8vector-dot 487
u8vector-empty? 478
u8vector-fill! 480
u8vector-ior 487
u8vector-ior! 487
u8vector-length 478
u8vector-mul 487
u8vector-mul! 487
u8vector-multi-copy! 481
u8vector-range-check 487
u8vector-ref 479
u8vector-reverse-copy 480
u8vector-set! 479
u8vector-sub 487
u8vector-sub! 487
u8vector-swap! 480
u8vector-unfold 478
u8vector-unfold-right 478
u8vector-xor 487
u8vector-xor! 487
u8vector= 480
u8vector=? 480
u8vector? 477
ucs->char 146
ucs-range->char-set 533
ucs-range->char-set! 533
ucs4->utf16 473
ucs4->utf8 472
uint-list->blob 622
uint16s 661
uint32s 661
uint64s 661
uint8s 661
unbox 547
uncaught-exception-reason 468
uncaught-exception? 468
undefined 125
undefined? 125
unfold 514
unfold-right 515
unify 819
unify-merge 819
unless 51, 332
unpack 643
unpack-skip 643
unquote 59
unquote-splicing 59
until 58
untrace-macro 93
unwind-protect 209
unwrap-syntax 95, 143
unzip1 514
unzip2 514
unzip3 514
unzip4 514
unzip5 514
update! 49
uri-compose 741
uri-compose-data 742
uri-decode 742
uri-decode-string 742
uri-decompose-authority 739
uri-decompose-data 740
uri-decompose-hierarchical 739
uri-encode 742
uri-encode-string 742
uri-merge 741
uri-parse 739
uri-ref 738
uri-scheme&specific 739
use 75
utf16->ucs4 473
utf16-length 473
utf8->string 473
utf8->ucs4 473
utf8-length 472
uvector->generator 373
uvector->list 484
uvector->vector 484
uvector-alias 486
uvector-binary-search 482
uvector-class-element-size 479
uvector-copy 480
uvector-copy! 482
uvector-length 478
uvector-ref 479
uvector-set! 479
uvector-size 478
uvector? 477

V

valid-sre? 560
valid-version-spec? 492
values 197
values->list 197
values-ref 197
vector 174
vector->@vector 484
vector->bits 578
vector->c128vector 484
vector->c32vector 484
vector->c64vector 484
vector->f16vector 484
vector->f32vector 484
vector->f64vector 484
vector->generator 373
vector->list 174
vector->s16vector 484
vector->s32vector 484
vector->s64vector 484
vector->s8vector 484
vector->string 175
vector->u16vector 484
vector->u32vector 484
vector->u64vector 484
vector->u8vector 484

vector-accumulator ... 543
vector-accumulator! ... 544
vector-any ... 520
vector-append ... 176
vector-append-subvectors ... 518
vector-binary-search ... 520
vector-concatenate ... 518
vector-copy ... 175
vector-copy! ... 175
vector-count ... 519, 615
vector-cumulate ... 519
vector-delete-neighbor-dups ... 523
vector-delete-neighbor-dups! ... 523
vector-ec ... 611
vector-empty? ... 518
vector-every ... 520
vector-fill! ... 175
vector-find-median ... 524
vector-find-median! ... 524
vector-fold ... 519, 615
vector-fold-right ... 519, 615
vector-for-each ... 176, 615
vector-for-each-with-index ... 176
vector-index ... 519
vector-index-right ... 519
vector-length ... 174
vector-map ... 176, 615
vector-map! ... 176, 615
vector-map-with-index ... 176
vector-map-with-index! ... 176
vector-merge ... 523
vector-merge! ... 523
vector-of-length-ec ... 611
vector-partition ... 520
vector-ref ... 174
vector-reverse! ... 521
vector-reverse-copy ... 518
vector-reverse-copy! ... 521
vector-select! ... 524
vector-separate! ... 524
vector-set! ... 174
vector-skip ... 520
vector-skip-right ... 520
vector-sort ... 523
vector-sort! ... 523
vector-sorted? ... 523
vector-stable-sort ... 523
vector-stable-sort! ... 523
vector-swap! ... 520
vector-tabulate ... 174
vector-unfold ... 517
vector-unfold! ... 521
vector-unfold-right ... 518
vector-unfold-right! ... 521
vector= ... 518
vector? ... 174
vectors-of ... 664
version-compare ... 492
version-satisfy? ... 492
version<=? ... 492
version<? ... 492
version=? ... 492
version>=? ... 492
version>? ... 492
vt100-compatible? ... 774

