CHICKEN User's Manual - The User's Manual

Chicken User's Manual

1 Introduction.....1

2 Basic mode of operation.....3

3 Introduction.....4

4 Compiler command line format.....5

5 Runtime options.....10

6 Examples.....12
- 6.1 A simple example (with one source file).....12
  - 6.1.1 Writing your source file.....12
  - 6.1.2 Compiling your program.....12
  - 6.1.3 Running your program.....12

6.2 An example with multiple files.....13
- 6.2.1 Writing your source files.....13
- 6.2.2 Compiling and running your program.....13

7 Extending the compiler.....15

8 Distributing compiled C files.....16

9 Using the interpreter.....17
- 9.1 Interpreter command line format.....17
- 9.2 Writing Scheme scripts.....18
- 9.3 Toplevel commands.....19
- 9.4 toplevel-command.....20
- 9.5 History access.....20
- 9.6 set-describer!.....20
- 9.7 Auto-completion and edition.....21
- 9.8 Accessing documentation.....21

10 Supported language.....22

11 Deviations from the standard.....23

12 Extensions to the standard.....24

13 Non standard read syntax.....27
- 13.1 Multiline Block Comment.....27
- 13.2 Expression Comment.....27
- 13.3 External Representation.....27
- 13.4 Syntax Expression.....27
- 13.5 Location Expression.....27
- 13.6 Keyword.....27
- 13.7 Multiline String Constant.....28
- 13.8 Multiline String Constant with Embedded Expressions.....28
- 13.9 Foreign Declare.....28
- 13.10 Sharp Prefixed Symbol.....28
- 13.11 Bang.....29
  - 13.11.1 Line Comment.....29
  - 13.11.2 Eof Object.....29

Chicken User's Manual

13 Non standard read syntax
13.11.3 DSSSL Formal Parameter List Annotation....29
13.11.4 Read Mark Invocation....29
13.12 Case Sensitive Expression....29
13.13 Case Insensitive Expression....29
13.14 Conditional Expansion....30

14 Non-standard macros and special forms....31
14.1 Making extra libraries and extensions available....31
14.1.1 require-extension....31
14.1.2 define-extension....32
14.2 Binding forms for optional arguments....32
14.2.1 :optional....33
14.2.2 case-lambda....33
14.2.3 let-optionals....33
14.2.4 let-optionals*....33
14.3 Other binding forms....34
14.3.1 and-let*....34
14.3.2 rec....34
14.3.3 cut....34
14.3.4 define-values....34
14.3.5 fluid-let....35
14.3.6 let-values....35
14.3.7 let*-values....35
14.3.8 letrec-values....35
14.3.9 parameterize....35
14.3.10 receive....36
14.3.11 set!-values....36
14.4 Substitution forms and macros....36
14.4.1 define-constant....36
14.4.2 define-inline....36
14.4.3 define-macro....37
14.4.4 define-for-syntax....37
14.5 Conditional forms....37
14.5.1 switch....37
14.5.2 unless....37
14.5.3 when....38
14.6 Record structures....38
14.6.1 define-record....38
14.6.2 define-record-printer....38
14.6.3 define-record-type....39
14.7 Other forms....39
14.7.1 assert....39
14.7.2 cond-expand....39
14.7.3 ensure....40
14.7.4 eval-when....40
14.7.5 include....40
14.7.6 nth-value....41
14.7.7 time....41

15 Pattern Matching....42
15.1 Pattern Matching Expressions....42
15.2 Patterns....44

Chicken User's Manual

15 Pattern Matching
15.3 Match Failure....45
15.4 Record Structures Pattern....46
15.5 Code Generation....46

16 Declarations....48
16.1 declare....48
16.2 always-bound....48
16.3 block....48
16.4 block-global....48
16.5 hide....48
16.6 bound-to-procedure....48
16.7 c-options....49
16.8 check-c-syntax....49
16.9 compress-literals....49
16.10 constant....49
16.11 export....49
16.12 emit-exports....49
16.13 emit-external-prototypes-first....50
16.14 disable-interrupts....50
16.15 disable-warning....50
16.16 import....50
16.17 inline....50
16.18 inline-limit....51
16.19 interrupts-enabled....51
16.20 keep-shadowed-macros....51
16.21 lambda-lift....51
16.22 link-options....51
16.23 no-argc-checks....51
16.24 no-bound-checks....52
16.25 no-procedure-checks....52
16.26 post-process....52
16.27 number-type....52
16.28 fixnum-arithmetic....52
16.29 run-time-macros....52
16.30 standard-bindings....52
16.31 extended-bindings....53
16.32 usual-integrations....53
16.33 unit....53
16.34 unsafe....53
16.35 uses....53

17 Parameters....55
17.1 make-parameter....55
17.2 case-sensitive....55
17.3 dynamic-load-libraries....55
17.4 command-line-arguments....55
17.5 current-read-table....56
17.6 exit-handler....56
17.7 eval-handler....56
17.8 force-finalizers....56
17.9 implicit-exit-handler....56
17.10 keyword-style....56

Chicken User's Manual

17 Parameters
17.11 load-verbose......56
17.12 repl-prompt......56
17.13 reset-handler......57

18 Unit library......58
18.1 Arithmetic......58
18.1.1 add1......58
18.1.2 sub1......58
18.1.3 bitwise-and......58
18.1.4 bitwise-ior......58
18.1.5 bitwise-xor......58
18.1.6 bitwise-not......58
18.1.7 arithmetic-shift......58
18.1.8 bit-set?......59
18.1.9 fixnum?......59
18.1.10 fx+......59
18.1.11 fx-......59
18.1.12 fx*......59
18.1.13 fx/......59
18.1.14 fxmod......59
18.1.15 fxneg......59
18.1.16 fxmin......59
18.1.17 fxmax......59
18.1.18 fx=......59
18.1.19 fx>......60
18.1.20 fx<......60
18.1.21 fx>=......60
18.1.22 fx<=......60
18.1.23 fxand......60
18.1.24 fxior......60
18.1.25 fxxor......60
18.1.26 fxnot......60
18.1.27 fxshl......60
18.1.28 fxshr......60
18.1.29 flonum?......61
18.1.30 fp+......61
18.1.31 fp-......61
18.1.32 fp*......61
18.1.33 fp/......61
18.1.34 fpneg......61
18.1.35 fpmin......61
18.1.36 fpmax......61
18.1.37 fp=......61
18.1.38 fp>......61
18.1.39 fp<......62
18.1.40 fp>=......62
18.1.41 fp<=......62
18.1.42 signum......62
18.1.43 finite?......62
18.2 File Input/Output......62
18.2.1 current-output-port......62
18.2.2 current-error-port......63

Chicken User's Manual

18 Unit library
18.2.3 flush-output....63
18.2.4 port-name....63
18.2.5 port-position....63
18.2.6 set-port-name!....63
18.3 Files....64
18.3.1 delete-file....64
18.3.2 file-exists?....64
18.3.3 rename-file....64
18.4 String ports....64
18.4.1 get-output-string....64
18.4.2 open-input-string....64
18.4.3 open-output-string....64
18.5 Feature identifiers....65
18.5.1 features....65
18.5.2 feature?....65
18.5.3 register-feature!....65
18.5.4 unregister-feature!....65
18.6 Keywords....65
18.6.1 get-keyword....66
18.6.2 keyword?....66
18.6.3 keyword->string....66
18.6.4 string->keyword....66
18.7 Exceptions....66
18.7.1 condition-case....66
18.7.2 breakpoint....67
18.8 Environment information and system interface....68
18.8.1 argv....68
18.8.2 exit....68
18.8.3 build-platform....69
18.8.4 chicken-version....69
18.8.5 errno....69
18.8.6 getenv....69
18.8.7 machine-byte-order....69
18.8.8 machine-type....70
18.8.9 on-exit....70
18.8.10 software-type....70
18.8.11 software-version....70
18.8.12 c-runtime....71
18.8.13 system....71
18.9 Execution time....71
18.9.1 cpu-time....71
18.9.2 current-milliseconds....71
18.9.3 current-seconds....71
18.9.4 current-gc-milliseconds....72
18.10 Interrupts and error-handling....72
18.10.1 enable-warnings....72
18.10.2 error....72
18.10.3 get-call-chain....72
18.10.4 print-call-chain....72
18.10.5 print-error-message....73
18.10.6 procedure-information....73
18.10.7 reset....73

Chicken User's Manual

18 Unit library
18.10.8 warning....73
18.10.9 singlestep....73
18.11 Garbage collection....73
18.11.1 gc....73
18.11.2 memory-statistics....74
18.11.3 set-finalizer!....74
18.11.4 set-gc-report!....74
18.12 Other control structures....74
18.12.1 andmap....74
18.12.2 ormap....74
18.12.3 promise?....75
18.13 String utilities....75
18.13.1 reverse-list->string....75
18.14 Generating uninterned symbols....75
18.14.1 gensym....75
18.14.2 string->uninterned-symbol....75
18.15 Standard Input/Output....75
18.15.1 port?....75
18.15.2 print....76
18.15.3 print*....76
18.16 User-defined named characters....76
18.16.1 char-name....76
18.17 Vectors....76
18.17.1 vector-copy!....77
18.17.2 vector-resize....77
18.18 The unspecified value....77
18.18.1 void....77
18.19 Continuations....77
18.19.1 call/cc....77
18.19.2 continuation-capture....77
18.19.3 continuation?....78
18.19.4 continuation-graft....78
18.19.5 continuation-return....78
18.20 Setters....78
18.20.1 setter....78
18.20.2 getter-with-setter....79
18.21 Reader extensions....79
18.21.1 define-reader-ctor....79
18.21.2 set-read-syntax!....79
18.21.3 set-sharp-read-syntax!....80
18.21.4 set-parameterized-read-syntax!....80
18.21.5 copy-read-table....80

19 Unit eval....81
19.1 Loading code....81
19.1.1 load....81
19.1.2 load-relative....81
19.1.3 load-noisily....81
19.1.4 load-library....82
19.1.5 set-dynamic-load-mode!....82
19.2 Read-eval-print loop....82
19.2.1 repl....83

Chicken User's Manual

19 Unit eval

19.3 Macros.....83
- 19.3.1 get-line-number.....83
- 19.3.2 macro?.....83
- 19.3.3 macroexpand.....83
- 19.3.4 macroexpand-1.....83
- 19.3.5 undefine-macro!.....83
- 19.3.6 syntax-error.....84

19.4 Loading extension libraries.....84
- 19.4.1 repository-path.....84
- 19.4.2 extension-information.....84
- 19.4.3 provide.....84
- 19.4.4 provided?.....84
- 19.4.5 require.....85
- 19.4.6 set-extension-specifier!.....85

19.5 System information.....85
- 19.5.1 chicken-home.....85

19.6 Eval.....86
- 19.6.1 eval.....86

20 Unit extras.....87
- 20.1 Lists.....87
  - 20.1.1 alist-ref.....87
  - 20.1.2 alist-update!.....87
  - 20.1.3 atom?.....87
  - 20.1.4 rassoc.....87
  - 20.1.5 butlast.....87
  - 20.1.6 chop.....88
  - 20.1.7 compress.....88
  - 20.1.8 flatten.....88
  - 20.1.9 intersperse.....88
  - 20.1.10 join.....88
  - 20.1.11 shuffle.....89
  - 20.1.12 tail?.....89

20.2 String-port extensions.....89
- 20.2.1 call-with-input-string.....89
- 20.2.2 call-with-output-string.....89
- 20.2.3 with-input-from-string.....89
- 20.2.4 with-output-to-string.....89

20.3 Formatted output.....90
- 20.3.1 printf.....90

20.4 Hash tables.....91

20.5 Queues.....91
- 20.5.1 list->queue.....91
- 20.5.2 make-queue.....91
- 20.5.3 queue?.....91
- 20.5.4 queue->list.....92
- 20.5.5 queue-add!.....92
- 20.5.6 queue-empty?.....92
- 20.5.7 queue-first.....92
- 20.5.8 queue-last.....92
- 20.5.9 queue-remove!.....92
- 20.5.10 queue-push-back!.....92

Chicken User's Manual

20 Unit extras
20.5.11 queue-push-back-list!....93
20.6 Sorting....93
20.6.1 merge....93
20.6.2 sort....93
20.6.3 sorted?....93
20.7 Random numbers....93
20.7.1 random....93
20.7.2 randomize....94
20.8 Input/Output extensions....94
20.8.1 make-input-port....94
20.8.2 make-output-port....94
20.8.3 pretty-print....94
20.8.4 pretty-print-width....94
20.8.5 read-file....95
20.8.6 read-line....95
20.8.7 write-line....95
20.8.8 read-lines....95
20.8.9 read-string....95
20.8.10 read-string!....95
20.8.11 write-string....95
20.8.12 read-token....96
20.8.13 with-error-output-to-port....96
20.8.14 with-input-from-port....96
20.8.15 with-output-to-port....96
20.9 Strings....96
20.9.1 conc....96
20.9.2 ->string....97
20.9.3 string-chop....97
20.9.4 string-chomp....97
20.9.5 string-compare3....97
20.9.6 string-intersperse....97
20.9.7 string-split....98
20.9.8 string-translate....98
20.9.9 string-translate*....98
20.9.10 substring=?....98
20.9.11 substring-index....98
20.10 Combinators....99
20.10.1 any?....99
20.10.2 constantly....99
20.10.3 complement....99
20.10.4 compose....99
20.10.5 conjoin....100
20.10.6 disjoin....100
20.10.7 each....100
20.10.8 flip....100
20.10.9 identity....100
20.10.10 project....101
20.10.11 list-of....101
20.10.12 noop....101
20.10.13 o....101
20.11 Binary searching....101
20.11.1 binary-search....101

Chicken User's Manual

21 Unit srfi-1........102

22 Unit srfi-4........103
- 22.1 make-XXXvector........103
- 22.2 u8vector->byte-vector........103
- 22.3 s8vector->byte-vector........103
- 22.4 u16vector->byte-vector........103
- 22.5 s16vector->byte-vector........103
- 22.6 u32vector->byte-vector........103
- 22.7 s32vector->byte-vector........103
- 22.8 f32vector->byte-vector........103
- 22.9 f64vector->byte-vector........104
- 22.10 byte-vector->u8vector........104
- 22.11 byte-vector->s8vector........104
- 22.12 byte-vector->u16vector........104
- 22.13 byte-vector->s16vector........104
- 22.14 byte-vector->u32vector........104
- 22.15 byte-vector->s32vector........104
- 22.16 byte-vector->f32vector........104
- 22.17 byte-vector->f64vector........104
- 22.18 subu8vector........105
- 22.19 subu16vector........105
- 22.20 subu32vector........105
- 22.21 subs8vector........105
- 22.22 subs16vector........105
- 22.23 subs32vector........105
- 22.24 subf32vector........105
- 22.25 subf64vector........105
- 22.26 read-u8vector........106
- 22.27 read-u8vector!........106
- 22.28 write-u8vector........106

23 Unit srfi-13........107

24 Unit srfi-14........108

25 Unit match........109

26 Unit regex........110
- 26.1 grep........110
- 26.2 glob->regexp........110
- 26.3 regexp........110
- 26.4 regexp?........110
- 26.5 string-match........111
- 26.6 string-match-positions........111
- 26.7 string-search........111
- 26.8 string-search-positions........111
- 26.9 string-split-fields........111
- 26.10 string-substitute........112
- 26.11 string-substitute*........112
- 26.12 regexp-escape........112

Chicken User's Manual

27 Unit srfi-18....113
27.1 thread-signal!....113
27.2 thread-quantum....113
27.3 thread-quantum-set!....113
27.4 thread-suspend!....114
27.5 thread-resume!....114
27.6 time->milliseconds....114
27.7 time->milliseconds....114

28 Unit posix....115
28.1 Directories....115
28.1.1 change-directory....115
28.1.2 current-directory....115
28.1.3 create-directory....115
28.1.4 delete-directory....115
28.1.5 directory....115
28.1.6 directory?....116
28.1.7 glob....116
28.1.8 set-root-directory!....116
28.2 Pipes....116
28.2.1 call-with-input-pipe....116
28.2.2 call-with-output-pipe....116
28.2.3 close-input-pipe....116
28.2.4 close-output-pipe....117
28.2.5 create-pipe....117
28.2.6 open-input-pipe....117
28.2.7 open-output-pipe....117
28.2.8 pipe/buf....117
28.2.9 with-input-from-pipe....117
28.2.10 with-output-to-pipe....117
28.3 Fifos....118
28.3.1 create-fifo....118
28.3.2 fifo?....118
28.4 File descriptors and low-level I/O....118
28.4.1 duplicate-fileno....118
28.4.2 file-close....119
28.4.3 file-open....119
28.4.4 file-mkstemp....119
28.4.5 file-read....119
28.4.6 file-select....119
28.4.7 file-write....120
28.4.8 file-control....120
28.4.9 fcntl/dupfd....120
28.4.10 fcntl/getfd....120
28.4.11 fcntl/setfd....120
28.4.12 fcntl/getfl....120
28.4.13 fcntl/setfl....120
28.4.14 fileno/stdin....120
28.4.15 fileno/stdout....121
28.4.16 fileno/stderr....121
28.4.17 open/rdonly....121
28.4.18 open/wronly....121
28.4.19 open/rdwr....121

Chicken User's Manual

28 Unit posix
28.4.20 open/read....121
28.4.21 open/write....121
28.4.22 open/creat....121
28.4.23 open/append....121
28.4.24 open/excl....121
28.4.25 open/noctty....121
28.4.26 open/nonblock....121
28.4.27 open/trunc....121
28.4.28 open/sync....122
28.4.29 open/fsync....122
28.4.30 open/binary....122
28.4.31 open/text....122
28.4.32 open-input-file*....122
28.4.33 open-output-file*....122
28.4.34 port->fileno....122
28.5 Retrieving file attributes....122
28.5.1 file-access-time....122
28.5.2 file-change-time....122
28.5.3 file-modification-time....123
28.5.4 file-stat....123
28.5.5 file-position....123
28.5.6 file-size....123
28.5.7 regular-file?....123
28.6 Changing file attributes....123
28.6.1 file-truncate....124
28.6.2 set-file-position!....124
28.7 Processes....124
28.7.1 current-process-id....124
28.7.2 parent-process-id....124
28.7.3 process-execute....124
28.7.4 process-fork....125
28.7.5 process-run....125
28.7.6 process-signal....125
28.7.7 process-wait....125
28.7.8 process....125
28.7.9 process*....126
28.7.10 sleep....126
28.8 Hard and symbolic links....126
28.8.1 symbolic-link?....126
28.8.2 create-symbolic-link....126
28.8.3 read-symbolic-link....127
28.8.4 file-link....127
28.9 Permissions....127
28.9.1 file-owner....127
28.9.2 file-permissions....127
28.9.3 file-read-access?....127
28.9.4 file-write-access?....127
28.9.5 file-execute-access?....127
28.9.6 change-file-mode....128
28.9.7 change-file-owner....128
28.9.8 current-user-id....128
28.9.9 current-group-id....128

Chicken User's Manual

28 Unit posix
28.9.10 current-effective-user-id 128
28.9.11 current-effective-group-id 128
28.9.12 process-group-id 128
28.9.13 group-information 129
28.9.14 get-groups 129
28.9.15 set-groups! 129
28.9.16 initialize-groups 129
28.9.17 perm/irusr 129
28.9.18 perm/iwusr 129
28.9.19 perm/ixusr 129
28.9.20 perm/irgrp 130
28.9.21 perm/iwgrp 130
28.9.22 perm/ixgrp 130
28.9.23 perm/iroth 130
28.9.24 perm/iwoth 130
28.9.25 perm/ixoth 130
28.9.26 perm/irwxu 130
28.9.27 perm/irwxg 130
28.9.28 perm/irwxo 130
28.9.29 perm/isvtx 130
28.9.30 perm/isuid 130
28.9.31 perm/isgid 130
28.9.32 set-user-id! 130
28.9.33 set-group-id! 131
28.9.34 set-process-group-id! 131
28.9.35 user-information 131
28.9.36 create-session 131
28.10 Record locking 131
28.10.1 file-lock 131
28.10.2 file-lock/blocking 132
28.10.3 file-test-lock 132
28.10.4 file-unlock 132
28.11 Signal handling 132
28.11.1 set-alarm! 132
28.11.2 set-signal-handler! 132
28.11.3 signal-handler 133
28.11.4 set-signal-mask! 133
28.11.5 signal-mask 133
28.11.6 signal-masked? 133
28.11.7 signal-mask! 133
28.11.8 signal-unmask! 133
28.11.9 signal/term 133
28.11.10 signal/kill 134
28.11.11 signal/int 134
28.11.12 signal/hup 134
28.11.13 signal/fpe 134
28.11.14 signal/ill 134
28.11.15 signal/segv 134
28.11.16 signal/abrt 134
28.11.17 signal/trap 134
28.11.18 signal/quit 134
28.11.19 signal/alrm 134

Chicken User's Manual

28 Unit posix
28.11.20 signal/vtalrm....134
28.11.21 signal/prof....134
28.11.22 signal/io....134
28.11.23 signal/urg....135
28.11.24 signal/chld....135
28.11.25 signal/cont....135
28.11.26 signal/stop....135
28.11.27 signal/tstp....135
28.11.28 signal/pipe....135
28.11.29 signal/xcpu....135
28.11.30 signal/xfsz....135
28.11.31 signal/usr1....135
28.11.32 signal/usr2....135
28.11.33 signal/winch....135
28.12 Environment access....135
28.12.1 current-environment....136
28.12.2 setenv....136
28.12.3 unsetenv....136
28.13 Memory mapped I/O....136
28.13.1 memory-mapped-file?....136
28.13.2 map-file-to-memory....136
28.13.3 memory-mapped-file-pointer....137
28.13.4 unmap-file-from-memory....137
28.14 Date and time routines....137
28.14.1 seconds->local-time....137
28.14.2 local-time->seconds....138
28.14.3 local-timezone-abbreviation....138
28.14.4 seconds->string....138
28.14.5 seconds->utc-time....138
28.14.6 utc-time->seconds....138
28.14.7 time->string....138
28.15 Raw exit....139
28.15.1 _exit....139
28.16 ERRNO values....139
28.16.1 errno/perm....139
28.16.2 errno/noent....139
28.16.3 errno/srch....139
28.16.4 errno/intr....139
28.16.5 errno/io....139
28.16.6 errno/noexec....139
28.16.7 errno/badf....139
28.16.8 errno/child....139
28.16.9 errno/nomem....139
28.16.10 errno/acces....140
28.16.11 errno/fault....140
28.16.12 errno/busy....140
28.16.13 errno/notdir....140
28.16.14 errno/isdir....140
28.16.15 errno/inval....140
28.16.16 errno/mfile....140
28.16.17 errno/nospc....140
28.16.18 errno/spipe....140

Chicken User's Manual

28 Unit posix
28.16.19 errno/pipe....140
28.16.20 errno/again....140
28.16.21 errno/rofs....140
28.16.22 errno/exist....140
28.16.23 errno/wouldblock....141
28.17 Finding files....141
28.17.1 find-files....141
28.18 Getting the hostname and system information....141
28.18.1 get-host-name....141
28.18.2 system-information....141
28.19 Setting the file buffering mode....141
28.19.1 set-buffering-mode!....142
28.20 Terminal ports....142
28.20.1 terminal-name....142
28.20.2 terminal-port?....142
28.21 How Scheme procedures relate to UNIX C functions....142
28.22 Windows specific notes....147
28.22.1 spawn/overlay....147
28.22.2 spawn/wait....147
28.22.3 spawn/nowait....148
28.22.4 spawn/nowaito....148
28.22.5 spawn/detach....148
28.22.6 process-spawn....148

29 Unit utils....149
29.1 Environment Query....149
29.1.1 apropos....149
29.1.2 apropos-list....149
29.2 Pathname operations....149
29.2.1 absolute-pathname?....149
29.2.2 decompose-pathname....149
29.2.3 make-pathname....150
29.2.4 make-absolute-pathname....150
29.2.5 pathname-directory....150
29.2.6 pathname-file....150
29.2.7 pathname-extension....150
29.2.8 pathname-replace-directory....150
29.2.9 pathname-replace-file....150
29.2.10 pathname-replace-extension....151
29.2.11 pathname-strip-directory....151
29.2.12 pathname-strip-extension....151
29.3 Temporary files....151
29.3.1 create-temporary-file....151
29.4 Deleting a file without signalling an error....151
29.4.1 delete-file*....151
29.5 Iterating over input lines and files....151
29.5.1 for-each-line....152
29.5.2 for-each-argv-line....152
29.5.3 port-for-each....152
29.5.4 port-map....152
29.6 Executing shell commands with formatstring and error checking....152
29.6.1 system*....152

Chicken User's Manual

29 Unit utils
29.7 Reading a file's contents....153
29.7.1 read-all....153
29.8 Miscellaneous handy things....153
29.8.1 shift!....153
29.8.2 unshift!....153

30 Unit tcp....154
30.1 tcp-listen....154
30.2 tcp-listener?....154
30.3 tcp-close....154
30.4 tcp-accept....154
30.5 tcp-accept-ready?....154
30.6 tcp-listener-port....155
30.7 tcp-listener-fileno....155
30.8 tcp-connect....155
30.9 tcp-addresses....155
30.10 tcp-port-numbers....155
30.11 tcp-abandon-port....156
30.12 tcp-buffer-size....156

31 Example....157

32 Unit lolevel....158
32.1 Foreign pointers....158
32.1.1 address->pointer....158
32.1.2 allocate....158
32.1.3 free....158
32.1.4 null-pointer....158
32.1.5 null-pointer?....158
32.1.6 object->pointer....159
32.1.7 pointer?....159
32.1.8 pointer=?....159
32.1.9 pointer->address....159
32.1.10 pointer->object....159
32.1.11 pointer-offset....159
32.1.12 pointer-u8-ref....160
32.1.13 pointer-s8-ref....160
32.1.14 pointer-u16-ref....160
32.1.15 pointer-s16-ref....160
32.1.16 pointer-u32-ref....160
32.1.17 pointer-s32-ref....160
32.1.18 pointer-f32-ref....160
32.1.19 pointer-f64-ref....161
32.1.20 pointer-u8-set!....161
32.1.21 pointer-s8-set!....161
32.1.22 pointer-u16-set!....161
32.1.23 pointer-s16-set!....161
32.1.24 pointer-u32-set!....161
32.1.25 pointer-s32-set!....162
32.1.26 pointer-f32-set!....162
32.1.27 pointer-f64-set!....162
32.1.28 align-to-word....162

Chicken User's Manual

32 Unit lolevel
32.2 Tagged pointers....162
32.2.1 tag-pointer....162
32.2.2 tagged-pointer?....163
32.2.3 pointer-tag....163
32.3 Extending procedures with data....163
32.3.1 extend-procedure....163
32.3.2 extended-procedure?....163
32.3.3 procedure-data....163
32.3.4 set-procedure-data!....163
32.4 Bytevectors....164
32.4.1 byte-vector....164
32.4.2 byte-vector?....164
32.4.3 byte-vector-fill!....164
32.4.4 byte-vector->list....164
32.4.5 byte-vector->string....164
32.4.6 byte-vector-length....165
32.4.7 byte-vector-ref....165
32.4.8 byte-vector-set!....165
32.4.9 list->byte-vector....165
32.4.10 make-byte-vector....165
32.4.11 make-static-byte-vector....165
32.4.12 static-byte-vector->pointer....166
32.4.13 string->byte-vector....166
32.5 Data in unmanaged memory....166
32.5.1 object-evict....166
32.5.2 object-evict-to-location....166
32.5.3 object-evicted?....167
32.5.4 object-size....167
32.5.5 object-release....167
32.5.6 object-unevict....167
32.6 Locatives....167
32.6.1 make-locative....167
32.6.2 make-weak-locative....168
32.6.3 locative?....168
32.6.4 locative-ref....168
32.6.5 locative-set!....168
32.6.6 locative->object....168
32.7 Accessing toplevel variables....168
32.7.1 global-bound?....169
32.7.2 global-ref....169
32.7.3 global-set!....169
32.8 Low-level data access....169
32.8.1 block-ref....169
32.8.2 block-set!....169
32.8.3 object-copy....170
32.8.4 make-record-instance....170
32.8.5 move-memory!....170
32.8.6 number-of-bytes....170
32.8.7 number-of-slots....171
32.8.8 record-instance?....171
32.8.9 record->vector....171
32.9 Procedure-call- and variable reference hooks....171

Chicken User's Manual

32 Unit lolevel
32.9.1 set-invalid-procedure-call-handler!....171
32.9.2 unbound-variable-value....172
32.10 Magic....172
32.10.1 object-become!....172

33 Interface to external functions and variables....173

34 Accessing external objects....174
34.1 foreign-code....174
34.2 foreign-value....174
34.3 foreign-declare....174
34.4 define-foreign-type....174
34.5 define-foreign-variable....175
34.6 define-foreign-record....175
34.6.1 TYPENAME-SLOTNAME....176
34.6.2 TYPENAME-SLOTNAME-set!....176
34.6.3 constructor....176
34.6.4 destructor....176
34.6.5 rename....177
34.7 define-foreign-enum....177
34.8 foreign-lambda....178
34.9 foreign-lambda*....178
34.10 foreign-safe-lambda....178
34.11 foreign-safe-lambda*....178
34.12 foreign-primitive....179

35 Foreign type specifiers....180
35.1 scheme-object....180
35.2 bool....180
35.3 byte unsigned-byte....180
35.4 char unsigned-char....180
35.5 short unsigned-short....180
35.6 int unsigned-int int32 unsigned-int32....180
35.7 integer unsigned-integer integer32 unsigned-integer32 integer64....180
35.8 long unsigned-long....180
35.9 float double....181
35.10 number....181
35.11 symbol....181
35.12 scheme-pointer....181
35.13 nonnull-scheme-pointer....181
35.14 c-pointer....181
35.15 nonnull-c-pointer....181
35.16 scheme-or-c-pointer....182
35.17 [nonnull-] byte-vector....182
35.18 [nonnull-] u8vector [nonnull-] u16vector [nonnull-] u32vector [nonnull-] s8vector [nonnull-] s16vector [nonnull-] s32vector [nonnull-] f32vector [nonnull-] f64vector....182
35.19 c-string....182
35.20 nonnull-c-string....182
35.21 [nonnull-] c-string*....182
35.22 void....182
35.23 (const TYPE)....183
35.24 (enum NAME)....183

Chicken User's Manual

35 Foreign type specifiers
35.25 (pointer TYPE) (c-pointer TYPE)....183
35.26 (nonnull-pointer TYPE) (nonnull-c-pointer TYPE)....183
35.27 (ref TYPE)....183
35.28 (struct NAME)....183
35.29 (template TYPE ARGTYPE ...)....183
35.30 (union NAME)....183
35.31 (instance CNAME SCHEMECLASS)....184
35.32 (instance-ref CNAME SCHEMECLASS)....184
35.33 (function RESULTTYPE (ARGUMENTTYPE1 ... [...]) [CALLCONV])....184

36 Mappings....185

37 Embedding....186
37.1 CHICKEN_parse_command_line....186
37.2 CHICKEN_initialize....186
37.3 CHICKEN_run....186
37.4 return-to-host....187
37.5 CHICKEN_eval....187
37.6 CHICKEN_eval_string....187
37.7 CHICKEN_eval_to_string....187
37.8 CHICKEN_eval_string_to_string....187
37.9 CHICKEN_apply....187
37.10 CHICKEN_apply_to_string....188
37.11 CHICKEN_read....188
37.12 CHICKEN_load....188
37.13 CHICKEN_get_error_message....188
37.14 CHICKEN_yield....188
37.15 CHICKEN_continue....190
37.16 CHICKEN_new_gc_root....191
37.17 CHICKEN_delete_gc_root....191
37.18 CHICKEN_gc_root_ref....191
37.19 CHICKEN_gc_root_set....191
37.20 CHICKEN_global_lookup....191
37.21 CHICKEN_global_ref....191
37.22 CHICKEN_global_set....192

38 Callbacks....193
38.1 define-external....193
38.2 C_callback....193
38.3 C_callback_adjust_stack_limits....194

39 Locations....195
39.1 define-location....195
39.2 let-location....195
39.3 location....195
39.4 Other support procedures....196
39.5 argc+argv....196

40 C interface....197
40.1 C_save....197
40.2 C_restore....197
40.3 C_fix....197

Chicken User's Manual

40 C interface
40.4 C_make_character....197
40.5 C_SCHEME_END_OF_LIST....197
40.6 C_word C_SCHEME_END_OF_FILE....197
40.7 C_word C_SCHEME_FALSE....197
40.8 C_word C_SCHEME_TRUE....198
40.9 C_string....198
40.10 C_string2....198
40.11 C_intern2....198
40.12 C_intern3....198
40.13 C_pair....198
40.14 C_flonum....198
40.15 C_int_to_num....198
40.16 C_mpointer....198
40.17 C_vector....199
40.18 C_list....199
40.19 C_alloc....199
40.20 C_SIZEOF_LIST....199
40.21 C_SIZEOF_STRING....199
40.22 C_SIZEOF_VECTOR....199
40.23 C_SIZEOF_INTERNED_SYMBOL....200
40.24 C_SIZEOF_PAIR....200
40.25 C_SIZEOF_FLONUM....200
40.26 C_SIZEOF_POINTER....200
40.27 C_SIZEOF_LOCATIVE....200
40.28 C_SIZEOF_TAGGED_POINTER....200
40.29 C_character_code....200
40.30 C_unfix....200
40.31 C_flonum_magnitude....200
40.32 C_c_string....201
40.33 C_num_to_int....201
40.34 C_pointer_address....201
40.35 C_header_size....201
40.36 C_header_bits....201
40.37 C_block_item....201
40.38 C_u_i_car....201
40.39 C_u_i_car....202
40.40 C_data_pointer....202
40.41 C_make_header....202
40.42 C_mutate....202
40.43 C_symbol_value....202
40.44 C_gc_protect....202
40.45 C_gc_unprotect....203
40.46 C_post_gc_hook....203