W

wait-all ... 647
weak-vector-length ... 187
weak-vector-ref ... 187
weak-vector-set! ... 187
weighted-samples-from ... 663
when ... 51, 332, 339
while ... 58, 333
with-builder ... 350
with-cf-subst ... 358
with-character-attribute ... 776
with-dynamic-extent ... 636
with-error-handler ... 210
with-error-to-port ... 219
with-exception-handler ... 211, 505
with-input-conversion ... 343
with-input-from-file ... 223
with-input-from-port ... 219
with-input-from-process ... 431
with-input-from-string ... 225
with-iterator ... 350
with-lock-file ... 702
with-locking-mutex ... 464
with-module ... 73
with-output-conversion ... 343
with-output-to-file ... 223
with-output-to-port ... 219
with-output-to-process ... 431
with-output-to-string ... 225
with-port-locking ... 218
with-ports ... 219
with-profiler ... 278
with-random-data-seed ... 660
with-signal-handlers ... 264
with-string-io ... 226
with-time-counter ... 471
without-echoing ... 452
wrap-with-input-conversion ... 343
wrap-with-output-conversion ... 343
write ... 233
write* ... 234
write-ber-integer ... 639
write-block ... 489
write-byte ... 239
write-bytevector ... 507
write-char ... 239
write-controls-copy ... 233
write-f16 ... 639
write-f32 ... 639
write-f64 ... 639
write-gauche-package-description ... 411
write-object ... 235
write-s16 ... 639
write-s32 ... 639
write-s64 ... 639
write-s8 ... 639
write-shared ... 233
write-simple ... 233
write-sint ... 639
write-stream ... 818

write-string 239
write-to-string 226
write-tree 793
write-u16 638
write-u32 638
write-u64 638
write-u8 .... 507, 638
write-uint 639
write-uvector 489
write-with-shared-structure 234
write/ss 234

X

x->generator 375
x->integer 121
x->lseq 387
x->number 121
x->string 155
xcons 513
xml-token-head 749
xml-token-kind 749
xml-token? 749
xsubstring 598

Z

zero? 110
zip 514
zlib-version 746
zstream-adler32 745
zstream-data-type 745
zstream-dictionary-adler32 746
zstream-params-set! 745
zstream-total-in 745
zstream-total-out 745

Appendix D Module Index

B

binary.io ... 638
binary.pack ... 641

C

compat.chibi-test ... 644
compat.norational ... 644
control.job ... 645
control.thread-pool ... 646
crypt.bcrypt ... 648

D

data.cache ... 649
data.heap ... 652
data.ideque ... 654
data.imap ... 654
data.queue ... 656
data.random ... 660
data.ring-buffer ... 665
data.sparse ... 666
data.trie ... 672
dbi ... 676
dbm ... 682
dbm.fsdbm ... 687
dbm.gdbm ... 688
dbm.ndbm ... 690
dbm.odbm ... 690

F

file.filter ... 691
file.util ... 692

G

gauche ... 78
gauche.array ... 315
gauche.base ... 322
gauche.cgen ... 323
gauche.charconv ... 339
gauche.collection ... 344
gauche.config ... 352
gauche.configure ... 352
gauche.connection ... 365
gauche.dictionary ... 366
gauche.fcntl ... 370
gauche.generator ... 372
gauche.hook ... 383
gauche.interactive ... 384
gauche.keyword ... 78
gauche.lazy ... 386
gauche.listener ... 389
gauche.logger ... 393
gauche.mop.propagate ... 395
gauche.mop.singleton ... 396
gauche.mop.validator ... 397
gauche.net ... 398
gauche.package ... 411
gauche.parameter ... 411
gauche.parseopt ... 413
gauche.partcont ... 417
gauche.process ... 421
gauche.record ... 433
gauche.reload ... 439
gauche.selector ... 440
gauche.sequence ... 441
gauche.syslog ... 448
gauche.termios ... 449
gauche.test ... 452
gauche.threads ... 457
gauche.time ... 468
gauche.unicode ... 471
gauche.uvector ... 476
gauche.version ... 491
gauche.vport ... 493

K

keyword ... 78

M

math.const ... 704
math.mt-random ... 704
math.prime ... 705

N

null ... 78

O

os.windows ... 708

R

rfc.822 ... 712
rfc.base64 ... 716
rfc.cookie ... 717
rfc.ftp ... 718
rfc.hmac ... 721
rfc.http ... 721
rfc.icmp ... 727
rfc.ip ... 727
rfc.json ... 728
rfc.md5 ... 730
rfc.mime ... 730
rfc.quoted-printable ... 736
rfc.sha ... 736
rfc.sha1 ... 736
rfc.tls ... 737
rfc.uri ... 738
rfc.zlib ... 743