41 chicken-setup....205
41.1 Extension libraries....205
41.2 Installing extensions....205
41.3 Creating extensions....205
41.4 Procedures and macros available in setup scripts....206
41.4.1 install-extension....206
41.4.2 install-program....207
41.4.3 install-script....208

Chicken User's Manual

41 chicken-setup
41.4.4 run....208
41.4.5 compile....208
41.4.6 make....208
41.4.7 patch....208
41.4.8 copy-file....208
41.4.9 move-file....209
41.4.10 remove-file*....209
41.4.11 find-library....209
41.4.12 find-header....209
41.4.13 try-compile....209
41.4.14 create-directory....209
41.4.15 installation-prefix....210
41.4.16 program-path....210
41.4.17 setup-root-directory....210
41.4.18 setup-build-directory....210
41.4.19 setup-verbose-flag....210
41.4.20 setup-install-flag....210
41.4.21 required-chicken-version....211
41.4.22 required-extension-version....211
41.4.23 cross-chicken....211
41.5 Examples for extensions....211
41.6 chicken-setup reference....214
41.7 Windows notes....215
41.8 Security....216
41.9 Other modes if installation....216
41.10 Linking extensions statically....216

42 Data Representation....218

43 Bugs and limitations....220

44 Introduction....221

45 General....222
45.1 Why yet another Scheme implementation?....222
45.2 What should I do if I find a bug?....222
45.3 Why are values defined with {{define-foreign-variable}} or {{define-constant}} or {{define-inline}} not seen outside of the containing source file?....222
45.4 How does {{cond-expand}} know which features are registered in used units?....222
45.5 Why are constants defined by {{define-constant}} not honoured in {{case}} constructs?....222
45.6 How can I enable case sensitive reading/writing in user code?....223
45.7 How can I change {{match-error-control}} during compilation?....223
45.8 Why doesn't CHICKEN support the full numeric tower by default?....223
45.9 How can I specialize a generic function method to match instances of every class?....223
45.10 Does CHICKEN support native threads?....224

46 Platform specific....225
46.1 How do I generate a DLL under MS Windows (tm) ?....225
46.2 How do I generate a GUI application under Windows(tm)?....225
46.3 Compiling very large files under Windows with the Microsoft C compiler fails with a message indicating insufficient heap space....225
46.4 When I run {{csi}} inside an emacs buffer under Windows, nothing happens....225

Chicken User's Manual

46 Platform specific
46.5 I load compiled code dynamically in a Windows GUI application and it crashes.....225
46.6 On Windows, {{csc.exe}} seems to be doing something wrong.....225

47 Customization.....227
47.1 How do I run custom startup code before the runtime-system is invoked?.....227
47.2 How can I add compiled user passes?.....227

48 Macros.....228
48.1 Why doesn't my fancy macro work in compiled code?.....228
48.2 Why are macros not visible outside of the compilation unit in which they are defined?.....228

49 Warnings and errors.....229
49.1 Why does my program crash when I use callback functions (from Scheme to C and back to Scheme again)?.....229
49.2 Why does the linker complain about a missing function {{ C ... toplevel}}?.....229
49.3 Why does the linker complain about a missing function {{ C toplevel}}?.....229
49.4 Why does my program crash when I compile a file with {{-unsafe}} or unsafe declarations?.....229
49.5 Why do I get a warning when I define a global variable named {{match}}?.....229
49.6 Why don't toplevel-continuations captured in interpreted code work?.....230
49.7 Why does {{define-reader-ctor}} not work in my compiled program?.....230
49.8 Why do built-in units, such as srfi-1, srfi-18, and posix fail to load?.....231
49.9 How can I increase the size of the trace shown when runtime errors are detected?.....231

50 Optimizations.....232
50.1 How can I obtain smaller executables?.....232
50.2 How can I obtain faster executables?.....232
50.3 Which non-standard procedures are treated specially when the {{extended-bindings}} or {{usual-integrations}} declaration or compiler option is used?.....232

51 Garbage collection.....234
51.1 Why does a loop that doesn't {{cons}} still trigger garbage collections?.....234
51.2 Why do finalizers not seem to work in simple cases in the interpeter?.....234

52 Interpreter.....235
52.1 Does CSI support history and autocompletion?.....235

53 Extensions.....236
53.1 How can I install Chicken eggs to a non-default location?.....236
53.2 Can I install chicken eggs as a non-root user?.....236

54 Acknowledgements.....237

55 Bibliography.....239

1 Introduction

(This document describes version 2.6)

CHICKEN is a compiler that translates Scheme source files into C, which in turn can be fed to a C-compiler to generate a standalone executable. An interpreter is also available and can be used as a scripting environment or for testing programs before compilation.

This package is distributed under the **BSD license** and as such is free to use and modify.

The method of compilation and the design of the runtime-system follow closely Henry Baker's CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. paper and expose a number of interesting properties:

- Consing (creation of data on the heap) is relatively inexpensive, because a generational garbage collection scheme is used, in which short-lived data structures are reclaimed extremely quickly.
- Moreover, `call-with-current-continuation` is practically for free and CHICKEN does not suffer under any performance penalties if first-class continuations are used in complex ways.

The generated C code is fully tail-recursive.

Some of the features supported by CHICKEN:

- SRFIs 0, 1, 2, 4, 6-19, 23, 25-31, 37-40, 42, 43, 45, 47, 55, 57, 60-63, 66, 69, 72, 78, 85 and 95.
- Lightweight threads based on first-class continuations
- Pattern matching with Andrew Wright's `match` package
- Record structures
- Extended comment- and string-literal syntaxes
- Libraries for regular expressions, string handling
- UNIX system calls and extended data structures
- Create interpreted or compiled shell scripts written in Scheme for UNIX or Windows
- Compiled C files can be easily distributed
- Allows the creation of fully self-contained statically linked executables
- On systems that support it, compiled code can be loaded dynamically

This manual is merely a reference for the CHICKEN system and assumes a working knowledge of Scheme.

The manual is split in the following sections:

Basic mode of operation
: Compiling Scheme files.

Using the compiler
: Explains how to use CHICKEN to compile programs and execute them.

Using the interpreter
: Invocation and usage of `csi`, the CHICKEN interpreter

Supported language
: The language implemented by CHICKEN (deviations from the standard and extensions).

Interface to external functions and variables
: Accessing C and C++ code and data.

chicken-setup
: Packaging and installing extension libraries.

Data representation
: How Scheme data is internally represented.

Bugs and limitations
: Yes, there are some.

FAQ
: A list of Frequently Asked Questions about CHICKEN (and their answers!).

Acknowledgements
: A list of some of the people that have contributed to make CHICKEN what it is.

Bibliography
: Links to documents that may be of interest.

2 Basic mode of operation

The compiler translates Scheme source code into fairly portable C that can be compiled and linked with most available C compilers. CHICKEN supports the generation of executables and libraries, linked either statically or dynamically. Compiled Scheme code can be loaded dynamically, or can be embedded in applications written in other languages. Separate compilation of modules is fully supported.

The most portable way of creating separately linkable entities is supported by so-called *unit*s. A unit is a single compiled object module that contains a number of toplevel expressions that are executed either when the unit is the *main* unit or if the unit is *used*. To use a unit, the unit has to be *declare*ed as used, like this:

```
(declare (uses UNITNAME))
```

The toplevel expressions of used units are executed in the order in which the units appear in the `uses` declaration. Units may be used multiple times and `uses` declarations may be circular (the unit is initialized at most once). To compile a file as a unit, add a `unit` declaration:

```
(declare (unit UNITNAME))
```

When compiling different object modules, make sure to have one main unit. This unit is called initially and initializes all used units before executing its toplevel expressions. The main-unit has no `unit` declaration.

Another method of using definitions in separate source files is to *include* them. This simply inserts the code in a given file into the current file:

```
(include "FILENAME")
```

Macro definitions are only available when processed by `include`. Macro definitions in separate units are not available, since they are defined at compile time, i.e the time when that other unit was compiled (macros can optionally be available at runtime, see `define-macro` in Substitution forms and macros).

On platforms that support dynamic loading of compiled code (like Windows and most ELF based systems like Linux or BSD) code can be compiled into a shared object (`.so`) and loaded dynamically into a running application.

Previous: index.html Next: Using the compiler

3 Introduction

The interface to `chicken` is intentionally simple. System dependent makefiles, shell-scripts or batch-files should perform any necessary steps before and after invocation of `chicken`. A program named `csc` provides a much simpler interface to the Scheme- and C-compilers and linker. Enter

```
csc -help
```

on the command line for more information.

4 Compiler command line format

```
chicken FILENAME {OPTION}
```

`FILENAME` is the complete pathname of the source file that is to be translated into C. A filename argument of `-` specifies that the source text should be read from standard input. Note that the filename has to be the first argument to `chicken`.

Possible options are:

-analyze-only
: Stop compilation after first analysis pass.

-benchmark-mode
: Equivalent to `-no-trace -no-lambda-info -optimize-level 3 -fixnum-arithmetic -disable-interrupts -block -lambda-lift`.

-block
: Enable block-compilation. When this option is specified, the compiler assumes that global variables are not modified outside this compilation-unit. Specifically, toplevel bindings are not seen by `eval` and unused toplevel bindings are removed.

-case-insensitive
: Enables the reader to read symbols case insensitive. The default is to read case sensitive (in violation of R5RS). This option registers the `case-insensitive` feature identifier.

-check-imports
: Search for references to undefined global variables. For each library unit accessed via `(declare (uses ...))`, the compiler will search a file named `UNITNAME.exports` in the current include path and load its contents into the *import-table* (if found). Also, export-information for extensions (accessed through `(require-extension ...)`) will be searched and stored in the import-table. If a required extension does not provide explicit export-information a `.exports` file is searched (as with used units). After the analysis phase of the compiler, referenced toplevel variables for which no assignment was found will generate a warning. Also, re-assignments of imported variables will trigger a warning.

-check-syntax
: Aborts compilation process after macro-expansion and syntax checks.

-compress-literals THRESHOLD
: Compiles quoted literals that exceed the size `THRESHOLD` as strings and parse the strings at run-time. This reduces the size of the code and speeds up compile-times of the host C compiler, but has a small run-time performance penalty. The size of a literal is computed by counting recursively the objects in the literal, so a vector counts as 1 plus the count of the elements, a pair counts as the counts of the car and the cdr, respectively. All other objects count 1.

-debug MODES
: Enables one or more compiler debugging modes. `MODES` is a string of characters that select debugging information about the compiler that will be printed to standard output.

```
     t          show time needed for compilation
     b          show breakdown of time needed for each compiler pass
     o          show performed optimizations
     r          show invocation parameters
     s          show program-size information and other statistics
     a          show node-matching during simplification
     p          show execution of compiler sub-passes
     l          show lambda-lifting information
     m          show GC statistics during compilation
     n          print the line-number database
     c          print every expression before macro-expansion
     u          lists all unassigned global variable references
```

```
    x          display information about experimental features
    D          when printing nodes, use node-tree output
    N          show the real-name mapping table
    U          show expressions after the secondary user pass
    0          show database before lambda-lifting pass
    L          show expressions after lambda-lifting
    M          show unit-information and syntax-/runtime-requirements
    1          show source expressions
    2          show canonicalized expressions
    3          show expressions converted into CPS
    4          show database after each analysis pass
    5          show expressions after each optimization pass
    6          show expressions after each inlining pass
    7          show expressions after complete optimization
    8          show database after final analysis
    9          show expressions after closure conversion
```

-debug-level LEVEL
: Selects amount of debug-information. `LEVEL` should be an integer.

```
    -debug-level 0             is equivalent to -no-trace -no-lambda-info
    -debug-level 1             is equivalent to -no-trace
    -debug-level 2             does nothing (the default)
```

-disable-interrupts
: Equivalent to the `(disable-interrupts)` declaration. No interrupt-checks are generated for compiled programs.

-disable-stack-overflow-checks
: Disables detection of stack overflows. This is equivalent to running the compiled executable with the `-:o` runtime option.

-disable-warning CLASS
: Disables specific class of warnings, may be given multiple times. The following classes are defined:

```
    usage              warnings related to command-line arguments
    type               warnings related to type-conversion
    ext                warnings related to extension libraries
    var                warnings related to variable- and syntax-definitions and
    const              warnings related to constant-definitions
    syntax             syntax-related warnings
    redef              warnings about redefinitions of standard- or extended-bi
    call               warnings related to known procedure calls
    ffi                warnings related to the foreign function interface
```

-dynamic
: This option should be used when compiling files intended to be loaded dynamically into a running Scheme program.

-epilogue FILENAME
: Includes the file named `FILENAME` at the end of the compiled source file. The include-path is not searched. This option may be given multiple times.

-emit-exports FILENAME
: Write exported toplevel variables to `FILENAME`.

-emit-external-prototypes-first
: Emit prototypes for callbacks defined with `define-external` before any other foreign declarations. This is sometimes useful, when C/C++ code embedded into the a Scheme program has

to access the callbacks. By default the prototypes are emitted after foreign declarations.

-explicit-use
: Disables automatic use of the units `library, eval` and `extras`. Use this option if compiling a library unit instead of an application unit.

-extend FILENAME
: Loads a Scheme source file or compiled Scheme program (on systems that support it) before compilation commences. This feature can be used to extend the compiler. This option may be given multiple times. The file is also searched in the current include path and in the extension-repository.

-extension
: Mostly equivalent to `-prelude '(define-extension <NAME>)'`, where `<NAME>` is the basename of the currently compiled file. Note that if you want to compile a file as a normal (dynamically loadable) extension library, you should also pass the `-shared` option.

-feature SYMBOL
: Registers `SYMBOL` to be a valid feature identifier for `cond-expand`.

-fixnum-arithmetic
: Equivalent to `(fixnum-arithmetic)` declaration. Assume all mathematical operations use small integer arguments.

-heap-size NUMBER
: Sets a fixed heap size of the generated executable to `NUMBER` bytes. The parameter may be followed by a `M` (`m`) or `K` (`k`) suffix which stand for mega- and kilobytes, respectively. The default heap size is 5 kilobytes. Note that only half of it is in use at every given time.

-heap-initial-size NUMBER
: Sets the size that the heap of the compiled application should have at startup time.

-heap-growth PERCENTAGE
: Sets the heap-growth rate for the compiled program at compile time (see: `-:hg`).

-heap-shrinkage PERCENTAGE
: Sets the heap-shrinkage rate for the compiled program at compile time (see: `-:hs`).

-help
: Print a summary of available options and the format of the command line parameters and exit the compiler.

-import FILENAME
: Read exports from linked or loaded libraries from given file. See also `-check-imports`. This is equivalent to declaring `(declare (import FILENAME))`. Implies `-check-imports`.

-include-path PATHNAME
: Specifies an additional search path for files included via the `include` special form. This option may be given multiple times. If the environment variable `CHICKEN_INCLUDE_PATH` is set, it should contain a list of alternative include pathnames separated by `;`. The environment variable `CHICKEN_HOME` is also considered as a search path.

-inline
: Enable procedure inlining for known procedures of a size below the threshold (which can be set through the `-inline-limit` option).

-inline-limit THRESHOLD
: Sets the maximum size of a potentially inlinable procedure. This option is only effective when inlining has been enabled with the `-inline` option. The default threshold is `10`.

-keyword-style STYLE
: Enables alternative keyword syntax, where `STYLE` may be either `prefix` (as in Common Lisp), `suffix` (as in DSSSL) or `none`. Any other value is ignored. The default is `suffix`.

-keep-shadowed-macros
: Do not remove macro definitions with the same name as assigned toplevel variables (the default is to remove the macro definition).

-lambda-lift
: Enable the optimization known as lambda-lifting.

-no-lambda-info
: Don't emit additional information for each `lambda` expression (currently the argument-list, after

alpha-conversion/renaming).

-no-trace
: Disable generation of tracing information. If a compiled executable should halt due to a runtime error, then a list of the name and the line-number (if available) of the last procedure calls is printed, unless `-no-trace` is specified. With this option the generated code is slightly faster.

-no-warnings
: Disable generation of compiler warnings.

-nursery NUMBER
-stack-size NUMBER
: Sets the size of the first heap-generation of the generated executable to `NUMBER` bytes. The parameter may be followed by a `M` (`m`) or `K` (`k`) suffix. The default stack-size depends on the target platform.

-optimize-leaf-routines
: Enable leaf routine optimization.

-optimize-level LEVEL
: Enables certain sets of optimization options. `LEVEL` should be an integer.

```
 -optimize-level 0          does nothing.
 -optimize-level 1          is equivalent to -optimize-leaf-routines
 -optimize-level 2          is currently the same as -optimize-level 1
 -optimize-level 3          is equivalent to -optimize-leaf-routines -unsafe
```

-output-file FILENAME
: Specifies the pathname of the generated C file. Default is `FILENAME.c`.

-postlude EXPRESSIONS
: Add `EXPRESSIONS` after all other toplevel expressions in the compiled file. This option may be given multiple times. Processing of this option takes place after processing of `-epilogue`.

-prelude EXPRESSIONS
: Add `EXPRESSIONS` before all other toplevel expressions in the compiled file. This option may be given multiple times. Processing of this option takes place before processing of `-prologue`.

-profile
-accumulate-profile
: Instruments the source code to count procedure calls and execution times. After the program terminates (either via an explicit `exit` or implicitly), profiling statistics are written to a file named `PROFILE`. Each line of the generated file contains a list with the procedure name, the number of calls and the time spent executing it. Use the `chicken-profile` program to display the profiling information in a more user-friendly form. Enter `chicken-profile` with no arguments at the command line to get a list of available options. The `-accumulate-profile` option is similar to `-profile`, but the resulting profile information will be appended to any existing `PROFILE` file. `chicken-profile` will merge and sum up the accumulated timing information, if several entries for the same procedure calls exist.

-profile-name FILENAME
: Specifies name of the generated profile information (which defaults to `PROFILE`. Implies `-profile`.

-prologue FILENAME
: Includes the file named `FILENAME` at the start of the compiled source file. The include-path is not searched. This option may be given multiple times.

-quiet
: Disables output of compile information.

-raw
: Disables the generation of any implicit code that uses the Scheme libraries (that is all runtime system files besides `runtime.c` and `chicken.h`).

-require-extension NAME
: Loads the extension `NAME` before the compilation process commences. This is identical to adding `(require-extension NAME)` at the start of the compiled program. If `-uses NAME` is also

given on the command line, then any occurrences of `-require-extension NAME` are replaced with `(declare (uses NAME))`.

-run-time-macros
: Makes macros also available at run-time. By default macros are not available at run-time.

-to-stdout
: Write compiled code to standard output instead of creating a `.c` file.

-unit NAME
: Compile this file as a library unit. Equivalent to `-prelude "(declare (unit NAME))"`

-unsafe
: Disable runtime safety checks.

-unsafe-libraries
: Marks the generated file for being linked with the unsafe runtime system. This should be used when generating shared object files that are to be loaded dynamically. If the marker is present, any attempt to load code compiled with this option will signal an error.

-uses NAME
: Use definitions from the library unit `NAME`. This is equivalent to `-prelude "(declare (uses NAME))"`

-no-usual-integrations
: Specifies that standard procedures and certain internal procedures may be redefined, and can not be inlined. This is equivalent to declaring `(not usual-integrations)`.

-version
: Prints the version and some copyright information and exit the compiler.

-verbose
: Prints progress information to standard output during compilation.

The environment variable `CHICKEN_OPTIONS` can be set to a string with default command-line options for the compiler.

5 Runtime options

After successful compilation a C source file is generated and can be compiled with a C compiler. Executables generated with CHICKEN (and the compiler itself) accept a small set of runtime options:

`-:?`
: Shows a list of the available runtime options and exits the program.

`-:b`
: Enter a read-eval-print-loop when an error is encountered.

`-:c`
: Forces console mode. Currently this is only used in the interpreter (`csi`) to force output of the `#;N>` prompt even if stdin is not a terminal (for example if running in an `emacs` buffer under Windows).

`-:d`
: Prints some debug-information at runtime.

`-:D`
: Prints some more debug-information at runtime.

`-:hNUMBER`
: Specifies fixed heap size

`-:hiNUMBER`
: Specifies the initial heap size

`-:hgPERCENTAGE`
: Sets the growth rate of the heap in percent. If the heap is exhausted, then it will grow by `PERCENTAGE`. The default is 200.

`-:hmNUMBER`
: Specifies a maximal heap size. The default is (2GB - 15).

`-:hsPERCENTAGE`
: Sets the shrink rate of the heap in percent. If no more than a quarter of `PERCENTAGE` of the heap is used, then it will shrink to `PERCENTAGE`. The default is 50. Note: If you want to make sure that the heap never shrinks, specify a value of `0`. (this can be useful in situations where an optimal heap-size is known in advance).

`-:o`
: Disables detection of stack overflows at run-time.

`-:sNUMBER`
: Specifies stack size.

`-:tNUMBER`
: Specifies symbol table size.

`-:fNUMBER`
: Specifies the maximal number of currently pending finalizers before finalization is forced.

`-:aNUMBER`
: Specifies the length of the buffer for recording a trace of the last invoked procedures. Defaults to 16.

`-:w`
: Enables garbage collection of unused symbols. By default unused and unbound symbols are not garbage collected.

`-:r`
: Writes trace output to stderr. This option has no effect with in files compiled with the `-no-trace` options.

`-:x`
: Raises uncaught exceptions of separately spawned threads in primordial thread. By default uncaught exceptions in separate threads are not handled, unless the primordial one explicitly joins them. When warnings are enabled (the default) and `-:x` is not given, a warning will be shown, though.

`-:B`
: Sounds a bell (ASCII 7) on every major garbage collection.

The argument values may be given in bytes, in kilobytes (suffixed with `K` or `k`), in megabytes (suffixed with `M` or `m`), or in gigabytes (suffixed with `G` or `g`). Runtime options may be combined, like `-:dc`, but everything following a `NUMBER` argument is ignored. So `-:wh64m` is OK, but `-:h64mw` will not enable GC of unused symbols.

6 Examples

6.1 A simple example (with one source file)

To compile a Scheme program (assuming a UNIX-like environment) consisting of a single source file, perform the following steps.

6.1.1 Writing your source file

In this example we will assume your source file is called `foo.scm`:

```
;;; foo.scm

(define (fac n)
  (if (zero? n)
      1
      (* n (fac (- n 1))) ) )

(write (fac 10))
(newline)
```

6.1.2 Compiling your program

Compile the file `foo.scm`:

```
% csc foo.scm
```

This will produce the `foo` executable:

```
% ls
foo  foo.scm
```

6.1.3 Running your program

To run your newly compiled executable use:

```
% foo
3628800
```

If you get a `foo: command not found` error, you might want to try with `./foo` instead (or, in Unix machines, modify your `PATH` environment variable to include your current directory).

6.2 An example with multiple files

If multiple bodies of Scheme code are to be combined into a single executable, then we have to compile each file and link the resulting object files together with the runtime system.

Lets consider an example where your program consists of multiple source files.

6.2.1 Writing your source files

The declarations in these files specify which of the compiled files is the main module, and which is the library module. An executable can only have one main module, since a program has only a single entry-point. In this case `foo.scm` is the main module, because it doesn't have a `unit` declaration:

```
;;; foo.scm

; The declaration marks this source file as dependant on the symbols provided
; by the bar unit:
(declare (uses bar))

(write (fac 10)) (newline)
```

`bar.scm` will be our library:

```
;;; bar.scm

; The declaration marks this source file as the bar unit.  The names of the
; units and your files don't need to match.
(declare (unit bar))

(define (fac n)
  (if (zero? n)
      1
      (* n (fac (- n 1))) ) )
```

6.2.2 Compiling and running your program

You should compile your two files with the following commands:

```
% csc -c bar.scm
% csc -c foo.scm
```

That should produce two files, `bar.o` and `foo.o`. They contain the code from your source files in compiled form.

To link your compiled files use the following command:

```
% csc foo.o bar.o -o foo
```

This should produce the `foo` executable, which you can run just as in the previous example. At this point you can also erase the `*.o` files.

You could avoid one step and link the two files just as `foo.scm` is compiled:

```
% csc -c bar.scm
% csc foo.scm bar.o -o foo
```

Note that if you want to distribute your program, you might want it to follow the GNU Coding Standards. One relatively easy way to achieve this is to use Autoconf and Automake, two tools made for this specific purpose. Consult http://galinha.ucpel.tche.br/autoconf - automake for some instructions on how you would do this.

7 Extending the compiler

The compiler supplies a couple of hooks to add user-level passes to the compilation process. Before compilation commences any Scheme source files or compiled code specified using the `-extend` option are loaded and evaluated. The parameters `user-options-pass, user-read-pass, user-preprocessor-pass, user-pass, user-pass-2` and `user-post-analysis-pass` can be set to procedures that are called to perform certain compilation passes instead of the usual processing (for more information about parameters see: Supported language.

[parameter] user-options-pass
: Holds a procedure that will be called with a list of command-line arguments and should return two values: the source filename and the actual list of options, where compiler switches have their leading `-` (hyphen) removed and are converted to symbols. Note that this parameter is invoked **before** processing of the `-extend` option, and so can only be changed in compiled user passes.

[parameter] user-read-pass
: Holds a procedure of three arguments. The first argument is a list of strings with the code passed to the compiler via `-prelude` options. The second argument is a list of source files including any files specified by `-prologue` and `-epilogue`. The third argument is a list of strings specified using `-postlude` options. The procedure should return a list of toplevel Scheme expressions.

[parameter] user-preprocessor-pass
: Holds a procedure of one argument. This procedure is applied to each toplevel expression in the source file **before** macro-expansion. The result is macro-expanded and compiled in place of the original expression.

[parameter] user-pass
: Holds a procedure of one argument. This procedure is applied to each toplevel expression **after** macro-expansion. The result of the procedure is then compiled in place of the original expression.

[parameter] user-pass-2
: Holds a procedure of three arguments, which is called with the canonicalized node-graph as its sole argument. The result is ignored, so this pass has to mutate the node-structure to cause any effect.

[parameter] user-post-analysis-pass
: Holds a procedure that will be called after the last performed program analysis. The procedure (when defined) will be called with three arguments: the program database, a getter and a setter-procedure which can be used to access and manipulate the program database, which holds various information about the compiled program. The getter procedure should be called with two arguments: a symbol representing the binding for which information should be retrieved, and a symbol that specifies the database-entry. The current value of the database entry will be returned or `#f`, if no such entry is available. The setter procedure is called with three arguments: the symbol and key and the new value. For information about the contents of the program database contact the author.

Loaded code (via the `-extend` option) has access to the library units `extras, srfi-1, srfi-4, utils, regex` and the pattern matching macros. Multithreading is not available.

Note that the macroexpansion/canonicalization phase of the compiler adds certain forms to the source program. These extra expressions are not seen by `user-preprocessor-pass` but by `user-pass`.

8 Distributing compiled C files

It is relatively easy to create distributions of Scheme projects that have been compiled to C. The runtime system of CHICKEN consists of only two handcoded C files (`runtime.c` and `chicken.h`), plus the file `chicken-config.h`, which is generated by the build process. All other modules of the runtime system and the extension libraries are just compiled Scheme code. The following example shows a minimal application, which should run without changes on the most frequent operating systems, like Windows, Linux or FreeBSD:

Let's take a simple "Hello, world!":

```
; hello.scm

(print "Hello, world!")
```

Compiled to C, we get `hello.c`. We need the files `chicken.h` and `runtime.c`, which contain the basic runtime system, plus the three basic library files `library.c`, `eval.c` and `extras.c` which contain the same functionality as the library linked into a plain CHICKEN-compiled application, or which is available by default in the interpreter, `csi`:

```
% csc hello.scm -O2 -d1
```

A simple makefile is needed as well:

```
# Makefile for UNIX systems

hello: hello.o runtime.o library.o eval.o extras.o
       $(CC) -o hello hello.o runtime.o library.o eval.o extras.o -lm

hello.o: chicken.h
runtime.o: chicken.h
library.o: chicken.h
eval.o: chicken.h
extras.o: chicken.h
```

Now we have all files together, and can create an tarball containing all the files:

```
% tar cf hello.tar Makefile hello.c runtime.c library.c eval.c extras.c chicken
% gzip hello.tar
```

This is of naturally rather simplistic. Things like enabling dynamic loading, estimating the optimal stack-size and selecting supported features of the host system would need more configuration- and build-time support. All this can be addressed using more elaborate build-scripts, makefiles or by using http://galinha.ucpel.tche.br/autoconf - automake.

Note also that the size of the application can still be reduced by removing `extras` and `eval` and compiling `hello.scm` with the `-explicit-use` option.

For more information, study the CHICKEN source code and/or get in contact with the author.

Previous: index.html

Next: Using the interpreter

9 Using the interpreter

CHICKEN provides an interpreter named `csi` for evaluating Scheme programs and expressions interactively.

9.1 Interpreter command line format

```
csi {FILENAME|OPTION}
```

where `FILENAME` specifies a file with Scheme source-code. If the extension of the source file is `.scm`, it may be omitted. The runtime options described in Compiler command line format are also available for the interpreter. If the environment variable `CSI_OPTIONS` is set to a list of options, then these options are additionally passed to every direct or indirect invocation of `csi`. Please note that runtime options (like `-:...`) can not be passed using this method. The options recognized by the interpreter are:

–
: Ignore everything on the command-line following this marker. Runtime options (`-:...`) are still recognized.

-i -case-insensitive
: Enables the reader to read symbols case insensitive. The default is to read case sensitive (in violation of R5RS). This option registers the `case-insensitive` feature identifier.

-b -batch
: Quit the interpreter after processing all command line options.

-e -eval EXPRESSIONS
: Evaluate `EXPRESSIONS`. This option implies `-batch` and `-quiet`, so no startup message will be printed and the interpreter exits after processing all `-eval` options and/or loading files given on the command-line.

-D -feature SYMBOL
: Registers `SYMBOL` to be a valid feature identifier for `cond-expand`.

-h -help
: Write a summary of the available command line options to standard output and exit.

-I -include-path PATHNAME
: Specifies an alternative search-path for files included via the `include` special form. This option may be given multiple times. If the environment variable `CHICKEN_INCLUDE_PATH` is set, it should contain a list of alternative include pathnames separated by `;`. The environment variable `CHICKEN_HOME` is also considered as a search path.

-k -keyword-style STYLE
: Enables alternative keyword syntax, where `STYLE` may be either `prefix` (as in Common Lisp) or `suffix` (as in DSSSL). Any other value is ignored.

-n -no-init
: Do not load initialization-file. If this option is not given and the file `./.csirc` or `$HOME/.csirc` exists, then it is loaded before the read-eval-print loop commences.

-w -no-warnings
: Disables any warnings that might be issued by the reader or evaluated code.

-q -quiet
: Do not print a startup message.

-s -script PATHNAME
: This is equivalent to `-batch -quiet -no-init PATHNAME`. Arguments following `PATHNAME` are available by using `command-line-arguments` and are not processed as interpreter options. Extra options in the environment variable `CSI_OPTIONS` are ignored.

-ss PATHNAME
: The same as `-s PATHNAME` but invokes the procedure `main` with the value of `(command-line-arguments)` as its single argument. If the main procedure returns an integer

result, then the interpreter is terminated, returning the integer as the status code back to the invoking process. Any other result terminates the interpreter with a zero exit status.

-R -require-extension NAME
: Equivalent to evaluating `(require-extension NAME)`.

-v -version
: Write the banner with version information to standard output and exit.

9.2 Writing Scheme scripts

Since UNIX shells use the `#!` notation for starting scripts, anything following the characters `#!` is ignored, with the exception of the special symbols `#!optional, #!key, #!rest` and `#!eof`.

The easiest way is to use the `-script` option like this:

```
% cat foo
#! /usr/local/bin/csi -script
(print (eval (with-input-from-string
                (car (command-line-arguments))
                 read)))

% chmod +x foo
% foo "(+ 3 4)"
7
```

The parameter `command-line-arguments` is set to a list of the parameters that were passed to the Scheme script. Scripts can be compiled to standalone executables (don't forget to declare used library units).

CHICKEN supports writing shell scripts in Scheme for these platforms as well, using a slightly different approach. The first example would look like this on Windows:

```
C:>type foo.bat
@;csibatch %0 %1 %2 %3 %4 %5 %6 %7 %8 %9
(print (eval (with-input-from-string
                (car (command-line-arguments))
                read)))

C:>foo "(+ 3 4)"
7
```

Like UNIX scripts, batch files can be compiled. Windows batch scripts do not accept more than 8 arguments.