S

scheme ... 78
scheme.base ... 504
scheme.bitwise ... 575
scheme.box ... 547
scheme.case-lambda ... 507
scheme.char ... 507
scheme.charset ... 533
scheme.comparator ... 551
scheme.complex ... 508
scheme.cxr ... 508
scheme.division ... 574
scheme.ephemeron ... 550
scheme.eval ... 508
scheme.file ... 509
scheme.finxum ... 579
scheme.flonum ... 581
scheme.generator ... 542
scheme.hash-table ... 536
scheme.ideque ... 540
scheme.inexact ... 509
scheme.lazy ... 509
scheme.list ... 512
scheme.list-queue ... 547
scheme.load ... 510
scheme.lseq ... 544
scheme.mapping ... 563
scheme.mapping.hash ... 563
scheme.process-context ... 510
scheme.r5rs ... 512
scheme.read ... 510
scheme.regex ... 552
scheme.repl ... 510
scheme.set ... 525
scheme.sort ... 522
scheme.stream ... 546
scheme.time ... 511
scheme.vector ... 517
scheme.vector.base ... 522
scheme.vector.c128 ... 522
scheme.vector.c64 ... 522
scheme.vector.f32 ... 522
scheme.vector.f64 ... 522
scheme.vector.s16 ... 522
scheme.vector.s32 ... 522
scheme.vector.s64 ... 522
scheme.vector.s8 ... 522
scheme.vector.u16 ... 522
scheme.vector.u32 ... 522
scheme.vector.u64 ... 522
scheme.vector.u8 ... 522
scheme.write ... 511
slib ... 746
srfi-1 ... 590
srfi-106 ... 623
srfi-111 ... 625
srfi-112 ... 625
srfi-113 ... 626
srfi-114 ... 626
srfi-115 ... 631
srfi-117 ... 631
srfi-118 ... 631
srfi-127 ... 632
srfi-13 ... 591
srfi-132 ... 314, 632
srfi-133 ... 632
srfi-14 ... 600
srfi-141 ... 632
srfi-143 ... 632
srfi-146 ... 632
srfi-146.hash ... 632
srfi-151 ... 633
srfi-152 ... 633
srfi-154 ... 635
srfi-158 ... 636
srfi-160 ... 636
srfi-162 ... 636
srfi-173 ... 637
srfi-19 ... 600
srfi-27 ... 605
srfi-29 ... 606
srfi-29.bundle ... 606
srfi-29.format ... 606
srfi-37 ... 607
srfi-4 ... 590
srfi-41 ... 609
srfi-42 ... 610
srfi-43 ... 615
srfi-5 ... 590
srfi-55 ... 616
srfi-60 ... 616
srfi-66 ... 618
srfi-69 ... 618
srfi-7 ... 590
srfi-74 ... 620
srfi-98 ... 622
sxml.serializer ... 771
sxml.ssax ... 747
sxml.sxpath ... 757
sxml.tools ... 766

T

text.console ... 773
text.csv ... 776
text.diff ... 779
text.edn ... 780
text.gettext ... 783
text.html-lite ... 784
text.parse ... 786
text.progress ... 788
text.sql ... 790
text.template ... 790
text.tr ... 791
text.tree ... 793
text.unicode ... 313

U

user 78
util.combinations 794
util.digest 795
util.dominator 796
util.isomorph 797
util.lcs 798
util.levenshtein 799
util.list 313
util.match 801
util.queue 313
util.rbtree 313
util.record 806
util.relation 807
util.sparse 313
util.stream 809
util.toposort 818
util.trie 313
util.unification 818

W

www.cgi 819
www.cgi.test 824

Appendix E Lexical syntax index

#

#! 41
#" 155
#* 154
#, ... 229
#/ 162
#[.... 148
#` 156
#\ 143
#c128 477
#c32 477
#c64 477
#f16 477
#f32 477
#f64 477
#s16 477
#s32 477
#s64 477
#s8 477
#u16 477
#u32 477
#u64 477
#u8 477

'

' 41

,

, ... 59
,@ 59

[

[.... 38

`

` 59

\

\x 39

|

| 137

Appendix F Class Index

For readability, the surrounding < and > are stripped off.

A

abandoned-mutex-exception 467
array 315
array-base 315
ax-tls 737

B

bimap 369
binary-heap 652
boolean 124
bottom 96
buffered-input-port 495
buffered-output-port 496

C

c128vector 477
c32vector 477
c64vector 477
cgen-node 327
cgen-type 331
cgen-unit 323
char 143
char-set 147
class 302
complex 109
compound-condition 213
condition 213
condition-meta 213
condition-variable 465

D

date 602
dbi-connection 678
dbi-driver 678
dbi-query 679
dbm 683
dbm-meta 684
deflating-port 743

E

edn-object 781
error 213

F

f16array 315
f16vector 477
f32array 315
f32vector 477
f64array 315
f64vector 477
fsdbm 687
ftp-connection 718
ftp-error 718

G

gdbm 688

H

hash-table 177
hmac 721
hook 383
http-error 721

I

identifier 142
inflating-port 743
integer 109
io-closed-error 215
io-error 214
io-read-error 214
io-unit-error 215
io-write-error 214

J

join-timeout-exception 467

K

keyword 139

L

list 125
listener 390
log-drain 393

M

mapping 563
mbed-tls 737
md5 730
mersenne-twister 704
message-condition 213
message-digest-algorithm 795
message-digest-algorithm-meta 795
mime-message 733
module 76
mtqueue 657
mutex 462

N

ndbm 690
null 125
number 109

O

object 96
object-set-relation 809
odbm 690

P

pair 125
parameter 412
port 217
port-error 214
procedure 188
process 427
process-abnormal-exit 427
process-time-counter 470
propagate-meta 396
propagate-mixin 396

Q

queue 657

R

rational 109
rbtree 313
read-error 214
real 109
real-time-counter 470
regexp 165
regmatch 165
relation 807

S

s16array 315
s16vector 477
s32array 315
s32vector 477
s64array 315
s64vector 477
s8array 315
s8vector 477
selector 440
serious-compound-condition 213
serious-condition 213
sha1 736
sha224 736
sha256 736
sha384 736
sha512 736
simple-relation 809
singleton-meta 396
singleton-mixin 397
sockaddr 398
sockaddr-in 398
sockaddr-in6 399
sockaddr-un 399
socket 400
sparse-f16matrix 670
sparse-f16vector 667
sparse-f32matrix 670
sparse-f32vector 667
sparse-f64matrix 670
sparse-f64vector 667
sparse-matrix 670
sparse-matrix-base 669
sparse-s16matrix 670
sparse-s16vector 667
sparse-s32matrix 670
sparse-s32vector 667
sparse-s64matrix 670
sparse-s64vector 667
sparse-s8matrix 670
sparse-s8vector 667
sparse-table 671
sparse-u16matrix 670
sparse-u16vector 667
sparse-u32matrix 670
sparse-u32vector 667
sparse-u64matrix 670
sparse-u64vector 667
sparse-u8matrix 670
sparse-u8vector 667
sparse-vector 667
sparse-vector-base 667
string 153
symbol 137
sys-addrinfo 410
sys-fdset 274
sys-flock 371
sys-group 258
sys-hostent 408
sys-passwd 258
sys-protoent 409
sys-servent 409
sys-sigset 261
sys-stat 255
sys-termios 450
sys-tm 269
system-error 213
system-time-counter 470

T

terminated-thread-exception 467
thread 459
thread-exception 467
thread-pool 646
time 270
time-counter 470
top 96
tree-map 182
trie 673

U

u16array 315
u16vector 477
u32array 315
u32vector 477
u64array 315
u64vector 477
u8array 315
u8vector 477
uncaught-exception 467
unhandled-signal-error 214
user-time-counter 470

V

validator-meta 397
vector 174
virtual-input-port 493
virtual-output-port 494
vt100 773

W

weak-vector 187
win:console-screen-buffer-info 710
win:input-record 710
windows-console 773
write-controls 232

Appendix G Variable Index

&

&condition 213
&error 213
&io-closed-error 215
&io-error 214
&io-port-error 214
&io-read-error 214
&io-write-error 214
&read-error 214
&serious 213

*

af-inet 624
af-inet6 624
af-unspec 624
ai-addrconfig 624
ai-all 624
ai-canonname 624
ai-numerichost 624
ai-v4mapped 624
argv 248
ipproto-ip 625
ipproto-tcp 625
ipproto-udp 625
load-path 239
msg-none 625
msg-oob 625
msg-peek 625
msg-waitall 625
primes 705
program-name 248
rfc2396-unreserved-char-set 743
rfc3986-unreserved-char-set 743
rfc822-atext-chars 714
rfc822-standard-tokenizers 714
shut-rd 625
shut-rdwr 625
shut-wr 625
small-prime-bound 706
sock-dgram 624
sock-stream 624
test-error 456
test-report-error 456