Since it is sometimes useful to run a script into the interpreter without actually running it (for example to test specific parts of it), the option `-ss` can be used as an alternative to `-script`. `-ss PATHNAME` is equivalent to `-script PATHNAME` but invokes `(main (command-line-arguments))` after loading all top-level forms of the script file. The result of `main` is returned as the exit status to the shell. Any non-numeric result exits with status zero:

```
% cat hi.scm
(define (main args)
  (print "Hi, " (car args))
  0)
% csi -ss hi.scm you
Hi, you
```

```
% csi -q
#;1> ,l hi.scm
#;2> (main (list "ye all"))
Hi, ye all
0
#;3>
```

9.3 Toplevel commands

The toplevel loop understands a number of special commands:

,?
: Show summary of available toplevel commands.

,l FILENAME ...
: Load files with given `FILENAME`s

,ln FILENAME ...
: Load files and print result(s) of each top-level expression.

,p EXP
: Pretty-print evaluated expression `EXP`.

,d EXP
: Describe result of evaluated expression `EXP`.

,du EXP
: Dump contents of the result of evaluated expression `EXP`.

,dur EXP N
: Dump `N` bytes of the result of evaluated expression `EXP`.

,exn
: Describes the last exception that occurred and adds it to the result history (it can be accessed using the # notation).

,q
: Quit the interpreter.

,r
: Show system information.

,s TEXT ...
: Execute shell-command.

,t EXP
: Evaluate form and print elapsed time.

,x EXP
: Pretty-print macroexpanded expression `EXP` (the expression is not evaluated).

,tr SYMBOL ...
: Enables tracing of the toplevel procedures with the given names.

```
#;1> (fac 10)                       ==> 3628800
#;2> ,tr fac
#;3> (fac 3)
|(fac 3)
| (fac 2)
|  (fac 1)
|   (fac 0)
|   fac -> 1
|  fac -> 1
| fac -> 2
|fac -> 6                          ==> 6
#;4> ,utr fac
```

```
#;5> (fac 3)                        ==> 6
```

,utr SYMBOL ...
: Disables tracing of the given toplevel procedures.

,br SYMBOL ...
: Sets a breakpoint at the procedures named `SYMBOL ...`. Breakpoint can also be trigged using the `breakpoint` procedure.

,ubr SYMBOL ...
: Removes breakpoints.

,c
: Continues execution from the last invoked breakpoint.

,breakall
: Enable breakpoints for all threads (this is the default).

,breakonly THREAD
: Enable breakpoints only for the thread returned by the expression `THREAD`.

,info
: Lists traced procedures and breakpoints.

,step EXPR
: Evaluates `EXPR` in single-stepping mode. On each procedure call you will be presented with a menu that allows stepping to the next call, leaving single-stepping mode or triggering a breakpoint. Note that you will see some internal calls, and unsafe or heavily optimized compiled code might not be stepped at all. Single-stepping mode is also possible by invoking the `singlestep` procedure.

You can define your own toplevel commands using the `toplevel-command` procedure:

9.4 toplevel-command

```
[procedure] (toplevel-command SYMBOL PROC [HELPSTRING])
```

Defines or redefines a toplevel interpreter command which can be invoked by entering `,SYMBOL`. `PROC` will be invoked when the command is entered and may read any required argument via `read` (or `read-line`). If the optional argument `HELPSTRING` is given, it will be listed by the `,?` command.

9.5 History access

The interpreter toplevel accepts the special object `#[INDEX]` which returns the result of entry number `INDEX` in the history list. If the expression for that entry resulted in multiple values, the first result (or an unspecified value for no values) is returned. If no `INDEX` is given (and if a whitespace or closing paranthesis character follows the `#`, then the result of the last expression is returned. Note that the value returned is implicitly quoted.

9.6 set-describer!

```
[procedure] (set-describer! TAG PROC)
```

Sets a custom description handler that invokes `PROC` when the `,d` command is invoked with a record-type object that has the type `TAG` (a symbol). `PROC` is called with two arguments: the object to be described and an output-port. It should write a possibly useful textual description of the object to the passed output-port. For example:

```
#;1> (define-record point x y)
#;2> (set-describer! 'point (lambda (pt o) (print "a point with x=" (point-x pt
#;3> ,d (make-point 1 2)
a point with x=1 and y=2
```

9.7 Auto-completion and edition

On platforms that support it, it is possible to get auto-completion of symbols, history (over different `csi` sessions) and a more feature-full editor for the expressions you type using the http://galinha.ucpel.tche.br/readline egg by Tony Garnock Jones. It is very useful for interactive use of csi.

To enable it install the egg and put this in your `~/.csirc` file:

```
(use readline regex)
(current-input-port (make-gnu-readline-port))
(gnu-history-install-file-manager (string-append (or (getenv "HOME") ".") "/.cs
```

More details are available in the egg's documentation.

9.8 Accessing documentation

You can access the manual directly from `csi` using the http://galinha.ucpel.tche.br/man extension by http://galinha.ucpel.tche.br/Mario Domenech Goulart.

To enable it install the egg and put this in your `~/.csirc` file:

```
(use man)
(man:load)
```

Then, in `csi`, you can search for definitions using `man:search` as in:

```
(man:search "case")
```

Note that the search uses regular expressions. To view the documentation for one entry from the manual, use `man:help` as in:

```
(man:help "case-lambda")
```

Previous: Using the compiler

Next: Supported language

10 Supported language

- Deviations from the standard
- Extensions to the standard
- Non standard read syntax
- Non standard macros and special forms
- Pattern matching
- Declarations
- Parameters
- Unit library basic Scheme definitions
- Unit eval evaluation and macro-handling
- Unit extras useful utility definitions
- Unit srfi-1 List Library
- Unit srfi-4 Homogeneous numeric vectors
- Unit srfi-13 String library
- Unit srfi-14 character set library
- Unit match pattern matching runtime-support
- Unit regex regular expressions
- Unit srfi-18 multithreading
- Unit posix Unix-like services
- Unit utils Shell scripting and file operations
- Unit tcp basic TCP-sockets
- Unit lolevel low-level operations

Previous: Using the interpreter

Next: Interface to external functions and variables

11 Deviations from the standard

- Identifiers are by default case-sensitive (see Compiler command line format).
- [4.1.3] The maximal number of arguments that may be passed to a compiled procedure or macro is 120. A macro-definition that has a single rest-parameter can have any number of arguments. If the `libffi` library is available on this platform, and if it is installed, then CHICKEN can take advantage of this. See the README file for more details.
- [4.2.2] `letrec` does evaluate the initial values for the bound variables sequentially and not in parallel, that is:

```
 (letrec ((x 1) (y 2)) (cons x y))
```

is equivalent to

```
 (let ((x (void)) (y (void)))
   (set! x 1)
   (set! y 2)
   (cons x y) )
```

where R5RS requires

```
 (let ((x (void)) (y (void)))
   (let ((tmp1 1) (tmp2 2))
     (set! x tmp1)
     (set! y tmp2)
     (cons x y) ) )
```

- [4.3] `syntax-rules` macros are not provided but available separately.
- [6.2.4] The runtime system uses the numerical string-conversion routines of the underlying C library and so does only understand standard (C-library) syntax for floating-point constants.
- [6.2.5] There is no built-in support for rationals, complex numbers or extended-precision integers (bignums). The routines `complex?`, `real?` and `rational?` are identical to the standard procedure `number?`. The procedures `numerator`, `denominator`, `rationalize`, `make-rectangular` and `make-polar` are not implemented. Support for extended numbers is available as a separate package, provided the GNU multiprecision library is installed.
- [6.2.6] The procedure `string->number` does not obey read/write invariance on inexact numbers.
- [6.4] The maximum number of values that can be passed to continuations captured using `call-with-current-continuation` is 120.
- [6.5] Code evaluated in `scheme-report-environment` or `null-environment` still sees non-standard syntax.
- [6.6.2] The procedure `char-ready?` always returns `#t` for terminal ports. The procedure `read` does not obey read/write invariance on inexact numbers.
- [6.6.3] The procedures `write` and `display` do not obey read/write invariance to inexact numbers.
- [6.6.4] The `transcript-on` and `transcript-off` procedures are not implemented.

Previous: Supported language

Next: Extensions to the standard

12 Extensions to the standard

[2.1] Identifiers may contain special characters if delimited with `| ... |`.

[2.3] The brackets `[ ... ]` are provided as an alternative syntax for `( ... )`. A number of reader extensions is provided. See Non standard read syntax.

[4] Numerous non-standard macros are provided. See Non standard macros and special forms for more information.

[4.1.4] Extended DSSSL style lambda lists are supported. DSSSL formal argument lists are defined by the following grammar:

```
<formal-argument-list> ==> <required-formal-argument>*
                           [(#!optional <optional-formal-argument>*)]
                           [(#!rest <rest-formal-argument>)]
                           [(#!key <key-formal-argument>*)]
<required-formal-argument> ==> <ident>
<optional-formal-argument> ==> <ident>
                            | (<ident> <initializer>)
<rest-formal-argument> ==> <ident>
<key-formal-argument> ==> <ident>
                         | (<ident> <initializer>)
<initializer> ==> <expr>
```

When a procedure is applied to a list of actual arguments, the formal and actual arguments are processed from left to right as follows:

- Variables in required-formal-arguments are bound to successive actual arguments starting with the first actual argument. It shall be an error if there are fewer actual arguments than required-formal-arguments.
- Next, variables in optional-formal-arguments are bound to any remaining actual arguments. If there are fewer remaining actual arguments than optional-formal-arguments, then variables are bound to the result of the evaluation of initializer if one was specified or otherwise to `#f`. The initializer is evaluated in an environment in which all previous formal arguments have been bound.
- If there is a rest-formal-argument, then it is bound to a list of all remaining actual arguments. The remaining actual arguments are also eligible to be bound to keyword-formal-arguments. If there is no rest-formal-argument and there are no keywords, the it shall be an error if there are any remaining actual arguments.
- If `#!key` was specified in the formal-argument-list, there shall be an even number of remaining actual arguments. These are interpreted as a series of pairs, where the first member of each pair is a keyword specifying the argument name, and th corresponding value. It shall be an error if the first member of a pair is not a keyword. It shall be an error if the argument name is not the same as a variable in a keyword-formal-argument, unless there is a rest-formal-argument. If the same argument name occurs more than once in the list of actual arguments, then the first value is used. If there is no actual argument for a particular keyword-formal-argument, then the variable is bound to the result of evaluating initializer if one was specified or `#f`. The initializer is evaluated in an environment in which all previous formal arguments have been bound.

It shall be an error for an `<ident>` to appear more than once in a formal-argument-list.

Example:

```
((lambda x x) 3 4 5 6)       => (3 4 5 6)
((lambda (x y #!rest z) z)
 3 4 5 6)                    => (5 6)
((lambda (x y #!optional z #!rest r #!key i (j 1))
    (list x y z i: i j: j))
 3 4 5 i: 6 i: 7)            => (3 4 5 i: 6 j: 1)
```

[4.1.6] `set!` for unbound toplevel variables is allowed. `set! (PROCEDURE ...) ...)` is supported, as CHICKEN implements SRFI-17. [4.2.1] The `cond` form supports SRFI-61.

[4.2.2] It is allowed for initialization values of bindings in a `letrec` construct to refer to previous variables in the same set of bindings, so

```
(letrec ((foo 123)
         (bar foo) )
  bar)
```

is allowed and returns `123`.

[4.2.3] `(begin)` is allowed in non-toplevel contexts and evaluates to an unspecified value.

[4.2.5] Delayed expressions may return multiple values.

[5.2.2] CHICKEN extends standard semantics by allowing internal definitions everywhere, and not only at the beginning of a body. A set of internal definitions is equivalent to a `letrec` form enclosing all following expressions in the body:

```
(let ((foo 123))
  (bar)
  (define foo 456)
  (baz foo) )
```

expands into

```
(let ((foo 123))
  (bar)
  (letrec ((foo 456))
    (baz foo) ) )
```

[5.2] `define` with a single argument is allowed and initializes the toplevel or local binding to an unspecified value. CHICKEN supports *curried* definitions, where the the variable name may also be a list specifying a name and a nested lambda list. So

```
(define ((make-adder x) y) (+ x y))
```

is equivalent to

```
(define (make-adder x) (lambda (y) (+ x y)))
```

[6] CHICKEN provides numerous non-standard procedures. See the manual sections on library units for more information.

[6.2.4] The special IEEE floating-point numbers *+nan*, *+inf* and *-inf* are supported, as is negative zero.

[6.3.4] User defined character names are supported. See `char-name`. Characters can be given in hexadecimal notation using the *#\xXX* syntax where *XX* specifies the character code. Character codes above

255 are supported and can be read (and are written) using the *#\uXXXX* and *#\UXXXXXXXX* notations.

Non-standard characters names supported are `#\tab`, `#\linefeed`, `#\return`, `#\alarm`, `#\vtab`, `#\nul`, `#\page`, `#\esc`, `#\delete` and `#\backspace`.

[6.3.5] CHICKEN supports special characters preceded with a backslash \ in quoted string constants. *\n* denotes the newline-character, *\r* carriage return, *\b* backspace, *\t* TAB, *\v* vertical TAB, *\a* alarm, *\f* formfeed, *\xXX* a character with the code `XX` in hex and *\uXXXX* (and *\UXXXXXXXX*) a unicode character with the code `XXXX`. The latter is encoded in UTF-8 format.

The third argument to `substring` is optional and defaults to the length of the string.

[6.4] `force` called with an argument that is not a promise returns that object unchanged. Captured continuations can be safely invoked inside before- and after-thunks of a `dynamic-wind` form and execute in the outer dynamic context of the `dynamic-wind` form.

Implicit non-multival continuations accept multiple values by discarding all but the first result. Zero values result in the continuation receiving an unspecified value. Note that this slight relaxation of the behaviour of returning mulitple values to non-multival continuations does not apply to explicit continuations (created with `call-with-current-continuation`).

[6.5] The second argument to `eval` is optional and defaults to the value of `(interaction-environment)`. `scheme-report-environment` and `null-environment` accept an optional 2nd parameter: if not `#f` (which is the default), toplevel bindings to standard procedures are mutable and new toplevel bindings may be introduced.

[6.6] The *tilde* character (`~`) is automatically expanded in pathnames. Additionally, if a pathname starts with `$VARIABLE...`, then the prefix is replaced by the value of the given environment variable.

[6.6.1] if the procedures `current-input-port` and `current-output-port` are called with an argument (which should be a port), then that argument is selected as the new current input- and output-port, respectively. The procedures `open-input-file`, `open-output-file`, `with-input-from-file`, `with-output-to-file`, `call-with-input-file` and `call-with-output-file` accept an optional second (or third) argument which should be one or more keywords, if supplied. These arguments specify the mode in which the file is opened. Possible values are the keywords `#:text`, `#:binary` or `#:append`.

Previous: Deviations from the standard

Next: Non standard read syntax

13 Non standard read syntax

13.1 Multiline Block Comment

```
#| ... |#
```

A multiline *block* comment. May be nested. Implements SRFI-30

13.2 Expression Comment

```
#;EXPRESSION
```

Treats `EXPRESSION` as a comment.

13.3 External Representation

```
#,(CONSTRUCTORNAME DATUM ...)
```

Allows user-defined extension of external representations. (For more information see the documentation for SRFI-10)

13.4 Syntax Expression

```
#'EXPRESSION
```

An abbreviation for `(syntax EXPRESSION)`.

13.5 Location Expression

```
#$EXPRESSION
```

An abbreviation for `(location EXPRESSION)`.

13.6 Keyword

```
#:SYMBOL
```

Syntax for keywords. Keywords are symbols that evaluate to themselves, and as such don't have to be quoted.

13.7 Multiline String Constant

```
#<<TAG
```

Specifies a multiline string constant. Anything up to a line equal to `TAG` (or end of file) will be returned as a single string:

```
(define msg #<<END
 "Hello, world!", she said.
END
)
```

is equivalent to

```
(define msg "\"Hello, world!\", she said.")
```

13.8 Multiline String Constant with Embedded Expressions

```
#<#TAG
```

Similar to `#<<`, but allows substitution of embedded Scheme expressions prefixed with `#` and optionally enclosed in curly brackets. Two consecutive `#`s are translated to a single `#`:

```
(define three 3)
(display #<#EOF
This is a simple string with an embedded `##' character
and substituted expressions: (+ three 99) ==> #(+ three 99)
(three is "#{three}")
EOF
)
```

prints

```
This is a simple string with an embedded `#' character
and substituted expressions: (+ three 99) ==> 102
(three is "3")
```

13.9 Foreign Declare

```
#> ... <#
```

Abbreviation for `foreign-declare " ... ")`.

13.10 Sharp Prefixed Symbol

```
#%...
```

Reads like a normal symbol.

13.11 Bang

```
#!...
```

Interpretation depends on the directly following characters. Only the following are recognized. Any other case results in a read error.

13.11.1 Line Comment

- If followed by whitespace or a slash, then everything up the end of the current line is ignored

13.11.2 Eof Object

- If followed by the character sequence `eof`, then the (self-evaluating) end-of-file object is returned

13.11.3 DSSSL Formal Parameter List Annotation

- If followed by any of the character sequences `optional`, `rest` or `key`, then a symbol with the same name (and prefixed with `#!`) is returned

13.11.4 Read Mark Invocation

- If a *read mark* with the same name as the token is registered, then its procedure is called and the result of the read-mark procedure will be returned

13.12 Case Sensitive Expression

```
#cs...
```

Read the next expression in case-sensitive mode (regardless of the current global setting).

13.13 Case Insensitive Expression

```
#ci...
```

Read the next expression in case-insensitive mode (regardless of the current global setting).

13.14 Conditional Expansion

```
#+FEATURE EXPR
```

Equivalent to

```
(cond-expand (FEATURE EXPR) (else))
```

Previous: Extensions to the standard

Next: Non standard macros and special forms

14 Non-standard macros and special forms

14.1 Making extra libraries and extensions available

14.1.1 require-extension

```
[syntax] (require-extension ID ...)
[syntax] (use ID ...)
```

This form does all necessary steps to make the libraries or extensions given in `ID ...` available. It loads syntactic extension, if needed and generates code for loading/linking with core library modules or separately installed extensions. `use` is just a shorter alias for `require-extension`. This implementation of `require-extension` is compliant to SRFI-55 (see the SRFI-55 document for more information).

During interpretation/evaluation `require-extension` performs one of the following:

- If `ID` names a built in features `chicken srfi-23 srfi-30 srfi-39 srfi-8 srfi-6 srfi-2 srfi-0 srfi-10 srfi-9 srfi-17 srfi-55`, then nothing is done.
- If `ID` names one of syntactic extensions `chicken-more-macros chicken-ffi-macros`, then this extension will be loaded.
- `ID` names one of the core library units shipped with CHICKEN, then a `(load-library 'ID)` will be performed.
- `ID` names an installed extension with the `syntax` or `require-at-runtime` attribute, then the equivalent of `(require-for-syntax 'ID)` is performed, probably followed by `(require ...)` for any run-time requirements.
- Otherwise `(require-extension ID)` is equivalent to `(require 'ID)`.

During compilation one of the following happens instead:

- If `ID` names a built in features `chicken srfi-23 srfi-30 srfi-39 srfi-8 srfi-6 srfi-2 srfi-0 srfi-10 srfi-9 srfi-17 srfi-55`, then nothing is done.
- If `ID` names one of syntactic extensions `chicken-more-macros chicken-ffi-macros`, then this extension will be loaded at compile-time, making the syntactic extensions available in compiled code.
- If `ID` names one of the core library units shipped with CHICKEN, or if the option `-uses ID` has been passed to the compiler then a `(declare (uses ID))` is generated.
- If `ID` names an installed extension with the `syntax` or `require-at-runtime` attribute, then the equivalent of `(require-for-syntax 'ID)` is performed, and code is emitted to `(require ...)` any needed run-time requirements.
- Otherwise `(require-extension ID)` is equivalent to `(require 'ID)`.

To make long matters short - just use `require-extension` and it will normally figure everything out for dynamically loadable extensions and core library units.

`ID` should be a pure extension name and should not contain any path prefixes (for example `dir/lib...`) is illegal).

`ID` may also be a list that designates an extension-specifier. Currently the following extension specifiers are defined:

- `(srfi NUMBER ...)` is required for SRFI-55 compatibility and is fully implemented
- `(version ID NUMBER)` is equivalent to `ID`, but checks at compile-time whether the extension named `ID` is installed and whether its version is equal or higher than `NUMBER`. `NUMBER` may be a string or a number, the comparison is done lexicographically (using `string>=?`).

See also: `set-extension-specifier!`

14.1.2 define-extension

```
[syntax] (define-extension NAME CLAUSE ...)
```

This macro simplifies the task of writing extensions that can be linked both statically and dynamically. If encountered in interpreted code or code that is compiled into a shared object (specifically if compiled with the feature `chicken-compile-shared`, done automatically by `csc` when compiling with the `-shared` or `-dynamic` option) then the code given by clauses of the form

```
(dynamic EXPRESSION ...)
```

are inserted into the output as a `begin` form.

If compiled statically (specifically if the feature `chicken-compiled-shared` has not been given), then this form expands into the following:

```
(declare (unit NAME))
(provide 'NAME)
```

and all clauses of the form

```
(static EXPRESSION ...)
```

all additionally inserted into the expansion.

As a convenience, the clause

```
(export IDENTIFIER ...)
```

is also allowed and is identical to `(declare (export IDENTIFIER ...))` (unless the `define-extension` form occurs in interpreted code, in with it is simply ignored).

Note that the compiler option `-extension NAME` is equivalent to prefixing the compiled file with

```
(define-extension NAME)
```

14.2 Binding forms for optional arguments

14.2.1 :optional

```
[syntax] (:optional ARGS DEFAULT)
```

Use this form for procedures that take a single optional argument. If ARGS is the empty list DEFAULT is evaluated and returned, otherwise the first element of the list ARGS. It is an error if ARGS contains more than one value.

```
(define (incr x . i) (+ x (:optional i 1)))
(incr 10)                                   ==> 11
(incr 12 5)                                 ==> 17
```

14.2.2 case-lambda

```
[syntax] (case-lambda (LAMBDA-LIST1 EXP1 ...) ...)
```

Expands into a lambda that invokes the body following the first matching lambda-list.

```
(define plus
  (case-lambda
    (() 0)
    ((x) x)
    ((x y) (+ x y))
    ((x y z) (+ (+ x y) z))
    (args (apply + args))))

(plus)                      ==> 9
(plus 1)                    ==> 1
(plus 1 2 3)                ==> 6
```

For more information see the documentation for SRFI-16

14.2.3 let-optionals

```
[syntax]  (let-optionals ARGS ((VAR1 DEFAULT1) ...) BODY ...)
```

Binding constructs for optional procedure arguments. ARGS should be a rest-parameter taken from a lambda-list. `let-optionals` binds `VAR1 ...` to available arguments in parallel, or to `DEFAULT1 ...` if not enough arguments were provided. `let-optionals*` binds `VAR1 ...` sequentially, so every variable sees the previous ones. it is an error if any excess arguments are provided.

```
(let-optionals '(one two) ((a 1) (b 2) (c 3))
  (list a b c) )                                 ==> (one two 3)
```

14.2.4 let-optionals*

```
[syntax]  (let-optionals* ARGS ((VAR1 DEFAULT1) ... [RESTVAR]) BODY ...)
```

Binding constructs for optional procedure arguments. `ARGS` should be a rest-parameter taken from a lambda-list. `let-optionals` binds `VAR1 ...` to available arguments in parallel, or to `DEFAULT1 ...` if not enough arguments were provided. `let-optionals*` binds `VAR1 ...` sequentially, so every variable sees the previous ones. If a single variable `RESTVAR` is given, then it is bound to any remaining arguments, otherwise it is an error if any excess arguments are provided.

```
(let-optionals* '(one two) ((a 1) (b 2) (c a))
  (list a b c) )                                 ==> (one two one)
```

14.3 Other binding forms

14.3.1 and-let*

```
[syntax] (and-let* (BINDING ...) EXP1 EXP2 ...)
```

SRFI-2. Bind sequentially and execute body. `BINDING` can be a list of a variable and an expression, a list with a single expression, or a single variable. If the value of an expression bound to a variable is `#f`, the `and-let*` form evaluates to `#f` (and the subsequent bindings and the body are not executed). Otherwise the next binding is performed. If all bindings/expressions evaluate to a true result, the body is executed normally and the result of the last expression is the result of the `and-let*` form. See also the documentation for SRFI-2.

14.3.2 rec

```
[syntax] (rec NAME EXPRESSION)
[syntax] (rec (NAME VARIABLE ...) BODY ...)
```

Allows simple definition of recursive definitions. `(rec NAME EXPRESSION)` is equivalent to `(letrec ((NAME EXPRESSION)) NAME)` and `(rec (NAME VARIABLE ...) BODY ...)` is the same as `(letrec ((NAME (lambda (VARIABLE ...) BODY ...))) NAME)`.

14.3.3 cut

```
[syntax] (cut SLOT ...)
[syntax] (cute SLOT ...)
```

Syntactic sugar for specializing parameters.

14.3.4 define-values

```
[syntax] (define-values (NAME ...) EXP)
```

Defines several variables at once, with the result values of expression `EXP`.

14.3.5 fluid-let

```
[syntax] (fluid-let ((VAR1 X1) ...) BODY ...)
```

Binds the variables `VAR1 ...` dynamically to the values `X1 ...` during execution of `BODY ...`.

14.3.6 let-values

```
[syntax] (let-values (((NAME ...) EXP) ...) BODY ...)
```

Binds multiple variables to the result values of `EXP ...`. All variables are bound simultaneously.

14.3.7 let*-values

```
[syntax] (let*-values (((NAME ...) EXP) ...) BODY ...)
```

Binds multiple variables to the result values of `EXP ...`. The variables are bound sequentially.

```
(let*-values (((a b) (values 2 3))
              ((p) (+ a b)) )
  p)                              ==> 5
```

14.3.8 letrec-values

```
[syntax] (letrec-values (((NAME ...) EXP) ...) BODY ...)
```

Binds the result values of `EXP ...` to multiple variables at once. All variables are mutually recursive.

```
(letrec-values (((odd even)
                   (values
                     (lambda (n) (if (zero? n) #f (even (sub1 n))))
                     (lambda (n) (if (zero? n) #t (odd (sub1 n)))) ) ) )
  (odd 17) )                          ==> #t
```

14.3.9 parameterize

```
[syntax] (parameterize ((PARAMETER1 X1) ...) BODY ...)
```

Binds the parameters `PARAMETER1 ...` dynamically to the values `X1 ...` during execution of `BODY ...`. (see also: `make-parameter` in @ref{Parameters}). Note that `PARAMETER` may be any expression that evaluates to a parameter procedure.

14.3.10 receive

```
[syntax] (receive (NAME1 ... [. NAMEn]) VALUEEXP BODY ...)
[syntax] (receive VALUEEXP)
```

SRFI-8. Syntactic sugar for `call-with-values`. Binds variables to the result values of `VALUEEXP` and evaluates `BODY ...`.

The syntax

```
(receive VALUEEXP)
```

is equivalent to

```
(receive _ VALUEEXP _)
```

14.3.11 set!-values

```
[syntax] (set!-values (NAME ...) EXP)
```

Assigns the result values of expression `EXP` to multiple variables.

14.4 Substitution forms and macros

14.4.1 define-constant

```
[syntax] (define-constant NAME CONST)
```

Define a variable with a constant value, evaluated at compile-time. Any reference to such a constant should appear textually **after** its definition. This construct is equivalent to `define` when evaluated or interpreted. Constant definitions should only appear at toplevel. Note that constants are local to the current compilation unit and are not available outside of the source file in which they are defined. Names of constants still exist in the Scheme namespace and can be lexically shadowed. If the value is mutable, then the compiler is careful to preserve its identity. `CONST` may be any constant expression, and may also refer to constants defined via `define-constant` previously. This for should only be used at top-level.

14.4.2 define-inline

```
[syntax] (define-inline (NAME VAR ... [. VAR]) BODY ...)
[syntax] (define-inline NAME EXP)
```

Defines an inline procedure. Any occurrence of `NAME` will be replaced by `EXP` or `(lambda (VAR ... [. VAR]) BODY ...)`. This is similar to a macro, but variable-names and -scope will be correctly handled. Inline substitutions take place **after** macro-expansion. `EXP` should be a lambda-expression. Any reference to `NAME` should appear textually **after** its definition. Note that inline procedures are local to the current compilation unit and are not available outside of the source file in which they are defined. Names of inline procedures still exist in the Scheme namespace and can be lexically shadowed. This construct is equivalent to `define` when evaluated or interpreted. Inline definitions should only appear at toplevel.

14.4.3 define-macro

```
[syntax] (define-macro (NAME VAR ... [. VAR]) EXP1 ...)
[syntax] (define-macro NAME (lambda (VAR ... [. VAR]) EXP1 ...))
[syntax] (define-macro NAME1 NAME2)
```

Define a globally visible macro special form. The macro is available as soon as it is defined, i.e. it is registered at compile-time. If the file containing this definition invokes `eval` and the declaration `run-time-macros` (or the command line option `-run-time-macros`) has been used, then the macro is visible in evaluated expressions during runtime. The second possible syntax for `define-macro` is allowed for portability purposes only. In this case the second argument **must** be a lambda-expression or a macro name. Only global macros can be defined using this form. `(define-macro NAME1 NAME2)` simply copies the macro definition from `NAME2` to `NAME1`, creating an alias.

Extended lambda list syntax (`#!optional`, etc.) can be used but note that arguments are source expressions and thus default values for optional or keyword arguments should take this into consideration.

14.4.4 define-for-syntax

```
[syntax] (define-for-syntax (NAME VAR ... [. VAR]) EXP1 ...)
[syntax] (define-for-syntax NAME [VALUE])
```

Defines the toplevel variable `NAME` at macro-expansion time. This can be helpful when you want to define support procedures for use in macro-transformers, for example.

14.5 Conditional forms

14.5.1 switch

```
[syntax] (switch EXP (KEY EXP1 ...) ... [(else EXPn ...)])
```

This is similar to `case`, but a) only a single key is allowed, and b) the key is evaluated.

14.5.2 unless

```
[syntax] (unless TEST EXP1 EXP2 ...)
```

Equivalent to:

```
(if (not TEST) (begin EXP1 EXP2 ...))
```

14.5.3 when

```
[syntax] (when TEST EXP1 EXP2 ...)
```

Equivalent to:

```
(if TEST (begin EXP1 EXP2 ...))
```

14.6 Record structures

14.6.1 define-record

```
[syntax] (define-record NAME SLOTNAME ...)
```

Defines a record type. Call `make-NAME` to create an instance of the structure (with one initialization-argument for each slot). `(NAME? STRUCT)` tests any object for being an instance of this structure. Slots are accessed via `(NAME-SLOTNAME STRUCT)` and updated using `(NAME-SLOTNAME-set! STRUCT VALUE)`.

```
(define-record point x y)
(define p1 (make-point 123 456))
(point? p1)                      ==> #t
(point-x p1)                     ==> 123
(point-y-set! p1 99)
(point-y p1)                     ==> 99
```

14.6.2 define-record-printer

```
[syntax] (define-record-printer (NAME RECORDVAR PORTVAR) BODY ...)
[syntax] (define-record-printer NAME PROCEDURE)
```

Defines a printing method for record of the type `NAME` by associating a procedure with the record type. When a record of this type is written using `display`, `write` or `print`, then the procedure is called with two arguments: the record to be printed and an output-port.

```
(define-record foo x y z)
(define f (make-foo 1 2 3))
(define-record-printer (foo x out)
  (fprintf out "#,(foo ~S ~S ~S)"
           (foo-x x) (foo-y x) (foo-z x)) )
(define-reader-ctor 'foo make-foo)
(define s (with-output-to-string
              (lambda () (write f))))
s                                   ==> "#,(foo 1 2 3)"
(equal? f (with-input-from-string
              s read)))             ==> #t
```

`define-record-printer` works also with SRFI-9 record types.

14.6.3 define-record-type

```
[syntax] (define-record-type NAME (CONSTRUCTOR TAG ...) PREDICATE (FIELD ACCESS
```

SRFI-9 record types. For more information see the documentation for SRFI-9.

14.7 Other forms

14.7.1 assert

```
[syntax] (assert EXP [STRING ARG ...])
```

Signals an error if `EXP` evaluates to false. An optional message `STRING` and arguments `ARG ...` may be supplied to give a more informative error-message. If compiled in *unsafe* mode (either by specifying the `-unsafe` compiler option or by declaring `(unsafe)`), then this expression expands to an unspecified value. The result is the value of `EXP`.

14.7.2 cond-expand

```
[syntax] (cond-expand FEATURE-CLAUSE ...)
```

Expands by selecting feature clauses. This form is allowed to appear in non-toplevel expressions.

Predefined feature-identifiers are "situation" specific:

compile
: `eval`, `library`, `match`, `compiling`, `srfi-11`, `srfi-15`, `srfi-31`, `srfi-26`, `srfi-16`, `utils`, `regex`, `srfi-4`, `match`, `srfi-1`, `srfi-69`, `srfi-28`, `extras`, `srfi-8`, `srfi-6`, `srfi-2`, `srfi-0`, `srfi-10`, `srfi-9`, `srfi-55`, `srfi-61` `chicken`, `srfi-23`, `srfi-30`, `srfi-39`, `srfi-62`, `srfi-17`, `srfi-12`.

load
: `srfi-69`, `srfi-28`, `extras`, `srfi-8`, `srfi-6`, `srfi-2`, `srfi-0`, `srfi-10`, `srfi-9`, `srfi-55`, `srfi-61`, `chicken`, `srfi-23`, `srfi-30`, `srfi-39`, `srfi-62`, `srfi-17`, `srfi-12`. `library` is implicit.

eval
: `match`, `csi`, `srfi-11`, `srfi-15`, `srfi-31`, `srfi-26`, `srfi-16`, `srfi-69`, `srfi-28`, `extras`, `srfi-8`, `srfi-6`, `srfi-2`, `srfi-0`, `srfi-10`, `srfi-9`, `srfi-55`, `srfi-61`, `chicken`, `srfi-23`, `srfi-30`, `srfi-39`, `srfi-62`, `srfi-17`, `srfi-12`. `library` is implicit.

The following feature-identifiers are available in all situations: `(machine-byte-order)`, `(machine-type)`, `(software-type)`, `(software-version)`, where the actual feature-identifier is platform dependent.

In addition the following feature-identifiers may exist: `applyhook`, `extraslot`, `ptables`, `dload`, `libffi`.

For further information, see the documentation for SRFI-0.

14.7.3 ensure

`[syntax] (ensure PREDICATE EXP [ARGUMENTS ...])`

Evaluates the expression `EXP` and applies the one-argument procedure `PREDICATE` to the result. If the predicate returns `#f` an error is signaled, otherwise the result of `EXP` is returned. If compiled in *unsafe* mode (either by specifying the `-unsafe` compiler option or by declaring `(unsafe)`), then this expression expands to an unspecified value. If specified, the optional `ARGUMENTS` are used as arguments to the invocation of the error-signalling code, as in `(error ARGUMENTS ...)`. If no `ARGUMENTS` are given, a generic error message is displayed with the offending value and `PREDICATE` expression.

14.7.4 eval-when

`[syntax] (eval-when (SITUATION ...) EXP ...)`

Controls evaluation/compilation of subforms. `SITUATION` should be one of the symbols `eval`, `compile` or `load`. When encountered in the evaluator, and the situation specifier `eval` is not given, then this form is not evaluated and an unspecified value is returned. When encountered while compiling code, and the situation specifier `compile` is given, then this form is evaluated at compile-time. When encountered while compiling code, and the situation specifier `load` is not given, then this form is ignored and an expression resulting into an unspecified value is compiled instead.

The following table should make this clearer:

	In compiled code	**In interpreted code**
`eval`	ignore	evaluate
`compile`	evaluate at compile time	ignore
`load`	compile as normal	ignore

The situation specifiers `compile-time` and `run-time` are also defined and have the same meaning as `compile` and `load`, respectively.

14.7.5 include

`[syntax] (include STRING)`

Include toplevel-expressions from the given source file in the currently compiled/interpreted program. If the included file has the extension `.scm`, then it may be omitted. The file is searched in the current directory and, if not found, in all directories specified in the `-include-path` option.

14.7.6 nth-value

`[syntax] (nth-value N EXP)`

Returns the `N`th value (counting from zero) of the values returned by expression `EXP`.

14.7.7 time

`[syntax] (time EXP1 ...)`

Evaluates `EXP1 ...` and prints elapsed time and some values about GC use, like time spent in major GCs, number of minor and major GCs.

Previous: Non standard read syntax

Next: Pattern matching

15 Pattern Matching

(This description has been taken mostly from Andrew Wright's postscript document)

Pattern matching allows complicated control decisions based on data structure to be expressed in a concise manner. Pattern matching is found in several modern languages, notably Standard ML, Haskell and Miranda. These syntactic extensions internally use the `match` library unit.

The basic form of pattern matching expression is:

```
(match exp [pat body] ...)
```

where `exp` is an expression, `pat` is a pattern, and `body` is one or more expressions (like the body of a lambda-expression). The `match` form matches its first subexpression against a sequence of patterns, and branches to the `body` corresponding to the first pattern successfully matched. For example, the following code defines the usual `map` function:

```
(define map
  (lambda (f l)
    (match l
      [() '()]
      [(x . y) (cons (f x) (map f y))])))
```

The first pattern `()` matches the empty list. The second pattern `(x . y)` matches a pair, binding `x` to the first component of the pair and `y` to the second component of the pair.

15.1 Pattern Matching Expressions

The complete syntax of the pattern matching expressions follows:

```
exp ::= (match exp clause ...)
     |  (match-lambda clause ...)
     |  (match-lambda* clause ...)
     |  (match-let ([pat exp] ...) body)
     |  (match-let* ([pat exp] ...) body)
     |  (match-letrec ([pat exp] ...) body)
     |  (match-let var ([pat exp] ...) body)
     |  (match-define pat exp)

clause ::= [pat body]
        |  [pat (=> identifier) body]

pat ::= identifier           matches anything, and binds identifier as a variab
     |  _                    anything
     |  ()                   itself (the empty list)
     |  #t                   itself
     |  #f                   itself
     |  string               an `equal?' string
     |  number               an `equal?' number
     |  character            an `equal?' character
     |  's-expression        an `equal?' s-expression
     |  (pat-1 ... pat-n)    a proper list of n elements
     |  (pat-1 ... pat-n . pat-n+1)
```

```
                             a list of n or more elements
     |  (pat-1 ... pat-n pat-n+1 ..k)
                             a proper list of n+k or more elements [1]
     |  #(pat-1 ... pat-n)   a vector of n elements
     |  #(pat-1 ... pat-n pat-n+1 ..k)
                             a vector of n+k or more elements
     |  ($ struct pat-1 ... pat-n)
                             a structure
     |  (= field pat)        a field of a structure
     |  (and pat-1 ... pat-n)
                             if all of pat-1 through pat-n match
     |  (or pat-1 ... pat-n)
                             if any of pat-1 through pat-n match
     |  (not pat-1 ... pat-n)
                             if none of pat-1 through pat-n match
     |  (? predicate pat-1 ... pat-n)
                             if predicate true and pat-1 through pat-n all matc
     |  (set! identifier)    anything, and binds identifier as a setter
     |  (get! identifier)    anything, and binds identifier as a getter
     |  `qp                  a quasipattern

qp ::= ()                    itself (the empty list)
    |  #t                    itself
    |  #f                    itself
    |  string                an `equal?' string
    |  number                an `equal?' number
    |  character             an `equal?' character
    |  symbol                an `equal?' symbol
    |  (qp-1 ... qp-n)       a proper list of n elements
    |  (qp-1 ... qp-n . qp-n+1)
                             a list of n or more elements
    |  (qp-1 ... qp-n qp-n+1 ..k)
                             a proper list of n+k or more elements
    |  #(qp-1 ... qp-n)      a vector of n elements
    |  #(qp-1 ... qp-n qp-n+1 ..k)
                             a vector of n+k or more elements
    |  ,pat                  a pattern
    |  ,@pat                 a pattern, spliced
```

The notation `..k` denotes a keyword consisting of three consecutive dots (ie., `...`), or two dots and an non-negative integer (eg., *`..1`*, *`..2`*), or three consecutive underscores (ie., `___`), or two underscores and a non-negative integer. The keywords *`..k`* and `__` *`k`* are equivalent. The keywords `...`, `___`, *`..0`*, and `__`*`0`* are equivalent.

The next subsection describes the various patterns.

The `match-lambda` and `match-lambda*` forms are convenient combinations of `match` and `lambda`, and can be explained as follows:

```
(match-lambda [pat body] ...)   =  (lambda (x) (match x [pat body] ...))
(match-lambda* [pat body] ...)  =  (lambda x (match x [pat body] ...))
```

where `x` is a unique variable. The `match-lambda` form is convenient when defining a single argument function that immediately destructures its argument. The `match-lambda*` form constructs a function that accepts any number of arguments; the patterns of `match-lambda*` should be lists.

The `match-let`, `match-let*`, `match-letrec`, and `match-define` forms generalize Scheme's `let`, `let*`, `letrec`, and `define` expressions to allow patterns in the binding position rather than just variables. For example, the following expression:

```
(match-let ([(x y z) (list 1 2 3)]) body ...)
```

binds `x` to 1, `y` to 2, and `z` to 3 in `body ...`. These forms are convenient for destructuring the result of a function that returns multiple values as a list or vector. As usual for `letrec` and `define`, pattern variables bound by `match-letrec` and `match-define` should not be used in computing the bound value.

The `match`, `match-lambda`, and `match-lambda*` forms allow the optional syntax `(=> identifier)` between the pattern and the body of a clause. When the pattern match for such a clause succeeds, the `identifier` is bound to a `failure procedure' of zero arguments within the `body`. If this procedure is invoked, it jumps back to the pattern matching expression, and resumes the matching process as if the pattern had failed to match. The `body` must not mutate the object being matched, otherwise unpredictable behavior may result.

15.2 Patterns

`identifier`: (excluding the reserved names `?`, `,`, `=`, `_`, `and`, `or`, `not`, `set!`, `get!`, `...`, and `..k` for non-negative integers `k`) matches anything, and binds a variable of this name to the matching value in the `body`.

_: matches anything, without binding any variables.

`()`, `#t`, `#f`, `string`, `number`, `character`, `'s-expression`: These constant patterns match themselves, i.e., the corresponding value must be `equal?` to the pattern.

`(pat-1 ... pat-n)`: matches a proper list of `n` elements that match `pat-1` through `pat-n`.

`(pat-1 ... pat-n . pat-n+1)`: matches a (possibly improper) list of at least `n` elements that ends in something matching `pat-n+1`.

`(pat-1 ... pat-n pat-n+1 ...)`: matches a proper list of `n` or more elements, where each element of the tail matches `pat-n+1`. Each pattern variable in `pat-n+1` is bound to a list of the matching values. For example, the expression:

```
(match '(let ([x 1][y 2]) z)
  [('let ((binding values) ...) exp)  body])
```

binds `binding` to the list `'(x y)`, `values` to the list \`'(1 2)`, and `exp` to `'z` in the body of the `match`-expression. For the special case where `pat-n+1` is a pattern variable, the list bound to that variable may share with the matched value.

`(pat-1 ... pat-n pat-n+1 ___)`: This pattern means the same thing as the previous pattern.

`(pat-1 ... pat-n pat-n+1 ..k)`: This pattern is similar to the previous pattern, but the tail must be at least `k` elements long. The pattern keywords `..0` and `...` are equivalent.

`(pat-1 ... pat-n ~ pat-n+1 __k)`: This pattern means the same thing as the previous pattern.

`#(pat-1 ... pat-n)`: matches a vector of length `n`, whose elements match `pat-1` through `pat-n`.

`#(pat-1 ... pat-n pat-n+1 ...)`: matches a vector of length `n` or more, where each element beyond `n` matches `pat-n+1`.

`#(pat-1 ... pat-n pat-n+1 ..k)`: matches a vector of length `n+k` or more, where each element beyond `n` matches `pat-n+1`.

`($ struct pat-1 ... pat-n)`: matches a structure declared with `define-record` or `define-record-type`.

`(= field pat)`: is intended for selecting a field from a structure. *field* may be any expression; it is applied to the value being matched, and the result of this application is matched against `pat`.

`(and pat-1 ... pat-n)`: matches if all of the subpatterns match. At least one subpattern must be present. This pattern is often used as `(and x pat)` to bind `x` to to the entire value that matches `pat` (cf. *as-patterns* in ML or Haskell).

`(or pat-1 ... pat-n)`: matches if any of the subpatterns match. At least one subpattern must be present. All subpatterns must bind the same set of pattern variables.

`(not pat-1 ... pat-n)`: matches if none of the subpatterns match. At least one subpattern must be present. The subpatterns may not bind any pattern variables.

`(? predicate pat-1 ... pat-n)`: In this pattern, `predicate` must be an expression evaluating to a single argument function. This pattern matches if `predicate` applied to the corresponding value is true, and the subpatterns `pat-1 ... pat-n` all match. The `predicate` should not have side effects, as the code generated by the pattern matcher may invoke predicates repeatedly in any order. The `predicate` expression is bound in the same scope as the match expression, i.e., free variables in `predicate` are not bound by pattern variables.

`(set! identifier)`: matches anything, and binds `identifier` to a procedure of one argument that mutates the corresponding field of the matching value. This pattern must be nested within a pair, vector, box, or structure pattern. For example, the expression:

```
(define x (list 1 (list 2 3)))
(match x [(_ (_ (set! setit)))  (setit 4)])
```

mutates the `cadadr` of `x` to 4, so that `x` is `'(1 (2 4))`.

`(get! identifier)`: matches anything, and binds `identifier` to a procedure of zero arguments that accesses the corresponding field of the matching value. This pattern is the complement to `set!`. As with `set!`, this pattern must be nested within a pair, vector, box, or structure pattern.

Quasipatterns: Quasiquote introduces a quasipattern, in which identifiers are considered to be symbolic constants. Like Scheme's quasiquote for data, `unquote` (,) and `unquote-splicing` (,@) escape back to normal patterns.

15.3 Match Failure

If no clause matches the value, the default action is to invoke the procedure `(match-error-procedure)` with the value that did not match. The default definition of `(match-error-procedure)` calls `error` with an appropriate message:

```
#;1> (match 1 (2 2))

Failed match:
Error: no matching clause for : 1
```

For most situations, this behavior is adequate, but it can be changed by altering the value of the parameter `match-error-control`:

{procedure} match-error-control

```
(match-error-control [MODE])
```

Selects a mode that specifies how `match...` macro forms are to be expanded. With no argument this procedure returns the current mode. A single argument specifies the new mode that decides what should happen if no match-clause applies. The following modes are supported:

Signal an error. This is the default.
1. :error

Signal an error and output the offending form.
1. :match

Omits `pair?` tests when the consequence is to fail in `car` or `cdr` rather than to signal an error.
1. :fail

unspecified Non-matching expressions will either fail in `car` or `cdr` or return an unspecified value. This mode applies to files compiled with the `unsafe` option or declaration.

When an error is signalled, the raised exception will be of kind `(exn match)`.

[procedure] match-error-procedure

```
(match-error-procedure [PROCEDURE])
```

Sets or returns the procedure called upon a match error. The procedure takes one argument, the value which failed to match. When the error control mode is `#:match` a second argument, the source form of the match expression is available.

15.4 Record Structures Pattern

The `$` pattern handles native record structures and SRFI-9 records transparently. Currently it is required that SRFI-9 record predicates are named exactly like the record type name, followed by a `?` (question mark) character.

15.5 Code Generation

Pattern matching macros are compiled into `if`-expressions that decompose the value being matched with standard Scheme procedures, and test the components with standard predicates. Rebinding or lexically shadowing the names of any of these procedures will change the semantics of the `match` macros. The names that should not be rebound or shadowed are:

```
null? pair? number? string? symbol? boolean? char? procedure? vector? list?
```

```
equal?
car cdr cadr cdddr ...
vector-length vector-ref
reverse length call/cc
```

Additionally, the code generated to match a structure pattern like `($ Foo pat-1 ... pat-n)` refers to the name `Foo?`. This name also should not be shadowed.

Previous: Non standard macros and special forms

Next: Declarations

16 Declarations

16.1 declare

```
[syntax] (declare DECLSPEC ...)
```

Process declaration specifiers. Declarations always override any command-line settings. Declarations are valid for the whole compilation-unit (source file), the position of the declaration in the source file can be arbitrary. Declarations are ignored in the interpreter but not in code evaluated at compile-time (by `eval-when` or in syntax extensions loaded via `require-extension` or `require-for-syntax`. `DECLSPEC` may be any of the following:

16.2 always-bound

```
[declaration specifier] (always-bound SYMBOL ...)
```

Declares that the given variables are always bound and accesses to those have not to be checked.

16.3 block

```
[declaration specifier] (block)
```

Assume global variables are never redefined. This is the same as specifying the `-block` option.

16.4 block-global

16.5 hide

```
[declaration specifier] (block-global SYMBOL ...)
[declaration specifier] (hide SYMBOL ...)
```

Declares that the toplevel bindings for `SYMBOL ...` should not be accessible from code in other compilation units or by `eval`. Access to toplevel bindings declared as block global is also more efficient.

16.6 bound-to-procedure

```
[declaration specifier] (bound-to-procedure SYMBOL ...)
```

Declares that the given identifiers are always bound to procedure values.

16.7 c-options

```
[declaration specifier] (c-options STRING ...)
```

Declares additional C/C++ compiler options that are to be passed to the subsequent compilation pass that translates C to machine code. This declaration will only work if the source file is compiled with the `csc` compiler driver.

16.8 check-c-syntax

```
[declaration specifier] (check-c-syntax)
[declaration specifier] (not check-c-syntax)
```

Enables or disables syntax-checking of embedded C/C++ code fragments. Checking C syntax is the default.

16.9 compress-literals

```
[declaration specifier] (compress-literals [THRESHOLD [INITIALIZER]])
```

The same as the `-compress-literals` compiler option. The threshold argument defaults to 50. If the optional argument `INITIALIZER` is given, then the literals will not be created at module startup, but when the procedure with this name will be called.

16.10 constant

```
[declaration specifier] (constant SYMBOL ...)
```

Declares the procedures with the names `SYMBOL ...` as constant, that is, as not having any side effects. This can help the compiler to remove non-side-effecting expressions.

16.11 export

```
[declaration specifier] (export SYMBOL ...)
```

The opposite of `hide`. All given identifiers will be exported and all toplevel variables not listed will be hidden and not be accessible outside of this compilation unit.

16.12 emit-exports

```
[declaration specifier] (emit-exports STRING)
```

Write exported toplevel variables to file with name `STRING`.

16.13 emit-external-prototypes-first

```
[declaration specifier] (emit-external-prototypes-first)
```

Emit prototypes for callbacks defined with `define-external` before any other foreign declarations. Equivalent to giving the `-emit-external-prototypes-first` option to the compiler.

16.14 disable-interrupts

```
[declaration specifier] (disable-interrupts)
[declaration specifier] (not interrupts-enabled)
```

Disable timer-interrupts checks in the compiled program. Threads can not be preempted in main- or library-units that contain this declaration.

16.15 disable-warning

```
[declaration specifier] (disable-warning CLASS ...)
```

Disable warnings of type `CLASS ...` (equivalent to the `-disable-warning CLASS` compiler option).

16.16 import

```
[declaration specifier] (import SYMBOL-OR-STRING ...)
```

Adds new imports to the list of externally available toplevel variables. Arguments to this declaration may be either strings (designating `.exports` files, without the file-extension) or symbols which directly designate imported variables.

16.17 inline

```
[declaration specifier] (inline)
[declaration specifier] (not inline)
[declaration specifier] (inline IDENTIFIER ...)
[declaration specifier] (not inline IDENTIFIER ...)
```

If given without an identifier-list, inlining of known procedures is enabled (this is equivalent to the `-inline` compiler option). When an identifier-list is given, then inlining is enabled only for the specified global procedures. The negated forms `(not inline)` and `(not inline IDENTIFIER)` disable global inlining, or inlining for the given global procedures only, respectively.

16.18 inline-limit

```
[declaration specifier] (inline-limit THRESHOLD)
```

Sets the maximum size of procedures which may potentially be inlined. The default threshold is `10`.

16.19 interrupts-enabled

```
[declaration specifier] (interrupts-enabled)
```

Enable timer-interrupts checks in the compiled program (the default).

16.20 keep-shadowed-macros

```
[declaration specifier] (keep-shadowed-macros)
```

Normally, when a toplevel variable is assigned or defined that has the same name as a macro, the macro-definition will be removed (in addition to showing a warning). This declaration will disable the removal of the macro.

16.21 lambda-lift

```
[declaration specifier] (lambda-lift)
```

Enables lambda-lifting (equivalent to the `-lambda-lift` option).

16.22 link-options

```
[declaration specifier] (link-options STRING ...)
```

Declares additional linker compiler options that are to be passed to the subsequent compilation pass that links the generated code into an executable or library. This declaration will only work if the source file is compiled with the `csc` compiler driver.

16.23 no-argc-checks

```
[declaration specifier] (no-argc-checks)
```

Disables argument count checking.

16.24 no-bound-checks

```
[declaration specifier] (no-bound-checks)
```

Disables the bound-checking of toplevel bindings.

16.25 no-procedure-checks

```
[declaration specifier] (no-procedure-checks)
```

Disables checking of values in operator position for being of procedure type.

16.26 post-process

```
[declaration specifier] (post-process STRING ...)
```

Arranges for the shell commands `STRING ...` to be invoked after the current file has been translated to C. Any occurrences of the substring `$@@` in the strings given for this declaration will be replaced by the pathname of the currently compiled file, without the file-extension. This declaration will only work if the source file is compiled with the `csc` compiler driver.

16.27 number-type

16.28 fixnum-arithmetic

```
[declaration specifier] ([number-type] TYPE)
[declaration specifier] (fixnum-arithmetic)
```

Declares that only numbers of the given type are used. `TYPE` may be `fixnum` or `generic` (which is the default).

16.29 run-time-macros

```
[declaration specifier] (run-time-macros)
```

Equivalent to the compiler option of the same name - macros defined in the compiled code are also made available at runtime.

16.30 standard-bindings

```
[declaration specifier] (standard-bindings SYMBOL ...)
[declaration specifier] (not standard-bindings SYMBOL ...)
```

Declares that all given standard procedures (or all if no symbols are specified) are never globally redefined. If `not` is specified, then all but the given standard bindings are assumed to be never redefined.

16.31 extended-bindings

```
[declaration specifier] (extended-bindings SYMBOL ...)
[declaration specifier] (not extended-bindings SYMBOL ...)
```

Declares that all given non-standard and CHICKEN-specific procedures

```
[declaration specifier] (or all if no symbols are specified) are never globally
```

If `not` is specified, then all but the given extended bindings are assumed to be never redefined.

16.32 usual-integrations

```
[declaration specifier] (usual-integrations SYMBOL ...)
[declaration specifier] (not usual-integrations SYMBOL ...)
```

Declares that all given standard and extended bindings (or all if no symbols are specified) are never globally redefined. If `not` is specified, then all but the given standard and extended bindings are assumed to be never redefined. Note that this is the default behaviour, unless the `-no-usual-integrations` option has been given.

16.33 unit

```
[declaration specifier] (unit SYMBOL)
```

Specify compilation unit-name (if this is a library)

16.34 unsafe

```
[declaration specifier] (unsafe)
[declaration specifier] (not safe)
```

Do not generate safety-checks. This is the same as specifying the `-unsafe` option. Also implies

```
(declare (no-bound-checks) (no-procedure-checks) (no-argc-checks))
```

16.35 uses

```
[declaration specifier] (uses SYMBOL ...)
```

Gives a list of used library-units. Before the toplevel-expressions of the main-module are executed, all used units evaluate their toplevel-expressions in the order in which they appear in this declaration. If a library unit A uses another unit B, then B's toplevel expressions are evaluated before A's. Furthermore, the used symbols are registered as features during compile-time, so `cond-expand` knows about them.

Previous: Pattern matching

Next: Parameters

17 Parameters

Certain behavior of the interpreter and compiled programs can be customized via 'parameters', where a parameter is a procedure of zero or one arguments. To retrieve the value of a parameter call the parameter-procedure with zero arguments. To change the setting of the parameter, call the parameter-procedure with the new value as argument:

```
(define foo (make-parameter 123))
(foo)                             ==> 123
(foo 99)
(foo)                             ==> 99
```

Parameters are fully thread-local, each thread of execution owns a local copy of a parameters' value.

CHICKEN implements SRFI-39.

17.1 make-parameter

```
[procedure] (make-parameter VALUE [GUARD])
```

Returns a procedure that accepts zero or one argument. Invoking the procedure with zero arguments returns `VALUE`. Invoking the procedure with one argument changes its value to the value of that argument (subsequent invocations with zero parameters return the new value). `GUARD` should be a procedure of a single argument. Any new values of the parameter (even the initial value) are passed to this procedure. The guard procedure should check the value and/or convert it to an appropriate form.

17.2 case-sensitive

If true, then `read` reads symbols and identifiers in case-sensitive mode and uppercase characters in symbols are printed escaped. Defaults to `#t`.

17.3 dynamic-load-libraries

A list of strings containing shared libraries that should be checked for explicitly loaded library units (this facility is not available on all platforms). See `load-library`.

17.4 command-line-arguments

Contains the list of arguments passed to this program, with the name of the program and any runtime options (all options starting with `-:`) removed.

17.5 current-read-table

A read-table object that holds read-procedures for special non-standard read-syntax (see `set-read-syntax!` for more information).

17.6 exit-handler

A procedure of a single optional argument. When `exit` is called, then this procedure will be invoked with the exit-code as argument. The default behavior is to terminate the program.

17.7 eval-handler

A procedure of one or two arguments. When `eval` is invoked, it calls the value of this parameter with the same arguments. The default behavior is to evaluate the argument expression and to ignore the second parameter.

17.8 force-finalizers

If true, force and execute all pending finalizers before exiting the program (either explicitly by `exit` or implicitly when the last toplevel expression has been executed). Default is `#t`.

17.9 implicit-exit-handler

A procedure of no arguments. When the last toplevel expression of the program has executed, then the value of this parameter is called. The default behaviour is to invoke all pending finalizers.

17.10 keyword-style

Enables alternative keyword syntax, where `STYLE` may be either `#:prefix` (as in Common Lisp) or `#:suffix` (as in DSSSL). Any other value disables the alternative syntaxes.

17.11 load-verbose

A boolean indicating whether loading of source files, compiled code (if available) and compiled libraries should display a message.

17.12 repl-prompt

A procedure that should evaluate to a string that will be printed before reading interactive input from the user in a read-eval-print loop. Defaults to `(lambda () "#;N> ")`.

17.13 reset-handler

A procedure of zero arguments that is called via `reset`. The default behavior in compiled code is to invoke the value of `(exit-handler)`. The default behavior in the interpreter is to abort the current computation and to restart the read-eval-print loop.

Previous: Declarations

Next: Unit library

18 Unit library

This unit contains basic Scheme definitions. This unit is used by default, unless the program is compiled with the `-explicit-use` option.

18.1 Arithmetic

18.1.1 add1

18.1.2 sub1

```
[procedure] (add1 N)
[procedure] (sub1 N)
```

Adds/subtracts 1 from `N`.

18.1.3 bitwise-and

18.1.4 bitwise-ior

18.1.5 bitwise-xor

18.1.6 bitwise-not

18.1.7 arithmetic-shift

```
[procedure] (bitwise-and N1 ...)
[procedure] (bitwise-ior N1 ...)
[procedure] (bitwise-xor N1 ...)
[procedure] (bitwise-not N)
[procedure] (arithmetic-shift N1 N2)
```

Binary integer operations. `arithmetic-shift` shifts the argument `N1` by `N2` bits to the left. If `N2` is negative, than `N1` is shifted to the right. These operations only accept exact integers or inexact integers in word range (32 bit signed on 32-bit platforms, or 64 bit signed on 64-bit platforms).

18.1.8 bit-set?

```
[procedure] (bit-set? N INDEX)
```

Returns `#t` if the bit at the position `INDEX` in the integer `N` is set, or `#f` otherwise. The rightmost/least-significant bit is bit 0.

18.1.9 fixnum?

```
[procedure] (fixnum? X)
```

Returns `#t` if `X` is a fixnum, or `#f` otherwise.

18.1.10 fx+

18.1.11 fx-

18.1.12 fx*

18.1.13 fx/

18.1.14 fxmod

18.1.15 fxneg

18.1.16 fxmin

18.1.17 fxmax

18.1.18 fx=

18.1.19 fx>

18.1.20 fx<

18.1.21 fx>=

18.1.22 fx<=

18.1.23 fxand

18.1.24 fxior

18.1.25 fxxor

18.1.26 fxnot

18.1.27 fxshl

18.1.28 fxshr

```
[procedure] (fx+ N1 N2)
[procedure] (fx- N1 N2)
[procedure] (fx* N1 N2)
[procedure] (fx/ N1 N2)
[procedure] (fxmod N1 N2)
[procedure] (fxneg N)
[procedure] (fxmin N1 N2)
[procedure] (fxmax N1 N2)
[procedure] (fx= N1 N2)
[procedure] (fx> N1 N2)
[procedure] (fx< N1 N2)
[procedure] (fx>= N1 N2)
[procedure] (fx<= N1 N2)
[procedure] (fxand N1 N2)
[procedure] (fxior N1 N2)
[procedure] (fxxor N1 N2)
[procedure] (fxnot N)
```

```
[procedure] (fxshl N1 N2)
[procedure] (fxshr N1 N2)
```

Arithmetic fixnum operations. These procedures do not check their arguments, so non-fixnum parameters will result in incorrect results. `fxneg` negates its argument.

On division by zero, `fx/` and `fxmod` signal a condition of kind `(exn arithmetic)`.

`fxshl` and `fxshr` perform arithmetic shift left and right, respectively.

18.1.29 flonum?

18.1.30 fp+

18.1.31 fp-

18.1.32 fp*

18.1.33 fp/

18.1.34 fpneg

18.1.35 fpmin

18.1.36 fpmax

18.1.37 fp=

18.1.38 fp>

18.1.39 fp<

18.1.40 fp>=

18.1.41 fp<=

```
[procedure] (flonum? X)
[procedure] (fp+ X Y)
[procedure] (fp- X Y)
[procedure] (fp* X Y)
[procedure] (fp/ X Y)
[procedure] (fpneg X)
[procedure] (fpmin X Y)
[procedure] (fpmax X Y)
[procedure] (fp= X Y)
[procedure] (fp> X Y)
[procedure] (fp< X Y)
[procedure] (fp>= X Y)
[procedure] (fp<= X Y)
```

Arithmetic floating-point operations. In safe mode, these procedures throw a type error with non-float arguments (except `flonum?`, which returns `#f`). In unsafe mode, these procedures do not check their arguments. A non-flonum argument in unsafe mode can crash the system.

18.1.42 signum

```
[procedure] (signum N)
```

Returns `1` if `N` is positive, `-1` if `N` is negative or `0` if `N` is zero. `signum` is exactness preserving.

18.1.43 finite?

```
[procedure] (finite? N)
```

Returns `#f` if `N` is negative or positive infinity, and `#f` otherwise.

18.2 File Input/Output

18.2.1 current-output-port

```
[procedure] (current-output-port [PORT])
```

Returns default output port. If `PORT` is given, then that port is selected as the new current output port.

Note that the default output port is not buffered. Use `set-buffering-mode!` if you need a different behaviour.

18.2.2 current-error-port

```
[procedure] (current-error-port [PORT])
```

Returns default error output port. If `PORT` is given, then that port is selected as the new current error output port.

Note that the default error output port is not buffered. Use `set-buffering-mode!` if you need a different behaviour.

18.2.3 flush-output

```
[procedure] (flush-output [PORT])
```

Write buffered output to the given output-port. `PORT` defaults to the value of `(current-output-port)`.

18.2.4 port-name

```
[procedure] (port-name PORT)
```

Fetch filename from `PORT`. This returns the filename that was used to open this file. Returns a special tag string, enclosed into parentheses for non-file ports.

18.2.5 port-position

```
[procedure] (port-position PORT)
```

Returns the current position of `PORT` as two values: row and column number. If the port does not support such an operation an error is signaled. This procedure is currently only available for input ports.

18.2.6 set-port-name!

```
[procedure] (set-port-name! PORT STRING)
```

Sets the name of `PORT` to `STRING`.

18.3 Files

18.3.1 delete-file

`[procedure] (delete-file STRING)`

Deletes the file with the pathname `STRING`. If the file does not exist, an error is signaled.

18.3.2 file-exists?

`[procedure] (file-exists? STRING)`

Returns `STRING` if a file with the given pathname exists, or `#f` otherwise.

18.3.3 rename-file

`[procedure] (rename-file OLD NEW)`

Renames the file or directory with the pathname `OLD` to `NEW`. If the operation does not succeed, an error is signaled.

18.4 String ports

18.4.1 get-output-string

`[procedure] (get-output-string PORT)`

Returns accumulated output of a port created with `(open-output-string)`.

18.4.2 open-input-string

`[procedure] (open-input-string STRING)`

Returns a port for reading from `STRING`.

18.4.3 open-output-string

`[procedure] (open-output-string)`

Returns a port for accumulating output in a string.

18.5 Feature identifiers

CHICKEN maintains a global list of *features* naming functionality available int the current system. Additionally the `cond-expand` form accesses this feature list to infer what features are provided. Predefined features are `chicken`, and the SRFIs (Scheme Request For Implementation) provided by the base system: `srfi-23, srfi-30, srfi-39`. If the `eval` unit is used (the default), the features `srfi-0, srfi-2, srfi-6, srfi-8, srfi-9` and `srfi-10` are defined. When compiling code (during compile-time) the feature `compiling` is registered. When evaluating code in the interpreter (csi), the feature `csi` is registered.

18.5.1 features

```
[procedure] (features)
```

Returns a list of all registered features that will be accepted as valid feature-identifiers by `cond-expand`.

18.5.2 feature?

```
[procedure] (feature? ID ...)
```

Returns `#t` if all features with the given feature-identifiers `ID ...` are registered.

18.5.3 register-feature!

```
[procedure] (register-feature! FEATURE ...)
```

Register one or more features that will be accepted as valid feature-identifiers by `cond-expand`. `FEATURE ...` may be a keyword, string or symbol.

18.5.4 unregister-feature!

```
[procedure] (unregister-feature! FEATURE ...)
```

Unregisters the specified feature-identifiers. `FEATURE ...` may be a keyword, string or symbol.

18.6 Keywords

Keywords are special symbols prefixed with `#:` that evaluate to themselves. Procedures can use keywords to accept optional named parameters in addition to normal required parameters. Assignment to and bindings of keyword symbols is not allowed. The parameter `keyword-style` and the compiler/interpreter option `-keyword-style` can be used to allow an additional keyword syntax, either compatible to Common LISP, or to DSSSL.