@

@vector-comparator 482

1

1/pi 704
180/pi 704

A

accessors of <class> 302
addr of <sys-addrinfo> 410
addresses of <sys-hostent> 408
addrlen of <sys-addrinfo> 410
AF_INET 404
AF_INET6 404
AF_UNIX 404
aliases of <sys-hostent> 408
aliases of <sys-servent> 409, 410
atime of <sys-stat> 255
attributes of <win:console-screen-buffer-info> 710

B

BACKGROUND_BLUE 710
BACKGROUND_GREEN 710
BACKGROUND_INTENSITY 710
BACKGROUND_RED 710
bag-comparator 533
base of <write-controls> 232
boolean-comparator 107
bsize of <gdbm> 688
bytevector-comparator 107

C

c-file of <cgen-unit> 324
c128vector-comparator 482
c32vector-comparator 482
c64vector-comparator 482
category of <class> 303
cc of <sys-termios> 450
cflag of <sys-termios> 450
char-ci-comparator 107
char-comparator 107
char-set:ascii 150
char-set:ascii-blank 151
char-set:ascii-control 151
char-set:ascii-digit 150
char-set:ascii-graphic 151
char-set:ascii-letter 150
char-set:ascii-letter+digit 150
char-set:ascii-lower-case 150
char-set:ascii-printing 151
char-set:ascii-punctuation 151
char-set:ascii-symbol 151
char-set:ascii-upper-case 150
char-set:ascii-whitespace 151
char-set:ascii-word 151
char-set:blank 150
char-set:C 152
char-set:Cc 151
char-set:Cf 151
char-set:Cn 151
char-set:Co 151
char-set:Cs 151
char-set:digit 150
char-set:empty 150

char-set:full 150
char-set:graphic 150
char-set:hex-digit 150
char-set:iso-control 150
char-set:letter 149
char-set:letter+digit 150
char-set:lower-case 150
char-set:L 152
char-set:Ll 151
char-set:Lm 151
char-set:Lo 151
char-set:Lt 151
char-set:Lu 151
char-set:M 152
char-set:Mc 151
char-set:Me 151
char-set:Mn 151
char-set:N 152
char-set:Nd 151
char-set:Nl 151
char-set:No 151
char-set:P 152
char-set:Pc 151
char-set:Pd 151
char-set:Pe 151
char-set:Pf 151
char-set:Pi 151
char-set:Po 151
char-set:printing 150
char-set:Ps 151
char-set:punctuation 150
char-set:S 152
char-set:Sc 151
char-set:Sk 151
char-set:Sm 151
char-set:So 151
char-set:symbol 150
char-set:title-case 150
char-set:upper-case 150
char-set:whitespace 150
char-set:word 150
char-set:Z 152
char-set:Zl 151
char-set:Zp 151
char-set:Zs 151
class of <sys-passwd> 258
close of <buffered-input-port> 495
close of <buffered-output-port> 496
close of <virtual-input-port> 494
close of <virtual-output-port> 495
column of <read-error> 214
complex-comparator 107
connection of <dbi-query> 679
content of <mime-part> 734
count of <time-result> 469
cpl of <class> 302
ctime of <sys-stat> 255
CTRL_BREAK_EVENT 709
CTRL_C_EVENT 709
cursor-position.x of <win:console-screen-buffer-info> 710
cursor-position.y of <win:console-screen-buffer-info> 710

D

day of <date> 602
default-comparator 106
default-random-source 605
defined-modules of <class> 303
dev of <sys-stat> 255
dir of <sys-passwd> 258
direct-methods of <class> 303
direct-slots of <class> 302
direct-subclasses of <class> 303
direct-supers of <class> 302
driver-name of <dbi-nonexistent-driver-error> 677

E

e 704
edn-comparator 781
ENABLE_ECHO_INPUT 709
ENABLE_LINE_INPUT 709
ENABLE_MOUSE_INPUT 709
ENABLE_PROCESSED_INPUT 709
ENABLE_PROCESSED_OUTPUT 709
ENABLE_WINDOW_INPUT 709
ENABLE_WRAP_AT_EOL_OUTPUT 709
environment of <listener> 390
eq-comparator 107
equal-comparator 107
eqv-comparator 107
errno of <system-error> 213
error-handler of <listener> 391
error-port of <listener> 390
evaluator of <listener> 390
event-type of <win:input-record> 711
exact-integer-comparator 107