18.6.1 get-keyword

```
[procedure] (get-keyword KEYWORD ARGLIST [THUNK])
```

Returns the argument from `ARGLIST` specified under the keyword `KEYWORD`. If the keyword is not found, then the zero-argument procedure `THUNK` is invoked and the result value is returned. If `THUNK` is not given, `#f` is returned.

```
(define (increase x . args)
  (+ x (get-keyword #:amount args (lambda () 1))) )
(increase 123)                                      ==> 124
(increase 123 #:amount 10)                          ==> 133
```

Note: the `KEYWORD` may actually be any kind of object.

18.6.2 keyword?

```
[procedure] (keyword? X)
```

Returns `#t` if `X` is a keyword symbol, or `#f` otherwise.

18.6.3 keyword->string

```
[procedure] (keyword->string KEYWORD)
```

Transforms `KEYWORD` into a string.

18.6.4 string->keyword

```
[procedure] (string->keyword STRING)
```

Returns a keyword with the name `STRING`.

18.7 Exceptions

CHICKEN implements the (currently withdrawn) SRFI-12 exception system. For more information, see the SRFI-12 document.

18.7.1 condition-case

```
[syntax] (condition-case EXPRESSION CLAUSE ...)
```

Evaluates `EXPRESSION` and handles any exceptions that are covered by `CLAUSE ...`, where `CLAUSE` should be of the following form:

```
CLAUSE = ([VARIABLE] (KIND ...) BODY ...)
```

If provided, `VARIABLE` will be bound to the signalled exception object. `BODY ...` is executed when the exception is a property- or composite condition with the kinds given `KIND ...` (unevaluated). If no clause applies, the exception is re-signalled in the same dynamic context as the `condition-case` form.

```
(define (check thunk)
  (condition-case (thunk)
    [(exn file) (print "file error")]
    [(exn) (print "other error")]
    [var () (print "something else")] ) )

(check (lambda () (open-input-file "")))   ; -> "file error"
(check (lambda () some-unbound-variable))  ; -> "othererror"
(check (lambda () (signal 99)))            ; -> "something else"

(condition-case some-unbound-variable
  [(exn file) (print "ignored)] ) ; -> signals error
```

18.7.2 breakpoint

```
[procedure] (breakpoint [NAME])
```

Programmatically triggers a breakpoint (similar to the `,br` top-level csi command).

All error-conditions signalled by the system are of kind `exn`. The following composite conditions are additionally defined:

`(exn arity)`

Signalled when a procedure is called with the wrong number of arguments.

`(exn type)`

Signalled on type-mismatch errors, for example when an argument of the wrong type is passed to a builtin procedure.

`(exn arithmetic)`

Signalled on arithmetic errors, like division by zero.

`(exn i/o)`

Signalled on input/output errors.

`(exn i/o file)`

Signalled on file-related errors.

`(exn i/o net)`

Signalled on network errors.

`(exn bounds)`

Signalled on errors caused by accessing non-existent elements of a collection.

`(exn runtime)`

Signalled on low-level runtime-system error-situations.

`(exn runtime limit)`

Signalled when an internal limit is exceeded (like running out of memory).

`(exn match)`

Signalled on errors raised by failed matches (see the section on `match`).

`(exn syntax)`

Signalled on syntax errors.

`(exn breakpoint)`

Signalled when a breakpoint is reached.

Notes:

- All error-exceptions (of the kind `exn`) are non-continuable.
- Error-exceptions of the `exn` kind have additional `arguments` and `location` properties that contain the arguments passed to the exception-handler and the name of the procedure where the error occurred (if available).
- When the `posix` unit is available and used, then a user-interrupt (`signal/int`) signals an exception of the kind `user-interrupt`.
- the procedure `condition-property-accessor` accepts an optional third argument. If the condition does not have a value for the desired property and if the optional argument is given, no error is signalled and the accessor returns the third argument.
- In composite conditionss all properties are currently collected in a single property-list, so in the case that to conditions have the same named property, only one will be visible.

18.8 Environment information and system interface

18.8.1 argv

`[procedure] (argv)`

Return a list of all supplied command-line arguments. The first item in the list is a string containing the name of the executing program. The other items are the arguments passed to the application. This list is freshly created on every invocation of `(argv)`. It depends on the host-shell whether arguments are expanded ('globbed') or not.

18.8.2 exit

`[procedure] (exit [CODE])`

Exit the running process and return exit-code, which defaults to 0 (Invokes `exit-handler`).

18.8.3 build-platform

```
[procedure] (build-platform)
```

Returns a symbol specifying the toolset which has been used for building the executing system, which is one of the following:

```
cygwin
msvc
mingw32
gnu
metrowerks
intel
watcom
unknown
```

18.8.4 chicken-version

```
[procedure] (chicken-version [FULL])
```

Returns a string containing the version number of the CHICKEN runtime system. If the optional argument `FULL` is given and true, then a full version string is returned.

18.8.5 errno

```
[procedure] (errno)
```

Returns the error code of the last system call.

18.8.6 getenv

```
[procedure] (getenv STRING)
```

Returns the value of the environment variable `STRING` or `#f` if that variable is not defined.

18.8.7 machine-byte-order

```
[procedure] (machine-byte-order)
```

Returns the symbol `little-endian` or `big-endian`, depending on the machine's byte-order.

18.8.8 machine-type

```
[procedure] (machine-type)
```

Returns a symbol specifying the processor on which this process is currently running, which is one of the following:

```
alpha
mips
hppa
ultrasparc
sparc
ppc
ia64
x86
x86-64
unknown
```

18.8.9 on-exit

```
[procedure] (on-exit THUNK)
```

Schedules the zero-argument procexdures `THUNK` to be executed before the process exits, either explicitly via `exit` or implicitly after exection of the last toplevel form. Note that finalizers for unreferenced finalized data are run before exit procedures.

18.8.10 software-type

```
[procedure] (software-type)
```

Returns a symbol specifying the operating system on which this process is currently running, which is one of the following:

```
windows
unix
macos
ecos
unknown
```

18.8.11 software-version

```
[procedure] (software-version)
```

Returns a symbol specifying the operating system version on which this process is currently running, which is one of the following:

```
linux
freebsd
netbsd
```

```
openbsd
macosx
hpux
solaris
sunos
unknown
```

18.8.12 c-runtime

`[procedure] (c-runtime)`

Returns a symbol that designates what kind of C runtime library has been linked with this version of the Chicken libraries. Possible return values are `static`, `dynamic` or `unknown`. On systems not compiled with the Microsoft C compiler, `c-runtime` always returns `unknown`.

18.8.13 system

`[procedure] (system STRING)`

Execute shell command. The functionality offered by this procedure depends on the capabilities of the host shell.

18.9 Execution time

18.9.1 cpu-time

`[procedure] (cpu-time)`

Returns the used CPU time of the current process in milliseconds as two values: the time spent in user code, and the time spent in system code. On platforms where user and system time can not be differentiated, system time will be always be 0.

18.9.2 current-milliseconds

`[procedure] (current-milliseconds)`

Returns the number of milliseconds since process- or machine startup.

18.9.3 current-seconds

`[procedure] (current-seconds)`

Returns the number of seconds since midnight, Jan. 1, 1970.

18.9.4 current-gc-milliseconds

`[procedure] (current-gc-milliseconds)`

Returns the number of milliseconds spent in major garbage collections since the last call of `current-gc-milliseconds` and returns an exact integer.

18.10 Interrupts and error-handling

18.10.1 enable-warnings

`[procedure] (enable-warnings [BOOL])`

Enables or disables warnings, depending on wether `BOOL` is true or false. If called with no arguments, this procedure returns `#t` if warnings are currently enabled, or `#f` otherwise. Note that this is not a parameter. The current state (wether warnings are enabled or disabled) is global and not thread-local.

18.10.2 error

`[procedure] (error [LOCATION] [STRING] EXP ...)`

Prints error message, writes all extra arguments to the value of `(current-error-port)` and invokes the current exception-handler. This conforms to SRFI-23. If `LOCATION` is given and a symbol, it specifies the *location* (the name of the procedure) where the error occurred.

18.10.3 get-call-chain

`[procedure] (get-call-chain [START [THREAD]])`

Returns a list with the call history. Backtrace information is only generated in code compiled without `-no-trace` and evaluated code. If the optional argument `START` is given, the backtrace starts at this offset, i.e. when `START` is 1, the next to last trace-entry is printed, and so on. If the optional argument `THREAD` is given, then the call-chain will only be constructed for calls performed by this thread.

18.10.4 print-call-chain

`[procedure] (print-call-chain [PORT [START [THREAD]]])`

Prints a backtrace of the procedure call history to `PORT`, which defaults to `(current-output-port)`.

18.10.5 print-error-message

```
[procedure] (print-error-message EXN [PORT [STRING]])
```

Prints an appropriate error message to `PORT` (which defaults to the value of `(current-output-port)` for the object `EXN`. `EXN` may be a condition, a string or any other object. If the optional argument `STRING` is given, it is printed before the error-message. `STRING` defaults to `"Error:"`.

18.10.6 procedure-information

```
[procedure] (procedure-information PROC)
```

Returns an s-expression with debug information for the procedure `PROC`, or `#f`, if `PROC` has no associated debug information.

18.10.7 reset

```
[procedure] (reset)
```

Reset program (Invokes `reset-handler`).

18.10.8 warning

```
[procedure] (warning STRING EXP ...)
```

Displays a warning message (if warnings are enabled with `enable-warnings`) and continues execution.

18.10.9 singlestep

```
[procedure] (singlestep THUNK)
```

Executes the code in the zero-procedure `THUNK` in single-stepping mode.

18.11 Garbage collection

18.11.1 gc

```
[procedure] (gc [FLAG])
```

Invokes a garbage-collection and returns the number of free bytes in the heap. The flag specifies whether a minor (`#f`) or major (`#t`) GC is to be triggered. If no argument is given, `#t` is assumed. When the argument is `#t`, all pending finalizers are executed.

18.11.2 memory-statistics

```
[procedure] (memory-statistics)
```

Performs a major garbage collection and returns a three element vector containing the total heap size in bytes, the number of bytes currently used and the size of the nursery (the first heap generation). Note that the actual heap is actually twice the size given in the heap size, because CHICKEN uses a copying semi-space collector.

18.11.3 set-finalizer!

```
[procedure] (set-finalizer! X PROC)
```

Registers a procedure of one argument `PROC`, that will be called as soon as the non-immediate data object `X` is about to be garbage-collected (with that object as its argument). Note that the finalizer will **not** be called when interrupts are disabled. This procedure returns `X`.

18.11.4 set-gc-report!

```
[procedure] (set-gc-report! FLAG)
```

Print statistics after every GC, depending on `FLAG`. A value of `#t` shows statistics after every major GC. A true value different from `#t` shows statistics after every minor GC. `#f` switches statistics off.

18.12 Other control structures

18.12.1 andmap

```
[procedure] (andmap PROC LIST1 ...)
```

Repeatedly calls `PROC` with arguments taken from `LIST1 ...`. If any invocation should return `#f`, the result of `andmap` is `#f`. If all invocations return a true result, then the result of `andmap` is `#t`.

18.12.2 ormap

```
[procedure] (ormap PROC LIST1 ...)
```

Repeatedly calls `PROC` with arguments taken from `LIST1 ...`. If any invocation should return a value different from `#f`, then this value is returned as the result of `ormap`. If all invocations return **#f**, then the result of `ormap` is `#f`.

18.12.3 promise?

```
[procedure] (promise? X)
```

Returns `#t` if `X` is a promise returned by `delay`, or `#f` otherwise.

18.13 String utilities

18.13.1 reverse-list->string

```
[procedure] (reverse-list->string LIST)
```

Returns a string with the characters in `LIST` in reverse order. This is equivalent to `(list->string (reverse LIST))`, but much more efficient.

18.14 Generating uninterned symbols

18.14.1 gensym

```
[procedure] (gensym [STRING-OR-SYMBOL])
```

Returns a newly created uninterned symbol. If an argument is provided, the new symbol is prefixed with that argument.

18.14.2 string->uninterned-symbol

```
[procedure] (string->uninterned-symbol STRING)
```

Returns a newly created, unique symbol with the name `STRING`.

18.15 Standard Input/Output

18.15.1 port?

```
[procedure] (port? X)
```

Returns `#t` if `X` is a port object or `#f` otherwise.

18.15.2 print

`[procedure] (print EXP1 EXP2 ...)`

Outputs the arguments `EXP1 EXP2 ...` using `display` and writes a newline character to the port that is the value of `(current-output-port)`. Returns its first argument.

18.15.3 print*

`[procedure] (print* EXP1 ...)`

Similar to `print`, but does not output a terminating newline character and performs a `flush-outout` after writing its arguments.

18.16 User-defined named characters

18.16.1 char-name

`[procedure] (char-name SYMBOL-OR-CHAR [CHAR])`

This procedure can be used to inquire about character names or to define new ones. With a single argument the behavior is as follows: If `SYMBOL-OR-CHAR` is a symbol, then `char-name` returns the character with this name, or `#f` if no character is defined under this name. If `SYMBOL-OR-CHAR` is a character, then the name of the character is returned as a symbol, or `#f` if the character has no associated name.

If the optional argument `CHAR` is provided, then `SYMBOL-OR-CHAR` should be a symbol that will be the new name of the given character. If multiple names designate the same character, then the `write` will use the character name that was defined last.

```
(char-name 'space)                  ==> #\space
(char-name #\space)                 ==> space
(char-name 'bell)                   ==> #f
(char-name (integer->char 7))       ==> #f
(char-name 'bell (integer->char 7))
(char-name 'bell)                   ==> #\bell
(char->integer (char-name 'bell))   ==> 7
```

18.17 Vectors

18.17.1 vector-copy!

`[procedure] (vector-copy! VECTOR1 VECTOR2 [COUNT])`

Copies contents of `VECTOR1` into `VECTOR2`. If the argument `COUNT` is given, it specifies the maximal number of elements to be copied. If not given, the minimum of the lengths of the argument vectors is copied.

Exceptions: `(exn bounds)`

18.17.2 vector-resize

`[procedure] (vector-resize VECTOR N [INIT])`

Creates and returns a new vector with the contents of `VECTOR` and length `N`. If `N` is greater than the original length of `VECTOR`, then all additional items are initialized to `INIT`. If `INIT` is not specified, the contents are initialized to some unspecified value.

18.18 The unspecified value

18.18.1 void

`[procedure] (void)`

Returns an unspecified value.

18.19 Continuations

18.19.1 call/cc

`[procedure] (call/cc PROCEDURE)`

An alias for `call-with-current-continuation`.

18.19.2 continuation-capture

`[procedure] (continuation-capture PROCEDURE)`

Creates a continuation object representing the current continuation and tail-calls `PROCEDURE` with this continuation as the single argument.

More information about this continuation API can be found in the paper http://repository.readscheme.org/ftp/papers/sw2001/feeley.pdf *A Better API for first class Continuations* by

Marc Feeley.

18.19.3 continuation?

`[procedure] (continuation? X)`

Returns `#t` if `X` is a continuation object, or `#f` otherwise.

18.19.4 continuation-graft

`[procedure] (continuation-graft CONT THUNK)`

Calls the procedure `THUNK` with no arguments and the implicit continuation `CONT`.

18.19.5 continuation-return

`[procedure] (continuation-return CONT VALUE ...)`

Returns the value(s) to the continuation `CONT`. `continuation-return` could be implemented like this:

```
(define (continuation-return k . vals)
  (continuation-graft
    k
    (lambda () (apply values vals)) ) )
```

18.20 Setters

SRFI-17 is fully implemented. For more information see: SRFI-17.

18.20.1 setter

`[procedure] (setter PROCEDURE)`

Returns the setter-procedure of `PROCEDURE`, or signals an error if `PROCEDURE` has no associated setter-procedure.

Note that `(set! (setter PROC) ...)` for a procedure that has no associated setter procedure yet is a very slow operation (the old procedure is replaced by a modified copy, which involves a garbage collection).

18.20.2 getter-with-setter

```
[procedure] (getter-with-setter GETTER SETTER)
```

Returns a copy of the procedure `GETTER` with the associated setter procedure `SETTER`. Contrary to the SRFI specification, the setter of the returned procedure may be changed.

18.21 Reader extensions

18.21.1 define-reader-ctor

```
[procedure] (define-reader-ctor SYMBOL PROC)
```

Define new read-time constructor for `#,` read syntax. For further information, see the documentation for SRFI-10.

18.21.2 set-read-syntax!

```
[procedure] (set-read-syntax! CHAR-OR-SYMBOL PROC)
```

When the reader is encounting the non-whitespace character `CHAR` while reading an expression from a given port, then the procedure `PROC` will be called with that port as its argument. The procedure should return a value that will be returned to the reader:

```
; A simple RGB color syntax:

(set-read-syntax! #\%
  (lambda (port)
    (apply vector
      (map (cut string->number <> 16)
```

(string-chop (read-string 6 port) 2)))))

```
(with-input-from-string "(1 2 %f0f0f0 3)" read)
; ==> (1 2 #(240 240 240) 3)
```

If `CHAR-OR-SYMBOL` is a symbol, then a so-called *read-mark* handler is defined. In that case the handler procedure will be called when a character-sequence of the form

```
#!SYMBOL
```

is encountered.

You can undo special handling of read-syntax by passing `#f` as the second argument (if the syntax was previously defined via `set-read-syntax!`).

Note that all of CHICKEN's special non-standard read-syntax is handled directly by the reader to disable built-in read-syntax, define a handler that triggers an error (for example).

18.21.3 set-sharp-read-syntax!

`[procedure] (set-sharp-read-syntax! CHAR-OR-SYMBOL PROC)`

Similar to `set-read-syntax!`, but allows defining new `#<CHAR> ...` reader syntax. If the first argument is a symbol, then this procedure is equivalent to `set-read-syntax!`.

18.21.4 set-parameterized-read-syntax!

`[procedure] (set-parameterized-read-syntax! CHAR-OR-SYMBOL PROC)`

Similar to `set-sharp-read-syntax!`, but intended for defining reader syntax of the form `#<NUMBER><CHAR> ...`. The handler procedure `PROC` will be called with two arguments: the input port and the number preceding the dispatching character. If the first argument is a symbol, then this procedure is equivalent to `set-read-syntax!`.

18.21.5 copy-read-table

`[procedure] (copy-read-table READ-TABLE)`

Returns a copy of the given read-table. You can access the currently active read-table with `(current-read-table)`.

Previous: Parameters

Next: Unit eval

19 Unit eval

This unit has support for evaluation and macro-handling. This unit is used by default, unless the program is compiled with the `-explicit-use` option.

19.1 Loading code

19.1.1 load

```
[procedure] (load FILE [EVALPROC])
```

Loads and evaluates expressions from the given source file, which may be either a string or an input port. Each expression read is passed to `EVALPROC` (which defaults to `eval`). On platforms that support it (currently native Windows, Linux ELF and Solaris), `load` can be used to load compiled programs:

```
% cat x.scm
(define (hello) (print "Hello!"))
% csc -s x.scm
% csi -q
#;1> (load "x.so")
; loading x.so ...
#;2> (hello)
Hello!
#;3>
```

The second argument to `load` is ignored when loading compiled code. If source code is loaded from a port, then that port is closed after all expressions have been read.

Compiled code can be re-loaded, but care has to be taken, if code from the replaced dynamically loaded module is still executing (i.e. if an active continuation refers to compiled code in the old module).

Support for reloading compiled code dynamically is still experimental.

19.1.2 load-relative

```
[procedure] (load-relative FILE [EVALPROC])
```

Similar to `load`, but loads `FILE` relative to the path of the currently loaded file.

19.1.3 load-noisily

```
[procedure] (load-noisily FILE #!key EVALUATOR TIME PRINTER)
```

As `load` but the result(s) of each evaluated toplevel-expression is written to standard output. If `EVALUATOR` is given and not `#f`, then each expression is evaluated by calling this argument with the read expression as argument. If `TIME` is given and not false, then the execution time of each expression is shown (as with the `time` macro). If `PRINTER` is given and not false, then each expression is printed before evaluation by applying the expression to the value of this argument, which should be a one-argument procedure.

19.1.4 load-library

`[procedure] (load-library UNIT [LIBRARYFILE])`

On platforms that support dynamic loading, `load-library` loads the compiled library unit `UNIT` (which should be a symbol). If the string `LIBRARYFILE` is given, then the given shared library will be loaded and the toplevel code of the contained unit will be executed. If no `LIBRARYFILE` argument is given, then the following libraries are checked for the required unit:

- a file named *`<UNIT>.so`*
- the files given in the parameter `dynamic-load-libraries`

If the unit is not found, an error is signaled. When the library unit can be successfully loaded, a feature-identifier named `UNIT` is registered. If the feature is already registered before loading, the `load-library` does nothing.

19.1.5 set-dynamic-load-mode!

`[procedure] (set-dynamic-load-mode! MODELIST)`

On systems that support dynamic loading of compiled code via the `dlopen(3)` interface (for example Linux and Solaris), some options can be specified to fine-tune the behaviour of the dynamic linker. `MODE` should be a list of symbols (or a single symbol) taken from the following set:

`local`
: If `local` is given, then any C/C++ symbols defined in the dynamically loaded file are not available for subsequently loaded files and libraries. Use this if you have linked foreign code into your dynamically loadable file and if you don't want to export them (for example because you want to load another file that defines the same symbols).

`global`
: The default is `global`, which means all C/C++ symbols are available to code loaded at a later stage.

`now`
: If `now` is specified, all symbols are resolved immediately.

`lazy`
: Unresolved symbols are resolved as code from the file is executed. This is the default.

Note that this procedure does not control the way Scheme variables are handled - this facility is mainly of interest when accessing foreign code.

19.2 Read-eval-print loop

19.2.1 repl

```
[procedure] (repl)
```

Start a new read-eval-print loop. Sets the `reset-handler` so that any invocation of `reset` restarts the read-eval-print loop. Also changes the current exception-handler to display a message, write any arguments to the value of `(current-error-port)` and reset.

19.3 Macros

19.3.1 get-line-number

```
[procedure] (get-line-number EXPR)
```

If `EXPR` is a pair with the car being a symbol, and line-number information is available for this expression, then this procedure returns the associated line number. If line-number information is not available, then `#f` is returned. Note that line-number information for expressions is only available in the compiler.

19.3.2 macro?

```
[procedure] (macro? SYMBOL)
```

Returns `#t` if there exists a macro-definition for `SYMBOL`.

19.3.3 macroexpand

```
[procedure] (macroexpand X)
```

If `X` is a macro-form, expand the macro (and repeat expansion until expression is a non-macro form). Returns the resulting expression.

19.3.4 macroexpand-1

```
[procedure] (macroexpand-1 X)
```

If `X` is a macro-form, expand the macro. Returns the resulting expression.

19.3.5 undefine-macro!

```
[procedure] (undefine-macro! SYMBOL)
```

Remove the current macro-definition of the macro named `SYMBOL`.

19.3.6 syntax-error

`[procedure] (syntax-error [LOCATION] MESSAGE ARGUMENT ...)`

Signals an exception of the kind `(exn syntax)`. Otherwise identical to `error`.

19.4 Loading extension libraries

This functionality is only available on platforms that support dynamic loading of compiled code. Currently Linux, BSD, Solaris, Windows (with Cygwin) and HP/UX are supported.

19.4.1 repository-path

[parameter] repository-path

Contains a string naming the path to the extension repository, which defaults to either the value of the environment variable `CHICKEN_REPOSITORY`, the value of the environment variable `CHICKEN_HOME` or the default library path (usually `/usr/local/lib/chicken` on UNIX systems).

19.4.2 extension-information

`[procedure] (extension-information ID)`

If an extension with the name `ID` is installed and if it has a setup-information list registered in the extension repository, then the info-list is returned. Otherwise `extension-information` returns `#f`.

19.4.3 provide

`[procedure] (provide ID ...)`

Registers the extension IDs `ID ...` as loaded. This is mainly intended to provide aliases for certain extension identifiers.

19.4.4 provided?

`[procedure] (provided? ID ...)`

Returns `#t` if the extension with the IDs `ID ...` are currently loaded, or `#f` otherwise.

19.4.5 require

```
[procedure] (require ID ...)
[procedure] (require-for-syntax ID ...)
```

If the extension library `ID` is not already loaded into the system, then `require` will lookup the location of the shared extension library and load it. If `ID` names a library-unit of the base system, then it is loaded via `load-library`. If no extension library is available for the given ID, then an attempt is made to load the file `ID.so` or `ID.scm` (in that order) from one of the following locations:

- the current include path, which defaults to the pathnames given in `CHICKEN_INCLUDE_PATH` and `CHICKEN_HOME`.
- the current directory

`ID` should be a string or a symbol. The difference between `require` and `require-for-syntax` is the the latter loads the extension library at compile-time (the argument is still evaluated), while the former loads it at run-time.

19.4.6 set-extension-specifier!

```
[procedure] (set-extension-specifier! SYMBOL PROC)
```

Registers the handler-procedure `PROC` as a extension-specifier with the name `SYMBOL`. This facility allows extending the set of valid extension specifiers to be used with `require-extension`. When `register-extension` is called with an extension specifier of the form `(SPEC ...)` and `SPEC` has been registered with `set-extension-specifier!`, then `PROC` will be called with two arguments: the specifier and the previously installed handler (or `#f` if no such handler was defined). The handler should return a new specifier that will be processed recursively. If the handler returns a vector, then each element of the vector will be processed recursively. Alternatively the handler may return a string which specifies a file to be loaded:

```
(eval-when (compile eval)
  (set-extension-specifier!
    'my-package
    (lambda (spec old)
      (make-pathname my-package-directory (->string (cadr spec))) ) ) )

(require-extension (my-package stuff))     ; --> expands into '(load "my-packag
```

Note that the handler has to be registered at compile time, if it is to be visible in compiled code.

19.5 System information

19.5.1 chicken-home

```
[procedure] (chicken-home)
```

Returns a string given the installation directory (usually `/usr/local/share/chicken` on UNIX-like systems). If the environment variable `CHICKEN_HOME` is set, then its value will be returned. As a last option, if the environment variable `CHICKEN_PREFIX` is set, then `chicken-home` will return `$CHICKEN_PREFIX/share`.

19.6 Eval

19.6.1 eval

`[procedure] (eval EXP [ENVIRONMENT])`

Evaluates `EXP` and returns the result of the evaluation. The second argument is optional and defaults to the value of `(interaction-environment)`.

Previous: Unit library

Next: Unit extras

20 Unit extras

This unit contains a collection of useful utility definitions. This unit is used by default, unless the program is compiled with the `-explicit-use` option.

20.1 Lists

20.1.1 alist-ref

```
[procedure] (alist-ref KEY ALIST [TEST [DEFAULT]])
```

Looks up `KEY` in `ALIST` using `TEST` as the comparison function (or `eqv?` if no test was given) and returns the cdr of the found pair, or `DEFAULT` (which defaults to `#f`).

20.1.2 alist-update!

```
[procedure] (alist-update! KEY VALUE ALIST [TEST])
```

If the list `ALIST` contains a pair of the form `(KEY . X)`, then this procedure replaces `X` with `VALUE` and returns `ALIST`. If `ALIST` contains no such item, then `alist-update!` returns `((KEY . VALUE) . ALIST)`. The optional argument `TEST` specifies the comparison procedure to search a matching pair in `ALIST` and defaults to `eqv?`.

20.1.3 atom?

```
[procedure] (atom? X)
```

Returns `#t` if `X` is not a pair. This is identical to `not-pair?` from Unit srfi-1 but kept for historical reasons.

20.1.4 rassoc

```
[procedure] (rassoc KEY LIST [TEST])
```

Similar to `assoc`, but compares `KEY` with the `cdr` of each pair in `LIST` using `TEST` as the comparison procedures (which defaults to `eqv?`.

20.1.5 butlast

```
[procedure] (butlast LIST)
```

Returns a fresh list with all elements but the last of `LIST`.

20.1.6 chop

```
[procedure] (chop LIST N)
```

Returns a new list of sublists, where each sublist contains `N` elements of `LIST`. If `LIST` has a length that is not a multiple of `N`, then the last sublist contains the remaining elements.

```
(chop '(1 2 3 4 5 6) 2) ==> ((1 2) (3 4) (5 6))
(chop '(a b c d) 3)     ==> ((a b c) (d))
```

20.1.7 compress

```
[procedure] (compress BLIST LIST)
```

Returns a new list with elements taken from `LIST` with corresponding true values in the list `BLIST`.

```
(define nums '(99 100 110 401 1234))
(compress (map odd? nums) nums)      ==> (99 401)
```

20.1.8 flatten

```
[procedure] (flatten LIST1 ...)
```

Returns `LIST1 ...` concatenated together, with nested lists removed (flattened).

20.1.9 intersperse

```
[procedure] (intersperse LIST X)
```

Returns a new list with `X` placed between each element.

20.1.10 join

```
[procedure] (join LISTOFLISTS [LIST])
```

Concatenates the lists in `LISTOFLISTS` with `LIST` placed between each sublist. `LIST` defaults to the empty list.

```
(join '((a b) (c d) (e)) '(x y)) ==> (a b x y c d x y e)
(join '((p q) () (r (s) t)) '(-))  ==> (p q - - r (s) t)
```

`join` could be implemented as follows:

```
(define (join lstoflsts #!optional (lst '()))
  (apply append (intersperse lstoflists lst)) )
```

20.1.11 shuffle

```
[procedure] (shuffle LIST)
```

Returns `LIST` with its elements sorted in a random order.

20.1.12 tail?

```
[procedure] (tail? X LIST)
```

Returns true if `X` is one of the tails (cdr's) of `LIST`.

20.2 String-port extensions

20.2.1 call-with-input-string

```
[procedure] (call-with-input-string STRING PROC)
```

Calls the procedure `PROC` with a single argument that is a string-input-port with the contents of `STRING`.

20.2.2 call-with-output-string

```
[procedure] (call-with-output-string PROC)
```

Calls the procedure `PROC` with a single argument that is a string-output-port. Returns the accumulated output-string.

20.2.3 with-input-from-string

```
[procedure] (with-input-from-string STRING THUNK)
```

Call procedure `THUNK` with the current input-port temporarily bound to an input-string-port with the contents of `STRING`.

20.2.4 with-output-to-string

```
[procedure] (with-output-to-string THUNK)
```

Call procedure `THUNK` with the current output-port temporarily bound to a string-output-port and return the accumulated output string.

20.3 Formatted output

20.3.1 printf

```
[procedure] (fprintf PORT FORMATSTRING ARG ...)
[procedure] (printf FORMATSTRING ARG)
[procedure] (sprintf FORMATSTRING ARG ...)
[procedure] (format FORMATSTRING ARG ...)
```

Simple formatted output to a given port (`fprintf`), the value of `(current-output-port)` (`printf`) or a string (`sprintf/format`). The `FORMATSTRING` can contain any sequence of characters. The character `~' prefixes special formatting directives:

`~%`

write newline character

`~N`

the same as `~%`

`~S`

write the next argument

`~A`

display the next argument

`~\n`

skip all whitespace in the format-string until the next non-whitespace character

`~B`

write the next argument as a binary number

`~O`

write the next argument as an octal number

`~X`

write the next argument as a hexadecimal number

`~C`

write the next argument as a character

`~~`

display `~'

`~!`

flush all pending output

`~?`

invoke formatted output routine recursively with the next two arguments as format-string and list of parameters

20.4 Hash tables

CHICKEN implements SRFI-69. For more information, see SRFI-69.

A setter for `hash-table-ref` is defined, so

```
(set! (hash-table-ref HT KEY) VAL)
```

is equivalent to

```
(hash-table-set! HT KEY VAL)
```

As an extension to SRFI-69, `hash-table-update!` and `hash-table-update!/default` return the new value (after applying the update procedure).

20.5 Queues

20.5.1 list->queue

```
[procedure] (list->queue LIST)
```

Returns `LIST` converted into a queue, where the first element of the list is the same as the first element of the queue. The resulting queue may share memory with the list and the list should not be modified after this operation.

20.5.2 make-queue

```
[procedure] (make-queue)
```

Returns a newly created queue.

20.5.3 queue?

```
[procedure] (queue? X)
```

Returns `#t` if `X` is a queue, or `#f` otherwise.

20.5.4 queue->list

```
[procedure] (queue->list QUEUE)
```

Returns `QUEUE` converted into a list, where the first element of the list is the same as the first element of the queue. The resulting list may share memory with the queue object and should not be modified.

20.5.5 queue-add!

```
[procedure] (queue-add! QUEUE X)
```

Adds `X` to the rear of `QUEUE`.

20.5.6 queue-empty?

```
[procedure] (queue-empty? QUEUE)
```

Returns `#t` if `QUEUE` is empty, or `#f` otherwise.

20.5.7 queue-first

```
[procedure] (queue-first QUEUE)
```

Returns the first element of `QUEUE`. If `QUEUE` is empty an error is signaled

20.5.8 queue-last

```
[procedure] (queue-last QUEUE)
```

Returns the last element of `QUEUE`. If `QUEUE` is empty an error is signaled

20.5.9 queue-remove!

```
[procedure] (queue-remove! QUEUE)
```

Removes and returns the first element of `QUEUE`. If `QUEUE` is empty an error is signaled

20.5.10 queue-push-back!

```
[procedure] (queue-push-back! QUEUE ITEM)
```

Pushes an item into the first position of a queue, i.e. the next `queue-remove!` will return `ITEM`.