F

f16vector-comparator 482
f32vector-comparator 482
f64vector-comparator 482
F_DUPFD 371
F_GETFD 370
F_GETFL 370
F_GETLK 371
F_GETOWN 371
F_OK 256
F_RDLCK 371
F_SETFD 370
F_SETFL 371
F_SETLK 371
F_SETLKW 371
F_SETOWN 371
F_UNLCK 372
F_WRLCK 372
family of <sys-addrinfo> 410
fatal-handler of <listener> 391
FD_CLOEXEC 370
file-mode of <dbm> 684
FILE_SHARE_READ 710
FILE_SHARE_WRITE 710
filenum of <buffered-input-port> 495
filenum of <buffered-output-port> 496
fill of <buffered-input-port> 495

finalizer of <listener> 390
fl-1/e 581
fl-1/log-10 582
fl-1/log-2 581
fl-1/log-phi 583
fl-1/pi 582
fl-1/sqrt-2 582
fl-2/pi 582
fl-2/sqrt-pi 582
fl-2pi 582
fl-4thrt-2 583
fl-cbrt-2 582
fl-cbrt-3 582
fl-cos-1 583
fl-degree 582
fl-e 581
fl-e-2 581
fl-e-euler 583
fl-e-pi/4 581
fl-epsilon 583
fl-euler 583
fl-fast-fl+* 583
fl-gamma-1/2 583
fl-gamma-1/3 583
fl-gamma-2/3 583
fl-greatest 583
fl-integer-exponent-nan 583
fl-integer-exponent-zero 583
fl-least 583
fl-log-10 582
fl-log-2 581
fl-log-3 582
fl-log-phi 583
fl-log-pi 582
fl-log10-e 581
fl-log2-e 581
fl-phi 583
fl-pi 582
fl-pi-squared 582
fl-pi/2 582
fl-pi/4 582
fl-sin-1 583
fl-sqrt-10 582
fl-sqrt-2 582
fl-sqrt-3 582
fl-sqrt-5 582
flags of <sys-addrinfo> 410
flush of <buffered-output-port> 496
flush of <virtual-output-port> 495
focus.set-focus of <win:input-record> 711
FOREGROUND_BLUE 710
FOREGROUND_GREEN 710
FOREGROUND_INTENSITY 710
FOREGROUND_RED 710
fx-greatest 579
fx-least 579
fx-width 579

G

GDBM_CACHESIZE 689
GDBM_CENTFREE 689
GDBM_COALESCEBLKS 689
GDBM_FAST 688
GDBM_FASTMODE 689
GDBM_INSERT 689
GDBM_NEWDB 688
GDBM_NOLOCK 688
GDBM_READER 688
GDBM_REPLACE 689
GDBM_SYNC 688
GDBM_SYNCMODE 689
GDBM_WRCREAT 688
GDBM_WRITER 688
gecos of <sys-passwd> 258
GENERIC_READ 710
GENERIC_WRITE 710
getb of <virtual-input-port> 493
getc of <virtual-input-port> 493
gets of <virtual-input-port> 493
gid of <sys-group> 258
gid of <sys-passwd> 258
gid of <sys-stat> 255

H

h-file of <cgen-unit> 324
hashmap-comparator 574
headers of <mime-part> 734
hmac-block-size of <message-digest-algorithm-meta> 795
hour of <date> 602
hour of <sys-tm> 269

I

iflag of <sys-termios> 450
index of <mime-part> 734
init-epilogue of <cgen-init> 324
init-prologue of <cgen-unit> 324
initargs of <class> 303
ino of <sys-stat> 255
input-delay of <vt100> 773
input-port of <listener> 390
integer-comparator 107
iport of <vt100> 773
isdst of <sys-tm> 270

K

key-convert of <dbm> 684
key.ascii-char of <win:input-record> 711
key.control-key-state of <win:input-record> 711
key.down of <win:input-record> 711
key.repeat-count of <win:input-record> 711
key.unicode-char of <win:input-record> 711
key.virtual-key-code of <win:input-record> 711
key.virtual-scan-code of <win:input-record> 711

L

LC_ALL ... 259
LC_COLLATE ... 259
LC_CTYPE ... 259
LC_MONETARY ... 259
LC_NUMERIC ... 259
LC_TIME ... 259
len of <sys-flock> ... 372
length of <queue> ... 657
length of <write-controls> ... 232
level of <write-controls> ... 232
lflag of <sys-termios> ... 450
line of <read-error> ... 214
list-comparator ... 107
lock-file-name of <lock-file-failure> ... 703
lock-policy of <log-drain> ... 394
log-drain of <ftp-connection> ... 718