20.5.11 queue-push-back-list!

```
[procedure] (queue-push-back-list! QUEUE LIST)
```

Pushes the items in item-list back onto the queue, so that `(car LIST)` becomes the next removable item.

20.6 Sorting

20.6.1 merge

```
[procedure] (merge LIST1 LIST2 LESS?)
[procedure] (merge! LIST1 LIST2 LESS?)
```

Joins two lists in sorted order. `merge!` is the destructive version of merge. `LESS?` should be a procedure of two arguments, that returns true if the first argument is to be ordered before the second argument.

20.6.2 sort

```
[procedure] (sort SEQUENCE LESS?)
[procedure] (sort! SEQUENCE LESS?)
```

Sort `SEQUENCE`, which should be a list or a vector. `sort!` is the destructive version of sort.

20.6.3 sorted?

```
[procedure] (sorted? SEQUENCE LESS?)
```

Returns true if the list or vector `SEQUENCE` is already sorted.

20.7 Random numbers

20.7.1 random

```
[procedure] (random N)
```

Returns an exact random integer from 0 to `N`-1.

20.7.2 randomize

```
[procedure] (randomize [X])
```

Set random-number seed. If `X` is not supplied, the current time is used. On startup (when the `extras` unit is initialized), the random number generator is initialized with the current time.

20.8 Input/Output extensions

20.8.1 make-input-port

```
[procedure] (make-input-port READ READY? CLOSE [PEEK])
```

Returns a custom input port. Common operations on this port are handled by the given parameters, which should be procedures of no arguments. `READ` is called when the next character is to be read and should return a character or `#!eof`. `READY?` is called when `char-ready?` is called on this port and should return `#t` or `#f`. `CLOSE` is called when the port is closed. `PEEK` is called when `peek-char` is called on this port and should return a character or `#!eof`. if the argument `PEEK` is not given, then `READ` is used instead and the created port object handles peeking automatically (by calling `READ` and buffering the character).

20.8.2 make-output-port

```
[procedure] (make-output-port WRITE CLOSE [FLUSH])
```

Returns a custom output port. Common operations on this port are handled by the given parameters, which should be procedures. `WRITE` is called when output is sent to the port and receives a single argument, a string. `CLOSE` is called when the port is closed and should be a procedure of no arguments. `FLUSH` (if provided) is called for flushing the output port.

20.8.3 pretty-print

```
[procedure] (pretty-print EXP [PORT])
[procedure] (pp EXP [PORT])
```

Print expression nicely formatted. `PORT` defaults to the value of `(current-output-port)`.

20.8.4 pretty-print-width

(Parameter) Specifies the maximal line-width for pretty printing, after which line wrap will occur.

20.8.5 read-file

```
[procedure] (read-file [FILE-OR-PORT [READER [MAXCOUNT]]])
```

Returns a list containing all toplevel expressions read from the file or port `FILE-OR-PORT`. If no argument is given, input is read from the port that is the current value of `(current-input-port)`. After all expressions are read, and if the argument is a port, then the port will not be closed. The `READER` argument specifies the procedure used to read expressions from the given file or port and defaults to `read`. The reader procedure will be called with a single argument (an input port). If `MAXCOUNT` is given then only up to `MAXCOUNT` expressions will be read in.

20.8.6 read-line

20.8.7 write-line

```
[procedure] (read-line [PORT [LIMIT]])
[procedure] (write-line STRING [PORT])
```

Line-input and -output. `PORT` defaults to the value of `(current-input-port)` and `(current-output-port)`, respectively. if the optional argument `LIMIT` is given and not `#f`, then `read-line` reads at most `LIMIT` characters per line.

20.8.8 read-lines

```
[procedure] (read-lines [PORT [MAX]])
```

Read `MAX` or fewer lines from `PORT`. `PORT` defaults to the value of `(current-input-port)`. `PORT` may optionally be a string naming a file.

20.8.9 read-string

20.8.10 read-string!

20.8.11 write-string

```
[procedure] (read-string [NUM [PORT]])
[procedure] (read-string! NUM STRING [PORT [START]])
[procedure] (write-string STRING [NUM [PORT]]
```

Read or write `NUM` characters from/to `PORT`, which defaults to the value of `(current-input-port)` or `(current-output-port)`, respectively. If `NUM` is `#f` or not given, then all data up to the end-of-file is read, or, in the case of `write-string` the whole string is written. If no more input is available, `read-string` returns the empty string. `read-string!` reads destructively into the given `STRING` argument, but never more characters that would fit into `STRING`. If `START` is given, then the read characters are stored starting at that position. `read-string!` returns the actual number of characters read.

20.8.12 read-token

```
[procedure] (read-token PREDICATE [PORT])
```

Reads characters from `PORT` (which defaults to the value of `(current-input-port)`) and calls the procedure `PREDICATE` with each character until `PREDICATE` returns false. Returns a string with the accumulated characters.

20.8.13 with-error-output-to-port

```
[procedure] (with-error-output-to-port PORT THUNK)
```

Call procedure `THUNK` with the current error output-port temporarily bound to `PORT`.

20.8.14 with-input-from-port

```
[procedure] (with-input-from-port PORT THUNK)
```

Call procedure `THUNK` with the current input-port temporarily bound to `PORT`.

20.8.15 with-output-to-port

```
[procedure] (with-output-to-port PORT THUNK)
```

Call procedure `THUNK` with the current output-port temporarily bound to `PORT`.

20.9 Strings

20.9.1 conc

```
[procedure] (conc X ...)
```

Returns a string with the string-represenation of all arguments concatenated together. `conc` could be implemented as

```
(define (conc . args)
  (apply string-append (map ->string args)) )
```

20.9.2 ->string

```
[procedure] (->string X)
```

Returns a string-representation of `X`.

20.9.3 string-chop

```
[procedure] (string-chop STRING LENGTH)
```

Returns a list of substrings taken by *chopping* `STRING` every `LENGTH` characters:

```
(string-chop "one two three" 4)  ==>  ("one " "two " "thre" "e")
```

20.9.4 string-chomp

```
[procedure] (string-chomp STRING [SUFFIX])
```

If `STRING` ends with `SUFFIX`, then this procedure returns a copy of its first argument with the suffix removed, otherwise returns `STRING` unchanged. `SUFFIX` defaults to `"\n"`.

20.9.5 string-compare3

```
[procedure] (string-compare3 STRING1 STRING2)
[procedure] (string-compare3-ci STRING1 STRING2)
```

Perform a three-way comparison between the `STRING1` and `STRING2`, returning either `-1` if `STRING1` is lexicographically less than `STRING2`, `0` if it is equal, or `1` if it s greater. `string-compare3-ci` performs a case-insensitive comparison.

20.9.6 string-intersperse

```
[procedure] (string-intersperse LIST [STRING])
```

Returns a string that contains all strings in `LIST` concatenated together. `STRING` is placed between each concatenated string and defaults to `" "`.

```
(string-intersperse '("one" "two") "three")
```

is equivalent to

```
(apply string-append (intersperse '("one" "two") "three"))
```

20.9.7 string-split

```
[procedure] (string-split STRING [DELIMITER-STRING [KEEPEMPTY]])
```

Split string into substrings separated by the given delimiters. If no delimiters are specified, a string comprising the tab, newline and space characters is assumed. If the parameter `KEEPEMPTY` is given and not `#f`, then empty substrings are retained:

```
(string-split "one two three") ==> ("one" "two" "three")
(string-split "foo:bar::baz:" ":" #t) ==> ("foo" "bar" "" "baz" "")
```

20.9.8 string-translate

```
[procedure] (string-translate STRING FROM [TO])
```

Returns a fresh copy of `STRING` with characters matching `FROM` translated to `TO`. If `TO` is omitted, then matching characters are removed. `FROM` and `TO` may be a character, a string or a list. If both `FROM` and `TO` are strings, then the character at the same position in `TO` as the matching character in `FROM` is substituted.

20.9.9 string-translate*

```
[procedure] (string-translate* STRING SMAP)
```

Substitutes elements of `STRING` according to `SMAP`. `SMAP` should be an association-list where each element of the list is a pair of the form `(MATCH \. REPLACEMENT)`. Every occurrence of the string `MATCH` in `STRING` will be replaced by the string `REPLACEMENT`:

```
(string-translate*
  "<h1>this is a \"string\"</h1>"
  '(("<" . "&lt:") (">" . "&gt;") ("\"" . "&quot;")) )
=>  "&lt;h1&gt;this is a &quot;string&quot;&lt;/ht&gt;"
```

20.9.10 substring=?

```
[procedure] (substring=? STRING1 STRING2 [START1 [START2 [LENGTH]]])
[procedure] (substring-ci=? STRING1 STRING2 [START1 [START2 [LENGTH]]])
```

Returns `#t` if the strings `STRING1` and `STRING2` are equal, or `#f` otherwise. The comparison starts at the positions `START1` and `START2` (which default to 0), comparing `LENGTH` characters (which defaults to the minimum of the remaining length of both strings).

20.9.11 substring-index

```
[procedure] (substring-index WHICH WHERE [START])
[procedure] (substring-index-ci WHICH WHERE [START])
```

Searches for first index in string `WHERE` where string `WHICH` occurs. If the optional argument `START` is given, then the search starts at that index. `substring-index-ci` is a case-insensitive version of `substring-index`.

20.10 Combinators

20.10.1 any?

```
[procedure] (any? X)
```

Ignores it's argument and always returns `#t`. This is actually useful sometimes.

20.10.2 constantly

```
[procedure] (constantly X ...)
```

Returns a procedure that always returns the values `X ...` regardless of the number and value of its arguments.

```
(constantly X) <=> (lambda args X)
```

20.10.3 complement

```
[procedure] (complement PROC)
```

Returns a procedure that returns the boolean inverse of `PROC`.

```
(complement PROC) <=> (lambda (x) (not (PROC x)))
```

20.10.4 compose

```
[procedure] (compose PROC1 PROC2 ...)
```

Returns a procedure that represents the composition of the argument-procedures `PROC1 PROC2 ...`.

```
(compose F G) <=> (lambda args
                      (call-with-values
                         (lambda () (apply G args))
                         F))
```

`(compose)` is equivalent to `values`.

20.10.5 conjoin

`[procedure] (conjoin PRED ...)`

Returns a procedure that returns `#t` if its argument satisfies the predicates `PRED ...`. <enscript highlight=scheme> ((conjoin odd? positive?) 33) = #t ((conjoin odd? positive?) -33) = #f </enscript>

20.10.6 disjoin

`[procedure] (disjoin PRED ...)`

Returns a procedure that returns `#t` if its argument satisfies any predicate `PRED ...`. <enscript highlight=scheme> ((disjoin odd? positive?) 32) = #t ((disjoin odd? positive?) -32) = #f </enscript>

20.10.7 each

`[procedure] (each PROC ...)`

Returns a procedure that applies `PROC ...` to its arguments, and returns the result(s) of the last procedure application. For example

```
(each pp eval)
```

is equivalent to

```
(lambda args
  (apply pp args)
  (apply eval args) )
```

`(each PROC)` is equivalent to `PROC` and `(each)` is equivalent to `noop`.

20.10.8 flip

`[procedure] (flip PROC)`

Returns a two-argument procedure that calls `PROC` with its arguments swapped: <enscript highlight=scheme> (flip PROC) (lambda (x y) (PROC y x)) </enscript>

20.10.9 identity

`[procedure] (identity X)`

Returns its sole argument `X`.

20.10.10 project

```
[procedure] (project N)
```

Returns a procedure that returns its `N`th argument (starting from 0).

20.10.11 list-of

```
[procedure] (list-of PRED)
```

Returns a procedure of one argument that returns `#t` when applied to a list of elements that all satisfy the predicate procedure `PRED`, or `#f` otherwise.

```
((list-of even?) '(1 2 3))   ==> #f
((list-of number?) '(1 2 3)) ==> #t
```

20.10.12 noop

```
[procedure] (noop X ...)
```

Ignores it's arguments, does nothing and returns an unspecified value.

20.10.13 o

```
[procedure] (o PROC ...)
```

A single value version of `compose` (slightly faster). `(o)` is equivalent to `identity`.

20.11 Binary searching

20.11.1 binary-search

```
[procedure] (binary-search SEQUENCE PROC)
```

Performs a binary search in `SEQUENCE`, which should be a sorted list or vector. `PROC` is called to compare items in the sequence, should accept a single argument and return an exact integer: zero if the searched value is equal to the current item, negative if the searched value is *less* than the current item, and positive otherwise. Returns the index of the found value or `#f` otherwise.

Previous: Unit eval

Next: Unit srfi-1

21 Unit srfi-1

List library, see the documentation for SRFI-1

Previous: Unit extras

Next: Unit srfi-4

22 Unit srfi-4

Homogeneous numeric vectors, see the documentation for SRFI-4 64-bit integer vectors (`u64vector` and `s64vector` are not supported.

The basic constructor procedures for number vectors are extended to allow allocating the storage in non garbage collected memory:

22.1 make-XXXvector

```
[procedure] (make-XXXvector SIZE [INIT NONGC FINALIZE])
```

Creates a SRFI-4 homogenous number vector of length `SIZE`. If `INIT` is given, it specifies the initial value for each slot in the vector. The optional arguments `NONGC` and `FINALIZE` define whether the vector should be allocated in a memory area not subject to garbage collection and whether the associated storage should be automatically freed (using finalization) when there are no references from Scheme variables and data. `NONGC` defaults to `#f` (the vector will be located in normal garbage collected memory) and `FINALIZE` defaults to `#t`. Note that the `FINALIZE` argument is only used when `NONGC` is true.

Additionally, the following procedures are provided:

22.2 u8vector->byte-vector

22.3 s8vector->byte-vector

22.4 u16vector->byte-vector

22.5 s16vector->byte-vector

22.6 u32vector->byte-vector

22.7 s32vector->byte-vector

22.8 f32vector->byte-vector

22.9 f64vector->byte-vector

```
[procedure] (u8vector->byte-vector U8VECTOR)
[procedure] (s8vector->byte-vector S8VECTOR)
[procedure] (u16vector->byte-vector U16VECTOR)
[procedure] (s16vector->byte-vector S16VECTOR)
[procedure] (u32vector->byte-vector U32VECTOR)
[procedure] (s32vector->byte-vector S32VECTOR)
[procedure] (f32vector->byte-vector F32VECTOR)
[procedure] (f64vector->byte-vector F64VECTOR)
```

Each of these procedures return the contents of the given vector as a 'packed' byte-vector. The byte order in that vector is platform-dependent (for example little-endian on an **Intel** processor). The returned byte-vector shares memory with the contents of the vector.

22.10 byte-vector->u8vector

22.11 byte-vector->s8vector

22.12 byte-vector->u16vector

22.13 byte-vector->s16vector

22.14 byte-vector->u32vector

22.15 byte-vector->s32vector

22.16 byte-vector->f32vector

22.17 byte-vector->f64vector

```
[procedure] (byte-vector->u8vector BYTE-VECTOR)
[procedure] (byte-vector->s8vector BYTE-VECTOR)
[procedure] (byte-vector->u16vector BYTE-VECTOR)
[procedure] (byte-vector->s16vector BYTE-VECTOR)
[procedure] (byte-vector->u32vector BYTE-VECTOR)
[procedure] (byte-vector->s32vector BYTE-VECTOR)
[procedure] (byte-vector->f32vector BYTE-VECTOR)
[procedure] (byte-vector->f64vector BYTE-VECTOR)
```

Each of these procedures return a vector where the argument `BYTE-VECTOR` is taken as a 'packed' representation of the contents of the vector. The argument-byte-vector shares memory with the contents of the vector.

22.18 subu8vector

22.19 subu16vector

22.20 subu32vector

22.21 subs8vector

22.22 subs16vector

22.23 subs32vector

22.24 subf32vector

22.25 subf64vector

```
[procedure] (subu8vector U8VECTOR FROM TO)
[procedure] (subu16vector U16VECTOR FROM TO)
[procedure] (subu32vector U32VECTOR FROM TO)
[procedure] (subs8vector S8VECTOR FROM TO)
[procedure] (subs16vector S16VECTOR FROM TO)
[procedure] (subs32vector S32VECTOR FROM TO)
[procedure] (subf32vector F32VECTOR FROM TO)
[procedure] (subf64vector F64VECTOR FROM TO)
```

Creates a number vector of the same type as the argument vector with the elements at the positions `FROM` up to but not including `TO`.

SRFI-17 Setters for `XXXvector-ref` are defined.

22.26 read-u8vector

`[procedure] (read-u8vector LENGTH [PORT])`

Reads `LENGTH` bytes from the `PORT` and returns a fresh `u8vector` or less of end-of-file is encountered. `PORT` defaults to the value of `(current-input-port)`. If `LENGTH` is `#f`, the vector will be filled completely until end-of-file is reached.

22.27 read-u8vector!

`[procedure] (read-u8vector! LENGTH U8VECTOR [PORT [START]])`

Reads `LENGTH` bytes from the `PORT` writing the read input into `U8VECTOR` beginning at `START` (or 0 if not given). `PORT` defaults to the value of `(current-input-port)`. If `LENGTH` is `#f`, the vector will be filled completely until end-of-file is reached. This procedure returns the number of bytes read.

22.28 write-u8vector

`[procedure] (write-u8vector U8VECTOR [PORT [START [END]]])`

Writes the bytes `U8VECTOR` between the indices `START` (inclusive) and `END` (exclusive) to `PORT`. `PORT` defaults to the value of `(current-output-port)`.

Previous: Unit srfi-1

Next: Unit srfi-13

23 Unit srfi-13

String library, see the documentation for SRFI-13

On systems that support dynamic loading, the `srfi-13` unit can be made available in the interpreter (`csi`) by entering

```
(require-extension srfi-13)
```

Previous: Unit srfi-4

Next: Unit srfi-14

24 Unit srfi-14

Character set library, see the documentation for SRFI-14

On systems that support dynamic loading, the `srfi-14` unit can be made available in the interpreter (`csi`) by entering

```
(require-extension srfi-14)
```

This library provides only the Latin-1 character set.

Previous: Unit srfi-13

Next: Unit match

25 Unit match

The runtime-support code for the Pattern Matching extensions. Note that to use the macros in normal compiled code it is not required to declare this unit as used. Only if forms containing these macros are to be expanded at runtime, this is needed.

Previous: Unit srfi-14

Next: Unit regex

26 Unit regex

This library unit provides support for regular expressions. The regular expression package used is PCRE (*Perl Compatible Regular Expressions*) written by Philip Hazel. See http://www.pcre.org for information about the particular regexp flavor and extensions provided by this library.

26.1 grep

```
[procedure] (grep REGEX LIST)
```

Returns all items of `LIST` that match the regular expression `REGEX`. This procedure could be defined as follows:

```
(define (grep regex lst)
  (filter (lambda (x) (string-search regex x)) lst) )
```

26.2 glob->regexp

```
[procedure] (glob->regexp PATTERN)
```

Converts the file-pattern `PATTERN` into a regular expression.

```
(glob->regexp "foo.*")
=> "foo\..*"
```

`PATTERN` should follow "glob" syntax. Allowed wildcards are

```
*
[C...]
[C1-C2]
[-C...]
?
```

26.3 regexp

```
[procedure] (regexp STRING [IGNORECASE [IGNORESPACE [UTF8]]])
```

Returns a precompiled regular expression object for `string`. The optional arguments `IGNORECASE`, `IGNORESPACE` and `UTF8` specify whether the regular expression should be matched with case- or whitespace-differences ignored, or whether the string should be treated as containing UTF-8 encoded characters, respectively.

26.4 regexp?

```
[procedure] (regexp? X)
```

Returns `#t` if `X` is a precompiled regular expression, or `#f` otherwise.

26.5 string-match

26.6 string-match-positions

```
[procedure] (string-match REGEXP STRING [START])
[procedure] (string-match-positions REGEXP STRING [START])
```

Matches the regular expression in `REGEXP` (a string or a precompiled regular expression) with `STRING` and returns either `#f` if the match failed, or a list of matching groups, where the first element is the complete match. If the optional argument `START` is supplied, it specifies the starting position in `STRING`. For each matching group the result-list contains either: `#f` for a non-matching but optional group; a list of start- and end-position of the match in `STRING` (in the case of `string-match-positions`); or the matching substring (in the case of `string-match`). Note that the exact string is matched. For searching a pattern inside a string, see below. Note also that `string-match` is implemented by calling `string-search` with the regular expression wrapped in `^ ... $`.

26.7 string-search

26.8 string-search-positions

```
[procedure] (string-search REGEXP STRING [START [RANGE]])
[procedure] (string-search-positions REGEXP STRING [START [RANGE]])
```

Searches for the first match of the regular expression in `REGEXP` with `STRING`. The search can be limited to `RANGE` characters.

26.9 string-split-fields

```
[procedure] (string-split-fields REGEXP STRING [MODE [START]])
```

Splits `STRING` into a list of fields according to `MODE`, where `MODE` can be the keyword `#:infix` (`REGEXP` matches field separator), the keyword `#:suffix` (`REGEXP` matches field terminator) or `#t` (`REGEXP` matches field), which is the default.

```
(define s "this is a string 1, 2, 3,")

(string-split-fields "[^ ]+" s)

  => ("this" "is" "a" "string" "1," "2," "3,")

(string-split-fields " " s #:infix)

  => ("this" "is" "a" "string" "1," "2," "3,")
```

```
(string-split-fields "," s #:suffix))

  => ("this is a string 1" " 2" " 3")
```

26.10 string-substitute

```
[procedure] (string-substitute REGEXP SUBST STRING [MODE])
```

Searches substrings in `STRING` that match `REGEXP` and substitutes them with the string `SUBST`. The substitution can contain references to subexpressions in `REGEXP` with the `\NUM` notation, where `NUM` refers to the NUMth parenthesized expression. The optional argument `MODE` defaults to 1 and specifies the number of the match to be substituted. Any non-numeric index specifies that all matches are to be substituted.

```
(string-substitute "([0-9]+) (eggs|chicks)"
                   "\\2 (\\1)" "99 eggs or 99 chicks" 2)
=> "99 eggs or chicks (99)"
```

26.11 string-substitute*

```
[procedure] (string-substitute* STRING SMAP [MODE])
```

Substitutes elements of `STRING` with `string-substitute` according to `SMAP`. `SMAP` should be an association-list where each element of the list is a pair of the form `(MATCH . REPLACEMENT)`. Every occurrence of the regular expression `MATCH` in `STRING` will be replaced by the string `REPLACEMENT`

```
(string-substitute* "<h1>Hello, world!</h1>"
                    '(("<[/A-Za-z0-9]+>" . "")))

=>  "Hello, world!"
```

26.12 regexp-escape

```
[procedure] (regexp-escape STRING)
```

Escapes all special characters in `STRING` with `\`, so that the string can be embedded into a regular expression.

```
(regexp-escape "^[0-9]+:.*$")
=>  "\\^\\[0-9\\]\\+:.\n.\\*\\$"
```

Previous: Unit match

Next: Unit srfi-18

27 Unit srfi-18

A simple multithreading package. This threading package follows largely the specification of SRFI-18. For more information see the documentation for SRFI-18.

Notes:

- `thread-start!` accepts a thunk (a zero argument procedure) as argument, which is equivalent to `(thread-start! (make-thread THUNK))`.
- `thread-sleep!` accepts a seconds real number value in addition to a time object.
- When an uncaught exception (i.e. an error) is signalled in a thread other than the primordial thread and warnings are enabled (see: `enable-warnings`, then a warning message is written to the port that is the value of `(current-error-port)`.
- Blocking I/O will block all threads, except for some socket operations (see the section about the `tcp` unit). An exception is the read-eval-print loop on UNIX platforms: waiting for input will not block other threads, provided the current input port reads input from a console.
- It is generally not a good idea for one thread to call a continuation created by another thread, if `dynamic-wind` is involved.
- When more than one thread compete for the current time-slice, the thread that was waiting first will become the next runnable thread.
- The dynamic environment of a thread consists of the following state:
    - The current input-, output- and error-port
    - The current exception handler
    - The values of all current parameters (created by `make-parameter`)
    - Any pending `dynamic-wind` thunks.

The following procedures are provided, in addition to the procedures defined in SRFI-18:

27.1 thread-signal!

`[procedure] (thread-signal! THREAD X)`

This will cause `THREAD` to signal the condition `X` once it is scheduled for execution. After signalling the condition, the thread continues with its normal execution.

27.2 thread-quantum

`[procedure] (thread-quantum THREAD)`

Returns the quantum of `THREAD`, which is an exact integer specifying the approximate time-slice of the thread in milliseconds.

27.3 thread-quantum-set!

`[procedure] (thread-quantum-set! THREAD QUANTUM)`

Sets the quantum of `THREAD` to `QUANTUM`.

27.4 thread-suspend!

`[procedure] (thread-suspend! THREAD)`

Suspends the execution of `THREAD` until resumed.

27.5 thread-resume!

`[procedure] (thread-resume! THREAD)`

Readies the suspended thread `THREAD`.

27.6 time->milliseconds

`[procedure] (time->milliseconds TIME)`

Converts a time object (as created via `current-time`) into an exact integer representing the number of milliseconds since process startup.

27.7 time->milliseconds

`[procedure] (time->milliseconds TIME)`

Converts a time object (as created via `current-time`) into an exact integer representing the number of milliseconds since process startup.

Previous: Unit regex

Next: Unit posix

28 Unit posix

This unit provides services as used on many UNIX-like systems. Note that the following definitions are not all available on non-UNIX systems like Windows. See below for Windows specific notes.

This unit uses the `regex`, `scheduler`, `extras` and `utils` units.

All errors related to failing file-operations will signal a condition of kind `(exn i/o file)`.

28.1 Directories

28.1.1 change-directory

`[procedure] (change-directory NAME)`

Changes the current working directory to `NAME`.

28.1.2 current-directory

`[procedure] (current-directory [DIR])`

Returns the name of the current working directory. If the optional argument `DIR` is given, then `(current-directory DIR)` is equivalent to `(change-directory DIR)`.

28.1.3 create-directory

`[procedure] (create-directory NAME)`

Creates a directory with the pathname `NAME`.

28.1.4 delete-directory

`[procedure] (delete-directory NAME)`

Deletes the directory with the pathname `NAME`. The directory has to be empty.

28.1.5 directory

`[procedure] (directory [PATHNAME [SHOW-DOTFILES?]])`

Returns a list with all files that are contained in the directory with the name `PATHNAME` (which defaults to the value of `(current-directory)`). If `SHOW-DOTFILES?` is given and not `#f`, then files beginning with .

are not included in the directory listing.

28.1.6 directory?

```
[procedure] (directory? NAME)
```

Returns `#t` if there exists a file with the name `NAME` and if that file is a directory, or `#f` otherwise.

28.1.7 glob

```
[procedure] (glob PATTERN1 ...)
```

Returns a list of the pathnames of all existing files matching `PATTERN1 ...`, which should be strings containing the usual file-patterns (with `*` matching zero or more characters and `?` matching zero or one character).

28.1.8 set-root-directory!

```
[procedure] (set-root-directory! STRING)
```

Sets the root directory for the current process to the path given in `STRING` (using the `chroot` function). If the current process has no root permissions, the operation will fail.

28.2 Pipes

28.2.1 call-with-input-pipe

28.2.2 call-with-output-pipe

```
[procedure] (call-with-input-pipe CMDLINE PROC [MODE])
[procedure] (call-with-output-pipe CMDLINE PROC [MODE])
```

Call `PROC` with a single argument: a input- or output port for a pipe connected to the subprocess named in `CMDLINE`. If `PROC` returns normally, the pipe is closed and any result values are returned.

28.2.3 close-input-pipe

28.2.4 close-output-pipe

```
[procedure] (close-input-pipe PORT)
[procedure] (close-output-pipe PORT)
```

Closes the pipe given in `PORT` and waits until the connected subprocess finishes. The exit-status code of the invoked process is returned.

28.2.5 create-pipe

```
[procedure] (create-pipe)
```

The fundamental pipe-creation operator. Calls the C function `pipe()` and returns 2 values: the file-descriptors of the input- and output-ends of the pipe.

28.2.6 open-input-pipe

```
[procedure] (open-input-pipe CMDLINE [MODE])
```

Spawns a subprocess with the command-line string `CMDLINE` and returns a port, from which the output of the process can be read. If `MODE` is specified, it should be the keyword `#:text` (the default) or `#:binary`.

28.2.7 open-output-pipe

```
[procedure] (open-output-pipe CMDLINE [MODE])
```

Spawns a subprocess with the command-line string `CMDLINE` and returns a port. Anything written to that port is treated as the input for the process. If `MODE` is specified, it should be the keyword `#:text` (the default) or `#:binary`.

28.2.8 pipe/buf

This variable contains the maximal number of bytes that can be written atomically into a pipe or FIFO.

28.2.9 with-input-from-pipe

28.2.10 with-output-to-pipe

```
[procedure] (with-input-from-pipe CMDLINE THUNK [MODE])
[procedure] (with-output-to-pipe CMDLINE THUNK [MODE])
```

Temporarily set the value of `current-input-port/current-output-port` to a port for a pipe connected to the subprocess named in `CMDLINE` and call the procedure `THUNK` with no arguments. After `THUNK` returns normally the pipe is closed and the standard input-/output port is restored to its previous value and any result values are returned.

```
(with-output-to-pipe
  "gs -dNOPAUSE -sDEVICE=jpeg -dBATCH -sOutputFile=signballs.jpg -g600x600 -q -
  (lambda ()
    (print #<<EOF
%!IOPSC-1993 %%Creator: HAYAKAWA Takashi<xxxxxxxx@xx.xxxxxx.xx.xx>
/C/neg/d/mul/R/rlineto/E/exp/H{{cvx def}repeat}def/T/dup/g/gt/r/roll/J/ifelse 8
H/A/copy(z&v4QX&93r9AxYQOZomQalxS2w!!O&vMYa43d6r93rMYvx2dca!D&cjSnjSnjjS3o!v&6A
X&55SAxM1CD7AjYxTTd62rmxCnTdSST0g&12wECST!&!J0g&D1!&xM0!J0g!l&544dC2Ac96ra!m&3A
F&&vGoGSnCT0g&wDmlvGoS8wpn6wpS2wTCpS1Sd7ov7Uk7o4Qkdw!&Mvlx1S7oZES3w!J!J!Q&7185d
Z&lx1CS9d9nE4!k&X&MY7!&1!J!x&jdnjdS3odS!N&mmx1C2wEc!G&150Nx4!n&2o!j&43r!U&0777d
]&2AY2A776ddT4oS3oSnMVC00VV0RRR45E42063rNz&v7UX&UOzF!F!J![&44ETCnVn!a&1CDN!Y&0M
V1c&j2AYdjmMdjjd!o&1r!M){( )T 0 4 3 r put T(/)g{T(9)g{cvn}{cvi}J}{($)g[]J}J
cvx}forall/moveto/p/floor/w/div/S/add 29 H[{[{]setgray fill}for Y}for showpage
EOF
) ) )
```

28.3 Fifos

28.3.1 create-fifo

```
[procedure] (create-fifo FILENAME [MODE])
```

Creates a FIFO with the name `FILENAME` and the permission bits `MODE`, which defaults to

```
 [procedure] (+ perm/irwxu perm/irwxg perm/irwxo)
```

28.3.2 fifo?

```
[procedure] (fifo? FILENAME)
```

Returns `#t` if the file with the name `FILENAME` names a FIFO.

28.4 File descriptors and low-level I/O

28.4.1 duplicate-fileno

```
[procedure] (duplicate-fileno OLD [NEW])
```

If `NEW` is given, then the file-descriptor `NEW` is opened to access the file with the file-descriptor `OLD`. Otherwise a fresh file-descriptor accessing the same file as `OLD` is returned.

28.4.2 file-close

`[procedure] (file-close FILENO)`

Closes the input/output file with the file-descriptor `FILENO`.

28.4.3 file-open

`[procedure] (file-open FILENAME FLAGS [MODE])`

Opens the file specified with the string `FILENAME` and open-flags `FLAGS` using the C function `open()`. On success a file-descriptor for the opened file is returned. `FLAGS` should be a bitmask containing one or more of the `open/...` values **or**ed together using `bitwise-ior` (or simply added together). The optional `MODE` should be a bitmask composed of one or more permission values like `perm/irusr` and is only relevant when a new file is created. The default mode is `perm/irwxu | perm/irgrp | perm/iroth`.

28.4.4 file-mkstemp

`[procedure] (file-mkstemp TEMPLATE-FILENAME)`

Create a file based on the given `TEMPLATE-FILENAME`, in which the six last characters must be *XXXXXX*. These will be replaced with a string that makes the filename unique. The file descriptor of the created file and the generated filename is returned. See the `mkstemp(3)` manual page for details on how this function works. The template string given is not modified.

Example usage:

```
(let-values (((fd temp-path) (file-mkstemp "/tmp/mytemporary.XXXXXX")))
 (let ((temp-port (open-output-file* fd)))
   (format temp-port "This file is ~A.~%" temp-path)
   (close-output-port temp-port)))
```

28.4.5 file-read

`[procedure] (file-read FILENO SIZE [BUFFER])`

Reads `SIZE` bytes from the file with the file-descriptor `FILENO`. If a string or bytevector is passed in the optional argument `BUFFER`, then this string will be destructively modified to contain the read data. This procedure returns a list with two values: the buffer containing the data and the number of bytes read.

28.4.6 file-select

`[procedure] (file-select READFDLIST WRITEFDLIST [TIMEOUT])`

Waits until any of the file-descriptors given in the lists `READFDLIST` and `WRITEFDLIST` is ready for input or output, respectively. If the optional argument `TIMEOUT` is given and not false, then it should specify the number of seconds after which the wait is to be aborted. This procedure returns two values: the lists of file-descriptors ready for input and output, respectively. `READFDLIST` and **WRITEFDLIST** may also by file-descriptors instead of lists. In this case the returned values are booleans indicating whether input/output is ready by `#t` or `#f` otherwise. You can also pass `#f` as `READFDLIST` or `WRITEFDLIST` argument, which is equivalent to `()`.

28.4.7 file-write

```
[procedure] (file-write FILENO BUFFER [SIZE])
```

Writes the contents of the string or bytevector `BUFFER` into the file with the file-descriptor `FILENO`. If the optional argument `SIZE` is given, then only the specified number of bytes are written.

28.4.8 file-control

```
[procedure] (file-control FILENO COMMAND  [ARGUMENT])
```

Performs the fcntl operation `COMMAND` with the given `FILENO` and optional `ARGUMENT`. The return value is meaningful depending on the `COMMAND`.

28.4.9 fcntl/dupfd

28.4.10 fcntl/getfd

28.4.11 fcntl/setfd

28.4.12 fcntl/getfl

28.4.13 fcntl/setfl

These variables contain commands for `file-control`.

28.4.14 fileno/stdin

28.4.15 fileno/stdout

28.4.16 fileno/stderr

These variables contain file-descriptors for the standard I/O files.

28.4.17 open/rdonly

28.4.18 open/wronly

28.4.19 open/rdwr

28.4.20 open/read

28.4.21 open/write

28.4.22 open/creat

28.4.23 open/append

28.4.24 open/excl

28.4.25 open/noctty

28.4.26 open/nonblock

28.4.27 open/trunc

28.4.28 open/sync

28.4.29 open/fsync

28.4.30 open/binary

28.4.31 open/text

Flags for use with `file-open`.

28.4.32 open-input-file*

28.4.33 open-output-file*

```
[procedure] (open-input-file* FILENO [OPENMODE])
[procedure] (open-output-file* FILENO [OPENMODE])
```

Opens file for the file-descriptor `FILENO` for input or output and returns a port. `FILENO` should be a positive exact integer. `OPENMODE` specifies an additional mode for opening the file (currently only the keyword `#:append` is supported, which opens an output-file for appending).

28.4.34 port->fileno

```
[procedure] (port->fileno PORT)
```

If `PORT` is a file- or tcp-port, then a file-descriptor is returned for this port. Otherwise an error is signaled.

28.5 Retrieving file attributes

28.5.1 file-access-time

28.5.2 file-change-time

28.5.3 file-modification-time

```
[procedure] (file-access-time FILE)
[procedure] (file-change-time FILE)
[procedure] (file-modification-time FILE)
```

Returns time (in seconds) of the last access, modification or change of `FILE`. `FILE` may be a filename or a file-descriptor. If the file does not exist, an error is signaled.

28.5.4 file-stat

```
[procedure] (file-stat FILE [LINK])
```

Returns a 13-element vector with the following contents: inode-number, mode (as with `file-permissions`), number of hard links, uid of owner (as with `file-owner`), gid of owner, size (as with `file-size`) and access-, change- and modification-time (as with `file-access-time`, `file-change-time` and `file-modification-time`, device id, device type (for special file inode, blocksize and blocks allocated. On Windows systems the last 4 values are undefined. If the optional argument `LINK` is given and not `#f`, then the file-statistics vector will be resolved for symbolic links (otherwise symbolic links are not resolved).

28.5.5 file-position

```
[procedure] (file-position FILE)
```

Returns the current file position of `FILE`, which should be a port or a file-descriptor.

28.5.6 file-size

```
[procedure] (file-size FILENAME)
```

Returns the size of the file designated by `FILE`. `FILE` may be a filename or a file-descriptor. If the file does not exist, an error is signaled.

28.5.7 regular-file?

```
[procedure] (regular-file? FILENAME)
```

Returns true, if `FILENAME` names a regular file (not a directory or symbolic link).

28.6 Changing file attributes

28.6.1 file-truncate

`[procedure] (file-truncate FILE OFFSET)`

Truncates the file `FILE` to the length `OFFSET`, which should be an integer. If the file-size is smaller or equal to `OFFSET` then nothing is done. `FILE` should be a filename or a file-descriptor.

28.6.2 set-file-position!

```
[procedure] (set-file-position! FILE POSITION [WHENCE])
[procedure] (set! (file-position FILE) POSITION)
```

Sets the current read/write position of `FILE` to `POSITION`, which should be an exact integer. `FILE` should be a port or a file-descriptor. `WHENCE` specifies how the position is to interpreted and should be one of the values `seek/set, seek/cur` and `seek/end`. It defaults to `seek/set`.

Exceptions: `(exn bounds)`, `(exn i/o file)`

28.7 Processes

28.7.1 current-process-id

`[procedure] (current-process-id)`

Returns the process ID of the current process.

28.7.2 parent-process-id

`[procedure] (parent-process-id)`

Returns the process ID of the parent of the current process.

28.7.3 process-execute

`[procedure] (process-execute PATHNAME [ARGUMENT-LIST [ENVIRONMENT-LIST]])`

Creates a new child process and replaces the running process with it using the C library function `execvp(3)`. If the optional argument `ARGUMENT-LIST` is given, then it should contain a list of strings which are passed as arguments to the subprocess. If the optional argument `ENVIRONMENT-LIST` is supplied, then the library function `execve(2)` is used, and the environment passed in `ENVIRONMENT-LIST` (which should be of the form `("<NAME>=<VALUE>" ...)` is given to the invoked process. Note that `execvp(3)` respects the current setting of the `PATH` environment variable while `execve(3)` does not.

On native Windows, `process-execute` ignores the `ENVIRONMENT-LIST` arguments.

28.7.4 process-fork

```
[procedure] (process-fork [THUNK])
```

Creates a new child process with the UNIX system call `fork()`. Returns either the PID of the child process or 0. If `THUNK` is given, then the child process calls it as a procedure with no arguments and terminates.

28.7.5 process-run

```
[procedure] (process-run PATHNAME [LIST])
```

Creates a new child process using the UNIX system call `fork()` that executes the program given by the string `PATHNAME` using the UNIX system call `execv()`. The PID of the new process is returned. If `LIST` is not specified, then `PATHNAME` is passed to a program named by the environment variable `SHELL` (or `/bin/sh`, if the variable is not defined), so usual argument expansion can take place.

28.7.6 process-signal

```
[procedure] (process-signal PID [SIGNAL])
```

Sends `SIGNAL` to the process with the id `PID` using the UNIX system call `kill()`. `SIGNAL` defaults to the value of the variable `signal/term`.

28.7.7 process-wait

```
[procedure] (process-wait [PID [NOHANG]])
```

Suspends the current process until the child process with the id `PID` has terminated using the UNIX system call `waitpid()`. If `PID` is not given, then this procedure waits for any child process. If `NOHANG` is given and not `#f` then the current process is not suspended. This procedure returns three values:

- `PID` or 0, if `NOHANG` is true and the child process has not terminated yet
- `#t` if the process exited normally or `#f` otherwise
- either the exit status, if the process terminated normally or the signal number that terminated/stopped the process.

28.7.8 process

```
[procedure] (process COMMANDLINE [ARGUMENTLIST [ENVIRONMENT]])
```

Passes the string `COMMANDLINE` to the host-system's shell that is invoked as a subprocess and returns three values: an input port from which data written by the sub-process can be read, an output port from which any data written to will be received as input in the sub-process and the process-id of the started sub-process. Blocking reads and writes to or from the ports returned by `process` only block the current thread, not other threads executing concurrently.

If `ARGUMENTLIST` is given, then the invocation of the subprocess is not done via the shell, but directly. The arguments are directly passed to `process-execute` (as is `ENVIRONMENT`). Not using the shell may be preferrable for security reasons.

On native Windows the `ARGUMENTLIST` and `ENVIRONMENT` arguments are ignored.

28.7.9 process*

```
[procedure] (process* COMMANDLINE [ARGUMENTLIST [ENVIRONMENT]])
```

Like `process` but returns 4 values: an input port from which data written by the sub-process can be read, an output port from which any data written to will be received as input in the sub-process, the process-id of the started sub-process, and an input port from which data written by the sub-process to `stderr` can be read.

28.7.10 sleep

```
[procedure] (sleep SECONDS)
```

Puts the process to sleep for `SECONDS`. Returns either 0 if the time has completely elapsed, or the number of remaining seconds, if a signal occurred.

28.8 Hard and symbolic links

28.8.1 symbolic-link?

```
[procedure] (symbolic-link? FILENAME)
```

Returns true, if `FILENAME` names a symbolic link.

28.8.2 create-symbolic-link

```
[procedure] (create-symbolic-link OLDNAME NEWNAME)
```

Creates a symbolic link with the filename `NEWNAME` that points to the file named `OLDNAME`.

28.8.3 read-symbolic-link

```
[procedure] (read-symbolic-link FILENAME)
```

Returns the filename to which the symbolic link `FILENAME` points.

28.8.4 file-link

```
[procedure] (file-link OLDNAME NEWNAME)
```

Creates a hard link from `OLDNAME` to `NEWNAME` (both strings).

28.9 Permissions

28.9.1 file-owner

```
[procedure] (file-owner FILE)
```

Returns the user-id of `FILE`. `FILE` may be a filename or a file-descriptor.

28.9.2 file-permissions

```
[procedure] (file-permissions FILE)
```

Returns the permission bits for `FILE`. You can test this value by performing bitwise operations on the result and the `perm/...` values. `FILE` may be a filename or a file-descriptor.

28.9.3 file-read-access?

28.9.4 file-write-access?

28.9.5 file-execute-access?

```
[procedure] (file-read-access? FILENAME)
[procedure] (file-write-access? FILENAME)
[procedure] (file-execute-access? FILENAME)
```

These procedures return `#t` if the current user has read, write or execute permissions on the file named `FILENAME`.

28.9.6 change-file-mode

```
[procedure] (change-file-mode FILENAME MODE)
```

Changes the current file mode of the file named `FILENAME` to `MODE` using the `chmod()` system call. The `perm/...` variables contain the various permission bits and can be combinded with the `bitwise-ior` procedure.

28.9.7 change-file-owner

```
[procedure] (change-file-owner FILENAME UID GID)
```

Changes the owner information of the file named `FILENAME` to the user- and group-ids `UID` and `GID` (which should be exact integers) using the `chown()` system call.

28.9.8 current-user-id

28.9.9 current-group-id

28.9.10 current-effective-user-id

28.9.11 current-effective-group-id

```
[procedure] (current-user-id)
[procedure] (current-group-id)
[procedure] (current-effective-user-id)
[procedure] (current-effective-group-id)
```

Return the user- and group-ids of the current process.

28.9.12 process-group-id

```
[procedure] (process-group-id PID)
```

Returns the process group ID of the process specified by `PID`.

28.9.13 group-information

```
[procedure] (group-information GROUP)
```

If `GROUP` specifies a valid group-name or group-id, then this procedure returns a list of four values: the group-name, the encrypted group password, the group ID and a list of the names of all group members. If no group with the given name or ID exists, then `#f` is returned.

28.9.14 get-groups

```
[procedure] (get-groups)
```

Returns a list with the supplementary group IDs of the current user.

28.9.15 set-groups!

```
[procedure] (set-groups! GIDLIST)
```

Sets the supplementrary group IDs of the current user to the IDs given in the list `GIDLIST`.

Only the superuser may invoke this procedure.

28.9.16 initialize-groups

```
[procedure] (initialize-groups USERNAME BASEGID)
```

Sets the supplementrary group IDs of the current user to the IDs from the user with name `USERNAME` (a string), including `BASEGID`.

Only the superuser may invoke this procedure.

28.9.17 perm/irusr

28.9.18 perm/iwusr

28.9.19 perm/ixusr

28.9.20 perm/irgrp

28.9.21 perm/iwgrp

28.9.22 perm/ixgrp

28.9.23 perm/iroth

28.9.24 perm/iwoth

28.9.25 perm/ixoth

28.9.26 perm/irwxu

28.9.27 perm/irwxg

28.9.28 perm/irwxo

28.9.29 perm/isvtx

28.9.30 perm/isuid

28.9.31 perm/isgid

These variables contain permission bits as used in `change-file-mode`.

28.9.32 set-user-id!

```
[procedure] (set-user-id! UID)
[setter] (set! (current-user-id) UID)
```

Sets the effective user id of the current process to `UID`, which should be a positive integer.

28.9.33 set-group-id!

```
[procedure] (set-group-id! GID)
[setter] (set! (current-group-id) GID)
```

Sets the effective group id of the current process to `GID`, which should be a positive integer.

28.9.34 set-process-group-id!

```
[procedure] (set-process-group-id! PID PGID)
[setter] (set! (process-group-id PID) PGID)
```

Sets the process group ID of the process specifed by `PID` to `PGID`.

28.9.35 user-information

```
[procedure] (user-information USER)
```

If `USER` specifes a valid username (as a string) or user ID, then the user database is consulted and a list of 7 values are returned: the user-name, the encrypted password, the user ID, the group ID, a user-specific string, the home directory and the default shell. If no user with this name or ID can be found, then `#f` is returned.

28.9.36 create-session

```
[procedure] (create-session)
```

Creates a new session if the calling process is not a process group leader and returns the session ID.

28.10 Record locking

28.10.1 file-lock

```
[procedure] (file-lock PORT [START [LEN]])
```

Locks the file associated with `PORT` for reading or writing (according to whether `PORT` is an input- or output-port). `START` specifies the starting position in the file to be locked and defaults to 0. `LEN` specifies the length of the portion to be locked and defaults to `#t`, which means the complete file. `file-lock` returns a *lock*-object.

28.10.2 file-lock/blocking

```
[procedure] (file-lock/blocking PORT [START [LEN]])
```

Similar to `file-lock`, but if a lock is held on the file, the current process blocks (including all threads) until the lock is released.

28.10.3 file-test-lock

```
[procedure] (file-test-lock PORT [START [LEN]])
```

Tests whether the file associated with `PORT` is locked for reading or writing (according to whether `PORT` is an input- or output-port) and returns either `#f` or the process-id of the locking process.

28.10.4 file-unlock

```
[procedure] (file-unlock LOCK)
```

Unlocks the previously locked portion of a file given in `LOCK`.

28.11 Signal handling

28.11.1 set-alarm!

```
[procedure] (set-alarm! SECONDS)
```

Sets an internal timer to raise the `signal/alrm` after `SECONDS` are elapsed. You can use the `set-signal-handler!` procedure to write a handler for this signal.

28.11.2 set-signal-handler!

```
[procedure] (set-signal-handler! SIGNUM PROC)
```

Establishes the procedure of one argument `PROC` as the handler for the signal with the code `SIGNUM`. `PROC` is called with the signal number as its sole argument. If the argument `PROC` is `#f` then any signal handler will be removed.

Note that is is unspecified in which thread of execution the signal handler will be invoked.

28.11.3 signal-handler

```
[procedure] (signal-handler SIGNUM)
```

Returns the signal handler for the code `SIGNUM` or `#f`.

28.11.4 set-signal-mask!

```
[procedure] (set-signal-mask! SIGLIST)
```

Sets the signal mask of the current process to block all signals given in the list `SIGLIST`. Signals masked in that way will not be delivered to the current process.

28.11.5 signal-mask

```
[procedure] (signal-mask)
```

Returns the signal mask of the current process.

28.11.6 signal-masked?

```
[procedure] (signal-masked? SIGNUM)
```

Returns whether the signal for the code `SIGNUM` is currently masked.

28.11.7 signal-mask!

```
[procedure] (signal-mask! SIGNUM)
```

Masks (blocks) the signal for the code `SIGNUM`.

28.11.8 signal-unmask!

```
[procedure] (signal-unmask! SIGNUM)
```

Unmasks (unblocks) the signal for the code `SIGNUM`.

28.11.9 signal/term

28.11.10 signal/kill

28.11.11 signal/int

28.11.12 signal/hup

28.11.13 signal/fpe

28.11.14 signal/ill

28.11.15 signal/segv

28.11.16 signal/abrt

28.11.17 signal/trap

28.11.18 signal/quit

28.11.19 signal/alrm

28.11.20 signal/vtalrm

28.11.21 signal/prof

28.11.22 signal/io

28.11.23 signal/urg

28.11.24 signal/chld

28.11.25 signal/cont

28.11.26 signal/stop

28.11.27 signal/tstp

28.11.28 signal/pipe

28.11.29 signal/xcpu

28.11.30 signal/xfsz

28.11.31 signal/usr1

28.11.32 signal/usr2

28.11.33 signal/winch

These variables contain signal codes for use with `process-signal`, `set-signal-handler!`, `signal-handler`, `signal-masked?`, `signal-mask!`, or `signal-unmask!`.

28.12 Environment access

28.12.1 current-environment

```
[procedure] (current-environment)
```

Returns a association list of the environment variables and their current values.

28.12.2 setenv

```
[procedure] (setenv VARIABLE VALUE)
```

Sets the environment variable named `VARIABLE` to `VALUE`. Both arguments should be strings. If the variable is not defined in the environment, a new definition is created.

28.12.3 unsetenv

```
[procedure] (unsetenv VARIABLE)
```

Removes the definition of the environment variable `VARIABLE` from the environment of the current process. If the variable is not defined, nothing happens.

28.13 Memory mapped I/O

28.13.1 memory-mapped-file?

```
[pocedure] (memory-mapped-file? X)
```

Returns `#t`, if `X` is an object representing a memory mapped file, or `#f` otherwise.

28.13.2 map-file-to-memory

```
[procedure] (map-file-to-memory ADDRESS LEN PROTECTION FLAG FILENO [OFFSET])
```

Maps a section of a file to memory using the C function `mmap()`. `ADDRESS` should be a foreign pointer object or `#f`; `LEN` specifies the size of the section to be mapped; `PROTECTION` should be one or more of the flags `prot/read, prot/write, prot/exec` or `prot/none` **bitwise-ior**ed together; `FLAG` should be one or more of the flags `map/fixed, map/shared, map/private, map/anonymous` or `map/file`; `FILENO` should be the file-descriptor of the mapped file. The optional argument `OFFSET` gives the offset of the section of the file to be mapped and defaults to 0. This procedure returns an object representing the mapped file section. The procedure `move-memory!` can be used to access the mapped memory.

28.13.3 memory-mapped-file-pointer

`[procedure] (memory-mapped-file-pointer MMAP)`

Returns a machine pointer to the start of the memory region to which the file is mapped.

28.13.4 unmap-file-from-memory

`[procedure] (unmap-file-from-memory MMAP [LEN])`

Unmaps the section of a file mapped to memory using the C function `munmap()`. `MMAP` should be a mapped file as returned by the procedure `map-file-to-memory`. The optional argument `LEN` specifies the length of the section to be unmapped and defaults to the complete length given when the file was mapped.

28.14 Date and time routines

28.14.1 seconds->local-time

`[procedure] (seconds->local-time SECONDS)`

Breaks down the time value represented in `SECONDS` into a 10 element vector of the form `#(seconds minutes hours mday month year wday yday dstflag timezone)`, in the following format:

seconds (0)
: the number of seconds after the minute (0 - 59)

minutes (1)
: the number of minutes after the hour (0 - 59)

hours (2)
: the number of hours past midnight (0 - 23)

mday (3)
: the day of the month (1 - 31)

month (4)
: the number of months since january (0 - 11)

year (5)
: the number of years since 1900

wday (6)
: the number of days since Sunday (0 - 6)

yday (7)
: the number of days since January 1 (0 - 365)

dstflag (8)
: a flag that is true if Daylight Saving Time is in effect at the time described.

timezone (9)
: the difference between UTC and the latest local standard time, in seconds west of UTC.

28.14.2 local-time->seconds

```
[procedure] (local-time->seconds VECTOR)
```

Converts the ten-element vector `VECTOR` representing the time value relative to the current timezone into the number of seconds since the first of January, 1970 UTC.

28.14.3 local-timezone-abbreviation

```
[procedure] (local-timezone-abbrevtiation)
```

Returns the abbreviation for the local timezone as a string.

28.14.4 seconds->string

```
[procedure] (seconds->string SECONDS)
```

Converts the local time represented in `SECONDS` into a string of the form `"Tue May 21 13:46:22 1991\n"`.

28.14.5 seconds->utc-time

```
[procedure] (seconds->utc-time SECONDS)
```

Similar to `seconds->local-time`, but interpretes `SECONDS` as UTC time.

28.14.6 utc-time->seconds

```
[procedure] (utc-time->seconds VECTOR)
```

Converts the ten-element vector `VECTOR` representing the UTC time value into the number of seconds since the first of January, 1970 UTC.

28.14.7 time->string

```
[procedure] (time->string VECTOR)
```

Converts the broken down time represented in the 10 element vector `VECTOR` into a string of the form `"Tue May 21 13:46:22 1991\n"`.

28.15 Raw exit

28.15.1 _exit

```
[procedure] (_exit [CODE])
```

Exits the current process without flushing any buffered output (using the C function `_exit`). Note that the `exit-handler` is not called when this procedure is invoked. The optional return-code `CODE` defaults to `0`.

28.16 ERRNO values

28.16.1 errno/perm

28.16.2 errno/noent

28.16.3 errno/srch

28.16.4 errno/intr

28.16.5 errno/io

28.16.6 errno/noexec

28.16.7 errno/badf

28.16.8 errno/child

28.16.9 errno/nomem

28.16.10 errno/acces

28.16.11 errno/fault

28.16.12 errno/busy

28.16.13 errno/notdir

28.16.14 errno/isdir

28.16.15 errno/inval

28.16.16 errno/mfile

28.16.17 errno/nospc

28.16.18 errno/spipe

28.16.19 errno/pipe

28.16.20 errno/again

28.16.21 errno/rofs

28.16.22 errno/exist

28.16.23 errno/wouldblock

These variables contain error codes as returned by `errno`.

28.17 Finding files

28.17.1 find-files

```
[procedure] (find-files DIRECTORY PREDICATE [ACTION [IDENTITY [LIMIT]]])
```

Recursively traverses the contents of `DIRECTORY` (which should be a string) and invokes the procedure `ACTION` for all files for which the procedure `PREDICATE` is true. `PREDICATE` may me a procedure of one argument or a regular-expression string. `ACTION` should be a procedure of two arguments: the currently encountered file and the result of the previous invocation of `ACTION`, or, if this is the first invocation, the value of `IDENTITY`. `ACTION` defaults to `cons`, `IDENTITY` defaults to `()`. `LIMIT` should a procedure of one argument that is called for each nested directory and which should return true, if that directory is to be traversed recursively. `LIMIT` may also be an exact integer that gives the maximum recursion depth. A depth of `0` means the files in the specified directory are traversed but not any nested directories. `LIMIT` may also be `#f` (the default), which is equivalent to `(constantly #t)`.

Note that `ACTION` is called with the full pathname of each file, including the directory prefix.

28.18 Getting the hostname and system information

28.18.1 get-host-name

```
[procedure] (get-host-name)
```

Returns the hostname of the machine that this process is running on.

28.18.2 system-information

```
[procedure] (system-information)
```

Invokes the UNIX system call `uname()` and returns a list of 5 values: system-name, node-name, OS release, OS version and machine.

28.19 Setting the file buffering mode

28.19.1 set-buffering-mode!

`[procedure] (set-buffering-mode! PORT MODE [BUFSIZE])`

Sets the buffering-mode for the file associated with `PORT` to `MODE`, which should be one of the keywords `#:full`, `#:line` or `#:none`. If `BUFSIZE` is specified it determines the size of the buffer to be used (if any).

28.20 Terminal ports

28.20.1 terminal-name

`[procedure] (terminal-name PORT)`

Returns the name of the terminal that is connected to `PORT`.

28.20.2 terminal-port?

`[procedure] (terminal-port? PORT)`

Returns `#t` if `PORT` is connected to a terminal and `#f` otherwise.

28.21 How Scheme procedures relate to UNIX C functions

`change-directory`

chdir

`change-file-mode`

chmod

`change-file-owner`

chown

`create-directory`

mkdir

`create-fifo`

mkfifo

`create-pipe`

pipe

`create-session`

setsid

`create-symbolic-link`

link

`current-directory`

curdir

`current-effective-groupd-id`

getegid

`current-effective-user-id`

geteuid

`current-group-id`

getgid

`current-parent-id`

getppid

`current-process-id`

getpid

`current-user-id`

getuid

`delete-directory`

rmdir

`duplicate-fileno`

dup/dup2

`_exit`

_exit

`file-close`

close

`file-access-time`

stat

`file-change-time`

stat

`file-modification-time`

stat

`file-execute-access?`

access

`file-open`

open

`file-lock`

fcntl

`file-position`

ftell/lseek

`file-read`

read

`file-read-access?`

access

`file-select`

select

`file-control`

fcntl

`file-stat`

stat

`file-test-lock`

fcntl

`file-truncate`

truncate/ftruncate

`file-unlock`

fcntl

`file-write`

write

`file-write-access?`

access

`get-groups`

getgroups

`get-host-name`

gethostname

`initialize-groups`

initgroups

`local-time->seconds`

mktime

`local-timezone-abbreviation`

localtime

`map-file-to-memory`

mmap

`open-input-file*`

fdopen

`open-output-file*`

fdopen

`open-input-pipe`

popen

`open-output-pipe`

popen

`port->fileno`

fileno

`process-execute`

execvp

`process-fork`

fork

`process-group-id`

getpgid

`process-signal`

kill

`process-wait`

waitpid

`close-input-pipe`

pclose

`close-output-pipe`

pclose

`read-symbolic-link`

readlink

`seconds->local-time`

localtime

`seconds->string`

ctime

`seconds->utc-time`

gmtime

`set-alarm!`

alarm

`set-buffering-mode!`

setvbuf

`set-file-position!`

fseek/seek

`set-groups!`

setgroups

`set-signal-mask!`

sigprocmask

`set-group-id!`

setgid

`set-process-group-id!`

setpgid

`set-user-id!`

setuid

`set-root-directory!`

chroot

`setenv`

setenv/putenv

`sleep`

sleep

`system-information`

uname

`terminal-name`

ttyname

`terminal-port?`

isatty

`time->string`

asctime

`unsetenv`

putenv

`unmap-file-from-memory`

munmap

```
user-information
```

getpwnam/getpwuid

```
utc-time->seconds
```

timegm

28.22 Windows specific notes

The following definitions are not supported for native Windows builds (compiled with the Microsoft tools or with MingW):

```
open/noctty  open/nonblock  open/fsync  open/sync
perm/isvtx  perm/isuid  perm/isgid
file-select file-control
signal/... (except signal/term, signal/int, signal/fpe, signal/ill, signal/segv
set-signal-mask!  signal-mask  signal-masked?  signal-mask!  signal-unmask!
user-information  group-information  get-groups  set-groups!  initialize-groups
errno/wouldblock
change-file-owner
current-user-id  current-group-id  current-effective-user-id  current-effective
set-user-id!  set-group-id!
create-session
process-group-id  set-process-group-id!
create-symbolic-link  read-symbolic-link
file-truncate
file-lock  file-lock/blocking  file-unlock  file-test-lock
create-fifo  fifo?
prot/...
map/...
map-file-to-memory  unmap-file-from-memory  memory-mapped-file-pointer  memory-
set-alarm!
terminal-port?  terminal-name
process-fork  process-signal
parent-process-id
set-root-directory!
utc-time->seconds local-timezone-abbreviation
```

Additionally, the following definitions are only available for Windows:

28.22.1 spawn/overlay

28.22.2 spawn/wait

28.22.3 spawn/nowait

28.22.4 spawn/nowaito

28.22.5 spawn/detach

These variables contains special flags that specify the exact semantics of `process-spawn`: `spawn/overlay` replaces the current process with the new one. `spawn/wait` suspends execution of the current process until the spawned process returns. `spawn/nowait` does the opposite (`spawn/nowaito` is identical, according to the Microsoft documentation) and runs the process asynchronously. `spawn/detach` runs the new process in the background, without being attached to a console.

28.22.6 process-spawn

```
[procedure] (process-spawn MODE FILENAME ARGUMENT ...)
```

Creates and runs a new process with the given filename and command-line arguments. `MODE` specifies how exactly the process should be executed and must be one or more of the `spawn/...` flags defined above.

Previous: Unit srfi-18

Next: Unit utils

29 Unit utils

This unit contains file/pathname oriented procedures, apropos, plus acts as a "grab bag" for procedures without a good home, and which don't have to be available by default (as compared to the extras unit.

This unit uses the `extras` and `regex` units.

29.1 Environment Query

29.1.1 apropos

```
[procedure] (apropos SYMBOL-PATTERN [ENVIRONMENT])
```

Displays symbols matching `SYMBOL-PATTERN` in the `ENVIRONMENT` on the `(current-output-port)`.

`SYMBOL-PATTERN`
: A symbol, string, or regex. When symbol or string substring matching is performed.

`ENVIRONMENT`
: An environment. When missing the `(interaction-environment)` is assumed.

29.1.2 apropos-list

```
[procedure] (apropos-list SYMBOL-PATTERN [ENVIRONMENT])
```

Like `apropos` but returns a list of matching symbols.

29.2 Pathname operations

29.2.1 absolute-pathname?

```
[procedure] (absolute-pathname? PATHNAME)
```

Returns `#t` if the string `PATHNAME` names an absolute pathname, and returns `#f` otherwise.

29.2.2 decompose-pathname

```
[procedure] (decompose-pathname PATHNAME)
```

Returns three values: the directory-, filename- and extension-components of the file named by the string `PATHNAME`. For any component that is not contained in `PATHNAME`, `#f` is returned.

29.2.3 make-pathname

29.2.4 make-absolute-pathname

```
[procedure] (make-pathname DIRECTORY FILENAME [EXTENSION [SEPARATOR]])
[procedure] (make-absolute-pathname DIRECTORY FILENAME [EXTENSION [SEPARATOR]])
```

Returns a string that names the file with the components `DIRECTORY, FILENAME` and (optionally) `EXTENSION` with `SEPARATOR` being the directory separation indicator (usually `/` on UNIX systems and `\` on Windows, defaulting to whatever platform this is running on). `DIRECTORY` can be `#f` (meaning no directory component), a string or a list of strings. `FILENAME` and `EXTENSION` should be strings or `#f`. `make-absolute-pathname` returns always an absolute pathname.

29.2.5 pathname-directory

```
[procedure] (pathname-directory PATHNAME)
```

29.2.6 pathname-file

```
[procedure] (pathname-file PATHNAME)
```

29.2.7 pathname-extension

```
[procedure] (pathname-extension PATHNAME)
```

Accessors for the components of `PATHNAME`. If the pathname does not contain the accessed component, then `#f` is returned.

29.2.8 pathname-replace-directory

```
[procedure] (pathname-replace-directory PATHNAME DIRECTORY)
```

29.2.9 pathname-replace-file

```
[procedure] (pathname-replace-file PATHNAME FILENAME)
```

29.2.10 pathname-replace-extension

```
[procedure] (pathname-replace-extension PATHNAME EXTENSION)
```

Return a new pathname with the specified component of `PATHNAME` replaced by a new value.

29.2.11 pathname-strip-directory

```
[procedure] (pathname-strip-directory PATHNAME)
```

29.2.12 pathname-strip-extension

```
[procedure] (pathname-strip-extension PATHNAME)
```

Return a new pathname with the specified component of `PATHNAME` stripped.

29.3 Temporary files

29.3.1 create-temporary-file

```
[procedure] (create-temporary-file [EXTENSION])
```

Creates an empty temporary file and returns its pathname. If `EXTENSION` is not given, then `.tmp` is used. If the environment variable `TMPDIR, TEMP` or `TMP` is set, then the pathname names a file in that directory.

29.4 Deleting a file without signalling an error

29.4.1 delete-file*

```
[procedure] (delete-file* FILENAME)
```

If the file `FILENAME` exists, it is deleted and `#t` is returned. If the file does not exist, nothing happens and `#f` is returned.

29.5 Iterating over input lines and files

29.5.1 for-each-line

```
[procedure] (for-each-line PROCEDURE [PORT])
```

Calls `PROCEDURE` for each line read from `PORT` (which defaults to the value of `(current-input-port)`. The argument passed to `PORCEDURE` is a string with the contents of the line, excluding any line-terminators. When all input has been read from the port, `for-each-line` returns some unspecified value.

29.5.2 for-each-argv-line

```
[procedure] (for-each-argv-line PROCEDURE)
```

```
Opens each file listed on the command line in order, passing one line
```

at a time into `PROCEDURE`. The filename `-` is interpreted as `(current-input-port)`. If no arguments are given on the command line it again uses the value of `(current-input-port)`. During execution of `PROCEDURE`, the current input port will be correctly bound to the current input source.

This code will act as a simple Unix cat(1) command:

```
(for-each-argv-line print)
```

29.5.3 port-for-each

```
[procedure] (port-for-each FN THUNK)
```

Apply `FN` to successive results of calling the zero argument procedure `THUNK` until it returns `#!eof`, discarding the results.

29.5.4 port-map

```
[procedure] (port-map FN THUNK)
```

Apply `FN` to successive results of calling the zero argument procedure `THUNK` until it returns `#!eof`, returning a list of the collected results.