M

mapping-comparator ... 570
max-length of <mtqueue> ... 657
maximum-window-size.x of <win:console-screen-buffer-info> ... 710
maximum-window-size.y of <win:console-screen-buffer-info> ... 710
mday of <sys-tm> ... 269
mem of <sys-group> ... 258
menu.command-id of <win:input-record> ... 711
message of <message-condition> ... 213
min of <sys-tm> ... 269
minute of <date> ... 602
mode of <sys-stat> ... 255
mon of <sys-tm> ... 269
month of <date> ... 602
mouse.button-state of <win:input-record> ... 711
mouse.control-key-state of <win:input-record> ... 711
mouse.event-flags of <win:input-record> ... 711
mouse.x of <win:input-record> ... 711
mouse.y of <win:input-record> ... 711
MSG_CTRUNC ... 407
MSG_DONTROUTE ... 407
MSG_EOR ... 407
MSG_OOB ... 407
MSG_PEEK ... 407
MSG_TRUNC ... 407
MSG_WAITALL ... 407
mtime of <sys-stat> ... 255
mutex of <abandoned-mutex-exception> ... 467

N

name of <cgen-unit> ... 323
name of <class> ... 302
name of <condition-variable> ... 465
name of <mutex> ... 463
name of <sys-group> ... 258
name of <sys-hostent> ... 408
name of <sys-passwd> ... 258
name of <sys-servent> ... 409
name of <thread> ... 460
nanosecond of <date> ... 602
nanosecond of <time> ... 270
nlink of <sys-stat> ... 255
nolock of <gdbm> ... 688
num-instance-slots of <class> ... 303
number-comparator ... 107

O

O_ACCMODE ... 370
O_APPEND ... 371
O_CREAT ... 371
O_EXCL ... 371
O_NOCTTY ... 371
O_NONBLOCK ... 371
O_RDONLY ... 370
O_RDWR ... 370
O_TRUNC ... 371
O_WRONLY ... 370
object of <json-construct-error> ... 729
oflag of <sys-termios> ... 450
oport of <vt100> ... 773
output-port of <listener> ... 390

P

pair-comparator ... 107
parameters of <mime-part> ... 734
parent of <mime-part> ... 734
passive of <ftp-connection> ... 718
passwd of <sys-group> ... 258
passwd of <sys-passwd> ... 258
path of <dbm> ... 683
path of <log-drain> ... 393
payload of <edn-object> ... 781
perm of <sys-stat> ... 255
PF_INET ... 404
PF_INET6 ... 404
PF_UNIX ... 404
pi ... 704
pi/180 ... 704
pi/2 ... 704
pi/4 ... 704
pid of <sys-flock> ... 372
pool of <thread-pool-shut-down> ... 646
port of <port-error> ... 214
port of <read-error> ... 214
port of <sys-servent> ... 409
position of <json-parse-error> ... 728
position of <read-error> ... 214
preamble of <cgen-unit> ... 324
prefix of <log-drain> ... 393
prepared of <dbi-query> ... 679
pretty of <write-controls> ... 232
printer of <listener> ... 390
process of <process-abnormal-exit> ... 427
program-name of <log-drain> ... 394
prompter of <listener> ... 390
proto of <sys-servent> ... 409, 410
protocol of <sys-addrinfo> ... 410
putb of <virtual-output-port> ... 494
putc of <virtual-output-port> ... 494
puts of <virtual-output-port> ... 494

R

R_OK ... 256
radix of <write-controls> ... 232
RAND_MAX ... 276
rational-comparator ... 107
rdev of <sys-stat> ... 255
reader of <listener> ... 390
ready of <buffered-input-port> ... 495
ready of <virtual-input-port> ... 494
real of <time-result> ... 469
real-comparator ... 107
reason of <uncaught-exception> ... 467
redefined of <class> ... 303
rw-mode of <dbm> ... 684

S

s16vector-comparator ... 482
s32vector-comparator ... 482
s64vector-comparator ... 482
s8vector-comparator ... 482
sec of <sys-tm> ... 269
second of <date> ... 602
second of <time> ... 270
seek of <buffered-input-port> ... 496
seek of <buffered-output-port> ... 496
seek of <virtual-input-port> ... 494
seek of <virtual-output-port> ... 495
set-comparator ... 533
shell of <sys-passwd> ... 258
SIGABRT ... 260
SIGALRM ... 260
SIGBUS ... 261
SIGCHLD ... 260
SIGCONT ... 260
SIGFPE ... 260
SIGHUP ... 260
SIGILL ... 260
SIGINT ... 260
SIGIO ... 261
SIGIOT ... 261
SIGKILL ... 260
signal of <unhandled-signal-error> ... 214
SIGPIPE ... 260
SIGPOLL ... 261
SIGPROF ... 261
SIGPWR ... 261
SIGQUIT ... 260
SIGSEGV ... 260
SIGSTKFLT ... 261
SIGSTOP ... 260
SIGTERM ... 261
SIGTRAP ... 261
SIGTSTP ... 261
SIGTTIN ... 261
SIGTTOU ... 261
SIGURG ... 261
SIGUSR1 ... 261
SIGUSR2 ... 261
SIGVTALRM ... 261
SIGWINCH ... 261
SIGXCPU ... 261
SIGXFSZ ... 261
size of <sys-stat> ... 255
size.x of <win:console-screen-buffer-info> ... 710
size.y of <win:console-screen-buffer-info> ... 710
slots of <class> ... 302
SO_BROADCAST ... 408
SO_ERROR ... 408
SO_KEEPALIVE ... 407
SO_OOBINLINE ... 408
SO_PRIORITY ... 408
SO_REUSEADDR ... 408
SO_TYPE ... 408
SOCK_DGRAM ... 404
SOCK_RAW ... 404
SOCK_STREAM ... 404
socktype of <sys-addrinfo> ... 410
SOL_IP ... 407
SOL_SOCKET ... 407
SOL_TCP ... 407
source of <mime-part> ... 734
span of <read-error> ... 214
specific of <condition-variable> ... 465
specific of <mutex> ... 463
specific of <thread> ... 460
sql-string of <sql-parse-error> ... 790
ssax:Prefix-XML ... 750
start of <sys-flock> ... 372
state of <mutex> ... 463
STD_ERROR_HANDLE ... 712
STD_INPUT_HANDLE ... 712
STD_OUTPUT_HANDLE ... 712
stream-null ... 809
string-ci-comparator ... 107
string-comparator ... 107
subtype of <mime-part> ... 733
sync of <gdbm> ... 688
sys of <time-result> ... 469
syslog-facility of <log-drain> ... 394
syslog-option of <log-drain> ... 394
syslog-priority of <log-drain> ... 394

T

tag of <edn-object> ... 781
TCIFLUSH ... 450
TCIOFF ... 451
TCIOFLUSH ... 450
TCION ... 451
TCOFLUSH ... 450
TCOOFF ... 451
TCOON ... 451
TCSADRAIN ... 450
TCSAFLUSH ... 450
TCSANOW ... 450
terminator of <terminated-thread-exception> ... 467
thread of <thread-exception> ... 467
time-duration ... 600
time-monotonic ... 600
time-process ... 601
time-tai ... 600
time-thread ... 601
time-utc ... 600
transfer-encoding of <mime-part> ... 734
transfer-type of <ftp-connection> ... 718
type of <mime-part> ... 733
type of <sys-flock> ... 371

type of <sys-stat> 255
type of <time> 270

U

u16vector-comparator 482
u32vector-comparator 482
u64vector-comparator 482
u8vector-comparator 482
uid of <sys-passwd> 258
uid of <sys-stat> 255
user of <time-result> 469
uvector-comparator 107

V

value-convert of <dbm> 684
vector-comparator 107

W

W_OK 256
wday of <sys-tm> 270
whence of <sys-flock> 372
width of <write-controls> 232
window-buffer-size.x of <win:input-record> .. 711
window-buffer-size.y of <win:input-record> .. 711
window.bottom of <win:console-screen-buffer-info> 710
window.left of <win:console-screen-buffer-info> 710
window.right of <win:console-screen-buffer-info> 710
window.top of <win:console-screen-buffer-info> 710

X

X_OK 256

Y

yday of <sys-tm> 270
year of <date> 602
year of <sys-tm> 270

Z

Z_ASCII 745
Z_BEST_COMPRESSION 744
Z_BEST_SPEED 744
Z_BINARY 745
Z_DEFAULT_COMPRESSION 744
Z_DEFAULT_STRATEGY 744
Z_FILTERED 744
Z_FIXED 745
Z_HUFFMAN_ONLY 744
Z_NO_COMPRESSION 744
Z_RLE 744
Z_TEXT 745
Z_UNKNOWN 745
zone-offset of <date> 602