29.6 Executing shell commands with formatstring and error checking

29.6.1 system*

```
[procedure] (system* FORMATSTRING ARGUMENT1 ...)
```

Similar to `(system (sprintf FORMATSTRING ARGUMENT1 ...))`, but signals an error if the invoked program should return a nonzero exit status.

29.7 Reading a file's contents

29.7.1 read-all

```
[procedure] (read-all [FILE-OR-PORT])
```

If `FILE-OR-PORT` is a string, then this procedure returns the contents of the file as a string. If `FILE-OR-PORT` is a port, all remaining input is read and returned as a string. The port is not closed. If no argument is provided, input will be read from the port that is the current value of `(current-input-port)`.

29.8 Miscellaneous handy things

29.8.1 shift!

```
[procedure] (shift! LIST [DEFAULT])
```

Returns the car of `LIST` (or `DEFAULT` if `LIST` is empty) and replaces the car of `LIST` with it's cadr and the cdr with the cddr. If `DEFAULT` is not given, and the list is empty, `#f` is returned. An example might be clearer, here:

```
(define lst '(1 2 3))
(shift! lst)             ==> 1, lst is now (2 3)
```

The list must at least contain 2 elements.

29.8.2 unshift!

```
[procedure] (unshift! X PAIR)
```

Sets the car of `PAIR` to `X` and the cdr to its cddr. Returns `PAIR`:

```
(define lst '(2))
(unshift! 99 lst)      ; lst is now (99 2)
```

Previous: Unit posix

Next: Unit tcp

30 Unit tcp

This unit provides basic facilities for communicating over TCP sockets. The socket interface should be mostly compatible to the one found in PLT Scheme.

This unit uses the `extras` unit.

All errors related to failing network operations will raise a condition of kind `(exn i/o network)`.

30.1 tcp-listen

```
[procedure] (tcp-listen TCPPORT [BACKLOG [HOST]])
```

Creates and returns a TCP listener object that listens for connections on `TCPPORT`, which should be an exact integer. `BACKLOG` specifies the number of maximally pending connections (and defaults to 4). If the optional argument `HOST` is given and not `#f`, then only incoming connections for the given host (or IP) are accepted.

30.2 tcp-listener?

```
[procedure] (tcp-listener? X)
```

Returns `#t` if `X` is a TCP listener object, or `#f` otherwise.

30.3 tcp-close

```
[procedure] (tcp-close LISTENER)
```

Reclaims any resources associated with `LISTENER`.

30.4 tcp-accept

```
[procedure] (tcp-accept LISTENER)
```

Waits until a connection is established on the port on which `LISTENER` is listening and returns two values: an input- and output-port that can be used to communicate with the remote process.

Note: this operation and any I/O on the ports returned will not block other running threads.

30.5 tcp-accept-ready?

```
[procedure] (tcp-accept-ready? LISTENER)
```

Returns `#t` if there are any connections pending on `LISTENER`, or `#f` otherwise.

30.6 tcp-listener-port

`[procedure] (tcp-listener-port LISTENER)`

Returns the port number assigned to `LISTENER` (If you pass `0` to `tcp-listen`, then the system will choose a port-number for you).

30.7 tcp-listener-fileno

`[procedure] (tcp-listener-port LISTENER)`

Returns the file-descriptor associated with `LISTENER`.

30.8 tcp-connect

`[procedure] (tcp-connect HOSTNAME [TCPPORT])`

Establishes a client-side TCP connection to the machine with the name `HOSTNAME` (a string) at `TCPPORT` (an exact integer) and returns two values: an input- and output-port for communicating with the remote process.

Note: any I/O on the ports returned will not block other running threads.

30.9 tcp-addresses

`[procedure] (tcp-addresses PORT)`

Returns two values for the input- or output-port `PORT` (which should be a port returned by either `tcp-accept` or `tcp-connect`): the IP address of the local and the remote machine that are connected over the socket associated with `PORT`. The returned addresses are strings in `XXX.XXX.XXX.XXX` notation.

30.10 tcp-port-numbers

`[procedure] (tcp-port-numbers PORT)`

Returns two values for the input- or output-port `PORT` (which should be a port returned by either `tcp-accept` or `tcp-connect`): the TCP port numbers of the local and the remote machine that are connected over the socket associated with `PORT`.

30.11 tcp-abandon-port

```
[procedure] (tcp-abandon-port PORT)
```

Marks the socket port `PORT` as abandoned. This is mainly useful to close down a port without breaking the connection.

30.12 tcp-buffer-size

```
[parameter] tcp-buffer-size
```

Sets the size of the output buffer. By default no output-buffering for TCP output is done, but to improve performance by minimizing the number of TCP packets, buffering may be turned on by setting this parameter to an exact integer greater zero. A buffer size of zero or `#f` turns buffering off. The setting of this parameter takes effect at the time when the I/O ports for a particular socket are created, i.e. when `tcp-connect` or `tcp-accept` is called.

Note that since output is not immediately written to the associated socket, you may need to call `flush-output`, once you want the output to be transmitted. Closing the output port will flush automatically.

31 Example

A very simple example follows. Say we have the two files `client.scm` and `server.scm`:

```
; client.scm
(declare (uses tcp))
(define-values (i o) (tcp-connect "localhost" 4242))
(write-line "Good Bye!" o)
(print (read-line i))
```

```
; server.scm
(declare (uses tcp))
(define l (tcp-listen 4242))
(define-values (i o) (tcp-accept l))
(write-line "Hello!" o)
(print (read-line i))
(close-input-port i)
(close-output-port o)
```

```
% csc server.scm
% csc client.scm
% ./server &
% ./client
Good Bye!
Hello!
```

Previous: Unit utils

Next: Unit lolevel

32 Unit lolevel

This unit provides a number of handy low-level operations. **Use at your own risk.**

This unit uses the `srfi-4` and `extras` units.

32.1 Foreign pointers

32.1.1 address->pointer

`[procedure] (address->pointer ADDRESS)`

Creates a new foreign pointer object initialized to point to the address given in the integer `ADDRESS`.

32.1.2 allocate

`[procedure] (allocate BYTES)`

Returns a pointer to a freshly allocated region of static memory. This procedure could be defined as follows:

```
(define allocate (foreign-lambda c-pointer "malloc" integer))
```

32.1.3 free

`[procedure] (free POINTER)`

Frees the memory pointed to by `POINTER`. This procedure could be defined as follows:

```
(define free (foreign-lambda c-pointer "free" integer))
```

32.1.4 null-pointer

`[procedure] (null-pointer)`

Another way to say `(address->pointer 0)`.

32.1.5 null-pointer?

`[procedure] (null-pointer? PTR)`

Returns `#t` if `PTR` contains a `NULL` pointer, or `#f` otherwise.

32.1.6 object->pointer

`[procedure] (object->pointer X)`

Returns a pointer pointing to the Scheme object X, which should be a non-immediate object. Note that data in the garbage collected heap moves during garbage collection.

32.1.7 pointer?

`[procedure] (pointer? X)`

Returns `#t` if `X` is a foreign pointer object, and `#f` otherwise.

32.1.8 pointer=?

`[procedure] (pointer=? PTR1 PTR2)`

Returns `#t` if the pointer-like objects `PTR1` and `PTR2` point to the same address.

32.1.9 pointer->address

`[procedure] (pointer->address PTR)`

Returns the address, to which the pointer `PTR` points.

32.1.10 pointer->object

`[procedure] (pointer->object PTR)`

Returns the Scheme object pointed to by the pointer `PTR`.

32.1.11 pointer-offset

`[procedure] (pointer-offset PTR N)`

Returns a new pointer representing the pointer `PTR` increased by `N`.

32.1.12 pointer-u8-ref

`[procedure] (pointer-u8-ref PTR)`

Returns the unsigned byte at the address designated by `PTR`.

32.1.13 pointer-s8-ref

`[procedure] (pointer-s8-ref PTR)`

Returns the signed byte at the address designated by `PTR`.

32.1.14 pointer-u16-ref

`[procedure] (pointer-u16-ref PTR)`

Returns the unsigned 16-bit integer at the address designated by `PTR`.

32.1.15 pointer-s16-ref

`[procedure] (pointer-s16-ref PTR)`

Returns the signed 16-bit integer at the address designated by `PTR`.

32.1.16 pointer-u32-ref

`[procedure] (pointer-u32-ref PTR)`

Returns the unsigned 32-bit integer at the address designated by `PTR`.

32.1.17 pointer-s32-ref

`[procedure] (pointer-s32-ref PTR)`

Returns the signed 32-bit integer at the address designated by `PTR`.

32.1.18 pointer-f32-ref

`[procedure] (pointer-f32-ref PTR)`

Returns the 32-bit float at the address designated by `PTR`.

32.1.19 pointer-f64-ref

```
[procedure] (pointer-f64-ref PTR)
```

Returns the 64-bit double at the address designated by `PTR`.

32.1.20 pointer-u8-set!

```
[procedure] (pointer-u8-set! PTR N)
[procedure] (set! (pointer-u8-ref PTR) N)
```

Stores the unsigned byte `N` at the address designated by `PTR`.

32.1.21 pointer-s8-set!

```
[procedure] (pointer-s8-set! PTR N)
[procedure] (set! (pointer-s8-ref PTR) N)
```

Stores the signed byte `N` at the address designated by `PTR`.

32.1.22 pointer-u16-set!

```
[procedure] (pointer-u16-set! PTR N)
[procedure] (set! (pointer-u16-ref PTR) N)
```

Stores the unsigned 16-bit integer `N` at the address designated by `PTR`.

32.1.23 pointer-s16-set!

```
[procedure] (pointer-s16-set! PTR N)
[procedure] (set! (pointer-s16-ref PTR) N)
```

Stores the signed 16-bit integer `N` at the address designated by `PTR`.

32.1.24 pointer-u32-set!

```
[procedure] (pointer-u32-set! PTR N)
[procedure] (set! (pointer-u32-ref PTR) N)
```

Stores the unsigned 32-bit integer `N` at the address designated by `PTR`.

32.1.25 pointer-s32-set!

```
[procedure] (pointer-s32-set! PTR N)
[procedure] (set! (pointer-s32-ref PTR) N)
```

Stores the 32-bit integer `N` at the address designated by `PTR`.

32.1.26 pointer-f32-set!

```
[procedure] (pointer-f32-set! PTR N)
[procedure] (set! (pointer-f32-ref PTR) N)
```

Stores the 32-bit floating-point number `N` at the address designated by `PTR`.

32.1.27 pointer-f64-set!

```
[procedure] (pointer-f64-set! PTR N)
[procedure] (set! (pointer-f64-ref PTR) N)
```

Stores the 64-bit floating-point number `N` at the address designated by `PTR`.

32.1.28 align-to-word

```
[procedure] (align-to-word PTR-OR-INT)
```

Accepts either a machine pointer or an integer as argument and returns a new pointer or integer aligned to the native word size of the host platform.

32.2 Tagged pointers

Tagged pointers are foreign pointer objects with an extra tag object.

32.2.1 tag-pointer

```
[procedure] (tag-pointer PTR TAG)
```

Creates a new tagged pointer object from the foreign pointer `PTR` with the tag `TAG`, which may an arbitrary Scheme object.

32.2.2 tagged-pointer?

```
[procedure] (tagged-pointer? X TAG)
```

Returns `#t`, if `X` is a tagged pointer object with the tag `TAG` (using an `eq?` comparison), or `#f` otherwise.

32.2.3 pointer-tag

```
[procedure] (pointer-tag PTR)
```

If `PTR` is a tagged pointer object, its tag is returned. If `PTR` is a normal, untagged foreign pointer object `#f` is returned. Otherwise an error is signalled.

32.3 Extending procedures with data

32.3.1 extend-procedure

```
[procedure] (extend-procedure PROCEDURE X)
```

Returns a copy of the procedure `PROCEDURE` which contains an additional data slot initialized to `X`. If `PROCEDURE` is already an extended procedure, then its data slot is changed to contain `X` and the same procedure is returned.

32.3.2 extended-procedure?

```
[procedure] (extended-procedure? PROCEDURE)
```

Returns `#t` if `PROCEDURE` is an extended procedure, or `#f` otherwise.

32.3.3 procedure-data

```
[procedure] (procedure-data PROCEDURE)
```

Returns the data object contained in the extended procedure `PROCEDURE`, or `#f` if it is not an extended procedure.

32.3.4 set-procedure-data!

```
[procedure] (set-procedure-data! PROCEDURE X)
```

Changes the data object contained in the extended procedure `PROCEDURE` to `X`.

```
(define foo
  (letrec ((f (lambda () (procedure-data x)))
           (x #f) )
    (set! x (extend-procedure f 123))
    x) )
(foo)                                              ==> 123
(set-procedure-data! foo 'hello)
(foo)                                              ==> hello
```

32.4 Bytevectors

32.4.1 byte-vector

`[procedure] (byte-vector FIXNUM ...)`

Returns a freshly allocated byte-vector with `FIXNUM ...` as its initial contents.

32.4.2 byte-vector?

`[procedure] (byte-vector? X)`

Returns `#t` if `X` is a byte-vector object, or `#f` otherwise.

32.4.3 byte-vector-fill!

`[procedure] (byte-vector-fill! BYTE-VECTOR N)`

Sets each element of `BYTE-VECTOR` to `N`, which should be an exact integer.

32.4.4 byte-vector->list

`[procedure] (byte-vector->list BYTE-VECTOR)`

Returns a list with elements taken from `BYTE-VECTOR`.

32.4.5 byte-vector->string

`[procedure] (byte-vector->string BYTE-VECTOR)`

Returns a string with the contents of `BYTE-VECTOR`.

32.4.6 byte-vector-length

```
[procedure] (byte-vector-length BYTE-VECTOR)
```

Returns the number of elements in `BYTE-VECTOR`.

32.4.7 byte-vector-ref

```
[procedure] (byte-vector-ref BYTE-VECTOR INDEX)
```

Returns the byte at the `INDEX`th position of `BYTE-VECTOR`.

32.4.8 byte-vector-set!

```
[procedure] (byte-vector-set! BYTE-VECTOR INDEX N)
[procedure] (set! (byte-vector-ref BYTE-VECTOR INDEX) N)
```

Sets the byte at the `INDEX`th position of `BYTE-VECTOR` to the value of the exact integer `n`.

32.4.9 list->byte-vector

```
[procedure] (list->byte-vector LIST)
```

Returns a byte-vector with elements taken from `LIST`, where the elements of `LIST` should be exact integers.

32.4.10 make-byte-vector

```
[procedure] (make-byte-vector SIZE [INIT])
```

Creates a new byte-vector of size `SIZE`. If `INIT` is given, then it should be an exact integer with which every element of the byte-vector is initialized.

32.4.11 make-static-byte-vector

```
[procedure] (make-static-byte-vector SIZE [INIT])
```

As `make-byte-vector`, but allocates the byte-vector in storage that is not subject to garbage collection. To free the allocated memory, one has to call `object-release` explicitly.

Exceptions: `(exn bounds)`, `(exn runtime)`

32.4.12 static-byte-vector->pointer

```
[procedure] (static-byte-vector->pointer PBYTE-VECTOR)
```

Returns a pointer object pointing to the data in the statically allocated byte-vector `PBYTE-VECTOR`.

32.4.13 string->byte-vector

```
[procedure] (string->byte-vector STRING)
```

Returns a byte-vector with the contents of `STRING`.

32.5 Data in unmanaged memory

32.5.1 object-evict

```
[procedure] (object-evict X [ALLOCATOR])
```

Copies the object `X` recursively into the memory pointed to by the foreign pointer object returned by `ALLOCATOR`, which should be a procedure of a single argument (the number of bytes to allocate). The freshly copied object is returned. This facility allows moving arbitrary objects into static memory, but care should be taken when mutating evicted data: setting slots in evicted vector-like objects to non-evicted data is not allowed. It **is** possible to set characters/bytes in evicted strings or byte-vectors, though. It is advisable **not** to evict ports, because they might be mutated by certain file-operations. `object-evict` is able to handle circular and shared structures, but evicted symbols are no longer unique: a fresh copy of the symbol is created, so

```
(define x 'foo)
(define y (object-evict 'foo))
y                              ==> foo
(eq? x y)                      ==> #f
(define z (object-evict '(bar bar)))
(eq? (car z) (cadr z))         ==> #t
```

The `ALLOCATOR` defaults to `allocate`.

32.5.2 object-evict-to-location

```
[procedure] (object-evict-to-location X PTR [LIMIT])
```

As `object-evict` but moves the object at the address pointed to by the machine pointer `PTR`. If the number of copied bytes exceeds the optional `LIMIT` then an error is signalled (specifically a composite condition of types `exn` and `evict`. The latter provides a `limit` property which holds the exceeded limit. Two values are returned: the evicted object and a new pointer pointing to the first free address after the evicted object.

32.5.3 object-evicted?

```
[procedure] (object-evicted? X)
```

Returns `#t` if `X` is a non-immediate evicted data object, or `#f` otherwise.

32.5.4 object-size

```
[procedure] (object-size X)
```

Returns the number of bytes that would be needed to evict the data object `X`.

32.5.5 object-release

```
[procedure] (object-release X [RELEASER])
```

Frees memory occupied by the evicted object `X` recursively. `RELEASER` should be a procedure of a single argument (a foreign pointer object to the static memory to be freed) and defaults to `free`.

32.5.6 object-unevict

```
[procedure] (object-unevict X [FULL])
```

Copies the object `X` and nested objects back into the normal Scheme heap. Symbols are re-interned into the symbol table. Strings and byte-vectors are **not** copied, unless `FULL` is given and not `#f`.

32.6 Locatives

A *locative* is an object that points to an element of a containing object, much like a *pointer* in low-level, imperative programming languages like *C*. The element can be accessed and changed indirectly, by performing access or change operations on the locative. The container object can be computed by calling the `location->object` procedure.

Locatives may be passed to foreign procedures that expect pointer arguments. The effect of creating locatives for evicted data (see `object-evict`) is undefined.

32.6.1 make-locative

```
[procedure] (make-locative EXP [INDEX])
```

Creates a locative that refers to the element of the non-immediate object `EXP` at position `INDEX`. `EXP` may be a vector, pair, string, byte-vector, SRFI-4 number-vector, or record. `INDEX` should be a fixnum. `INDEX` defaults to 0.

32.6.2 make-weak-locative

```
[procedure] (make-weak-locative EXP [INDEX])
```

Creates a *weak* locative. Even though the locative refers to an element of a container object, the container object will still be reclaimed by garbage collection if no other references to it exist.

32.6.3 locative?

```
[procedure] (locative? X)
```

Returns `#t` if `X` is a locative, or `#f` otherwise.

32.6.4 locative-ref

```
[procedure] (locative-ref LOC)
```

Returns the element to which the locative `LOC` refers. If the containing object has been reclaimed by garbage collection, an error is signalled.

32.6.5 locative-set!

```
[procedure] (locative-set! LOC X)
[procedure] (set! (locative-ref LOC) X)
```

Changes the element to which the locative `LOC` refers to `X`. If the containing object has been reclaimed by garbage collection, an error is signalled.

32.6.6 locative->object

```
[procedure] (locative->object LOC)
```

Returns the object that contains the element referred to by `LOC` or `#f` if the container has been reclaimed by garbage collection.

32.7 Accessing toplevel variables

32.7.1 global-bound?

```
[procedure] (global-bound? SYMBOL)
```

Returns `#t`, if the global (*toplevel*) variable with the name `SYMBOL` is bound to a value, or `#f` otherwise.

32.7.2 global-ref

```
[procedure] (global-ref SYMBOL)
```

Returns the value of the global variable `SYMBOL`. If no variable under that name is bound, an error is signalled.

Note that it is not possible to access a toplevel binding with `global-ref` or `global-set!` if it has been hidden in compiled code via `(declare (hide ...))`, or if the code has been compiled in `block` mode.

32.7.3 global-set!

```
[procedure] (global-set! SYMBOL X)
[procedure] (set! (global-ref SYMBOL) X)
```

Sets the global variable named `SYMBOL` to the value `X`.

32.8 Low-level data access

32.8.1 block-ref

```
[procedure] (block-ref BLOCK INDEX)
```

Returns the contents of the `INDEX`th slot of the object `BLOCK`. `BLOCK` may be a vector, record structure, pair or symbol.

32.8.2 block-set!

```
[procedure] (block-set! BLOCK INDEX X)
[procedure] (set! (block-ref BLOCK INDEX) X)
```

Sets the contents of the `INDEX`th slot of the object `BLOCK` to the value of `X`. `BLOCK` may be a vector, record structure, pair or symbol.

32.8.3 object-copy

```
[procedure] (object-copy X)
```

Copies `X` recursively and returns the fresh copy. Objects allocated in static memory are copied back into garbage collected storage.

32.8.4 make-record-instance

```
[procedure] (make-record-instance SYMBOL ARG1 ...)
```

Returns a new instance of the record type `SYMBOL`, with its slots initialized to `ARG1 ...`. To illustrate:

```
(define-record point x y)
```

expands into something quite similar to:

```
(begin
  (define (make-point x y)
    (make-record-instance 'point x y) )
  (define (point? x)
    (and (record-instance? x)
         (eq? 'point (block-ref x 0)) ) )
  (define (point-x p) (block-ref p 1))
  (define (point-x-set! p x) (block-set! p 1 x))
  (define (point-y p) (block-ref p 2))
  (define (point-y-set! p y) (block-set! p 1 y)) )
```

32.8.5 move-memory!

```
[procedure] (move-memory! FROM TO [BYTES])
```

Copies `BYTES` bytes of memory from `FROM` to `TO`. `FROM` and `TO` may be strings, primitive byte-vectors, SRFI-4 byte-vectors (see: @ref{Unit srfi-4}), memory mapped files, foreign pointers (as obtained from a call to `foreign-lambda`, for example) or locatives. if `BYTES` is not given and the size of the source or destination operand is known then the maximal number of bytes will be copied. Moving memory to the storage returned by locatives will cause havoc, if the locative refers to containers of non-immediate data, like vectors or pairs.

32.8.6 number-of-bytes

```
[procedure] (number-of-bytes BLOCK)
```

Returns the number of bytes that the object `BLOCK` contains. `BLOCK` may be any non-immediate value.

32.8.7 number-of-slots

```
[procedure] (number-of-slots BLOCK)
```

Returns the number of slots that the object `BLOCK` contains. `BLOCK` may be a vector, record structure, pair or symbol.

32.8.8 record-instance?

```
[procedure] (record-instance? X)
```

Returns `#t` if `X` is an instance of a record type. See also: `make-record-instance`.

32.8.9 record->vector

```
[procedure] (record->vector BLOCK)
```

Returns a new vector with the type and the elements of the record `BLOCK`.

32.9 Procedure-call- and variable reference hooks

32.9.1 set-invalid-procedure-call-handler!

```
[procedure] (set-invalid-procedure-call-handler! PROC)
```

Sets an internal hook that is invoked when a call to an object other than a procedure is executed at runtime. The procedure `PROC` will in that case be called with two arguments: the object being called and a list of the passed arguments.

```
;;; Access sequence-elements as in ARC:

(set-invalid-procedure-call-handler!
  (lambda (proc args)
    (cond [(string? proc) (apply string-ref proc args)]
          [(vector? proc) (apply vector-ref proc args)]
          [else (error "call of non-procedure" proc)] ) ) )

("hello" 4)    ==>  #\o
```

This facility does not work in code compiled with the *unsafe* setting.

32.9.2 unbound-variable-value

`[procedure] (unbound-variable-value [X])`

Defines the value that is returned for unbound variables. Normally an error is signalled, use this procedure to override the check and return `X` instead. To set the default behavior (of signalling an error), call `unbound-variable-value` with no arguments.

This facility does not work in code compiled with the *unsafe* setting.

32.10 Magic

32.10.1 object-become!

`[procedure] (object-become! ALIST)`

Changes the identity of the value of the car of each pair in `ALIST` to the value of the cdr. Both values may not be immediate (i.e. exact integers, characters, booleans or the empty list).

```
(define x "i used to be a string")
(define y '#(and now i am a vector))
(object-become! (list (cons x y)))
x                                    ==> #(and now i am a vector)
y                                    ==> #(and now i am a vector)
(eq? x y)                            ==> #t
```

Note: this operation invokes a major garbage collection.

The effect of using `object-become!` on evicted data (see `object-evict`) is undefined.

Previous: Unit tcp

Next: Interface to external functions and variables

33 Interface to external functions and variables

- Accessing external objects
- Foreign type specifiers
- Embedding
- Callbacks
- Locations
- Other support procedures
- C interface

Previous: Supported language

Next: chicken-setup

34 Accessing external objects

34.1 foreign-code

```
[syntax] (foreign-code STRING ...)
```

Executes the embedded C/C++ code `STRING ...`, which should be a sequence of C statements, which are executed and return an unspecified result.

```
(foreign-code "doSomeInitStuff();")     =>  #<unspecified>
```

Code wrapped inside `foreign-code` may not invoke callbacks into Scheme.

34.2 foreign-value

```
[syntax] (foreign-value STRING TYPE)
```

Evaluates the embedded C/C++ expression `STRING`, returning a value of type given in the foreign-type specifier `TYPE`.

```
(print (foreign-value "my_version_string" c-string))
```

34.3 foreign-declare

```
[syntax] (foreign-declare STRING ...)
```

Include given strings verbatim into header of generated file.

34.4 define-foreign-type

```
[syntax] (define-foreign-type NAME TYPE [ARGCONVERT [RETCONVERT]])
```

Defines an alias for `TYPE` with the name `NAME` (a symbol). `TYPE` may be a type-specifier or a string naming a C type. The namespace of foreign type specifiers is separate from the normal Scheme namespace. The optional arguments `ARGCONVERT` and `RETCONVERT` should evaluate to procedures that map argument- and result-values to a value that can be transformed to `TYPE`:

```
(define-foreign-type char-vector
  nonnull-c-string
  (compose list->string vector->list)
  (compose list->vector string->list) )

(define strlen
  (foreign-lambda int "strlen" char-vector) )

(strlen '#(#\a #\b #\c))                      ==> 3
```

```
(define memset
  (foreign-lambda char-vector "memset" char-vector char int) )

(memset '#(#_ #_ #_) #\X 3)                    ==> #(#\X #\X #\X)
```

Foreign type-definitions are only visible in the compilation-unit in which they are defined, so use `include` to use the same definitions in multiple files.

34.5 define-foreign-variable

`[syntax] (define-foreign-variable NAME TYPE [STRING])`

Defines a foreign variable of name `NAME` (a symbol). `STRING` should be the real name of a foreign variable or parameterless macro. If `STRING` is not given, then the variable name `NAME` will be converted to a string and used instead. All references and assignments (via `set!`) are modified to correctly convert values between Scheme and C representation. This foreign variable can only be accessed in the current compilation unit, but the name can be lexically shadowed. Note that `STRING` can name an arbitrary C expression. If no assignments are performed, then `STRING` doesn't even have to specify an lvalue.

```
#>
enum { abc=3, def, ghi };
<#

(define-macro (define-simple-foreign-enum . items)
  `(begin
     ,@(map (match-lambda
              [(name realname) `(define-foreign-variable ,name int ,realname)]
              [name `(define-foreign-variable ,name int)] )
     items) ) )

(define-simple-foreign-enum abc def ghi)

ghi                               ==> 5
```

34.6 define-foreign-record

`[syntax] (define-foreign-record NAME [DECL ...] SLOT ...)`

Defines accessor procedures for a C structure definition. `NAME` should either be a symbol or a list of the form `(TYPENAME FOREIGNNAME)`. If `NAME` is a symbol, then a C declaration will be generated that defines a C struct named `struct NAME`. If `NAME` is a list, then no struct declaration will be generated and `FOREIGNNAME` should name an existing C record type. A foreign-type specifier named `NAME` (or `TYPENAME`) will be defined as a pointer to the given C structure. A `SLOT` definition should be a list of one of the following forms:

```
(TYPE SLOTNAME)
```

or

```
(TYPE SLOTNAME SIZE)
```

The latter form defines an array of `SIZE` elements of the type `TYPE` embedded in the structure. For every slot, the following accessor procedures will be generated:

34.6.1 TYPENAME-SLOTNAME

```
(TYPENAME-SLOTNAME FOREIGN-RECORD-POINTER [INDEX])
```

A procedure of one argument (a pointer to a C structure), that returns the slot value of the slot `SLOTNAME`. If a `SIZE` has been given in the slot definition, then an additional argument `INDEX` is required that specifies the index of an array-element.

34.6.2 TYPENAME-SLOTNAME-set!

```
(TYPENAME-SLOTNAME-set! FOREIGN-RECORD-POINTER [INXDEX] VALUE)
```

A procedure of two arguments (a pointer to a C structure) and a value, that sets the slot value of the slot `SLOTNAME` in the structure. If a `SIZE` has been given in the slot definition, then an additional argument `INDEX` is required for the array index.

If a slot type is of the form `(const ...)`, then no setter procedure will be generated. Slots of the types `(struct ...)` or `(union ...)` are accessed as pointers to the embedded struct (or union) and no setter will be generated.

Additionally, special record-declarations (`DECL ...`) may be given, where each declaration consists of a list of the form `(KEYWORD ARGUMENT ...)`. The available declarations are:

34.6.3 constructor

```
(constructor: NAME)
```

Generate a constructor-procedure with no arguments that has the name `NAME` (a symbol) that returns a pointer to a structure of this type. The storage will be allocated with `malloc(3)`.

34.6.4 destructor

```
(destructor: NAME)
```

Generate a destructor function with the name `NAME` that takes a pointer to a structure of this type as its single argument and releases the storage with `free(3)`. If the argument is `#f`, the destructor procedure does nothing.

34.6.5 rename

```
(rename: EXPRESSION)
```

Evaluates `EXPRESSION` at compile-/macro-expansion-time and applies the result, which should be a procedure, to the string-representation of the name of each accessor-procedure generated. Another (or the same) string should be returned, which in turn is taken as the actual name of the accessor.

An example:

```
(require-for-syntax 'srfi-13)

(define-foreign-record Some_Struct
  (rename: (compose string-downcase (cut string-translate <> "_" "-")))
  (constructor: make-some-struct)
  (destructor: free-some-struct)
  (int xCoord)
  (int yCoord) )
```

will generate the following procedures:

```
(make-some-struct)              --> C-POINTER
(free-some-struct C-POINTER)

(some-struct-xcoord C-POINTER)  --> NUMBER
(some-struct-ycoord C-POINTER)  --> NUMBER

(some-struct-xcoord-set! C-POINTER NUMBER)
(some-struct-ycoord-set! C-POINTER NUMBER)
```

34.7 define-foreign-enum

```
[syntax] (define-foreign-enum TYPENAME ITEM ...)
```

Defines a foreign type (as with `define-foreign-type`) that maps the elements of a C/C++ enum (or a enum-like list of constants) to and from a set of symbols. `TYPENAME` names a foreign type that converts a symbol argument from the set `ITEM ...` into the appropriate enum value when passed as an argument to a foreign function. A list of symbols passed as an argument will be combined using `bitwise-ior`. An empty list will be passed as 0 (zero). Results of the enum type are automatically converted into a symbol (note that combinations are not supported in this case). `TYPENAME` may alternatively be a list of the form `(SCHEMENAME REALTYPE)` - in this case `REALTYPE` designates the native type used. The default native type is `"TYPENAME"`. Additionally two procedures are generated named `SCHEMENAME->number` and `number->SCHEMENAME` which take one argument and convert a symbol (or a list of symbols) into its numeric value and vice versa.

Here a heavily contrived example:

```
#>
enum foo { a_foo = 4, b_foo, c_foo };
enum foo bar(enum foo x) { printf("%d\n", x); return b_foo; }
<#

(define-foreign-enum (foo (enum "foo")) a_foo b_foo (c c_foo))
```

```
(define bar (foreign-lambda foo bar foo))

(pp (bar '()))
(pp (bar 'a_foo))
(pp (bar '(b_foo c)))
```

34.8 foreign-lambda

```
[syntax] (foreign-lambda RETURNTYPE NAME ARGTYPE ...)
```

Represents a binding to an external routine. This form can be used in the position of an ordinary `lambda` expression. `NAME` specifies the name of the external procedure and should be a string or a symbol.

34.9 foreign-lambda*

```
[syntax] (foreign-lambda* RETURNTYPE ((ARGTYPE VARIABLE) ...) STRING ...)
```

Similar to `foreign-lambda`, but instead of generating code to call an external function, the body of the C procedure is directly given in `STRING ...`:

```
(define my-strlen
  (foreign-lambda* int ((c-string str))
    "int n = 0; while(*(str++)) ++n; C_return(n);") )

(my-strlen "one two three")             ==> 13
```

For obscure technical reasons you should use the `C_return` macro instead of the normal `return` statement to return a result from the foreign lambda body as some cleanup code has to be run before execution commences in the calling code.

34.10 foreign-safe-lambda

```
[syntax] (foreign-safe-lambda RETURNTYPE NAME ARGTYPE ...)
```

This is similar to `foreign-lambda`, but also allows the called function to call Scheme functions and allocate Scheme data-objects. See Callbacks.

34.11 foreign-safe-lambda*

```
[syntax] (foreign-safe-lambda* RETURNTYPE ((ARGTYPE VARIABLE)...) STRING ...)
```

This is similar to `foreign-lambda*`, but also allows the called function to call Scheme functions and allocate Scheme data-objects. See Callbacks.

34.12 foreign-primitive

```
[syntax] (foreign-primitive [RETURNTYPE] ((ARGTYPE VARIABLE) ...) STRING ...)
```

This is also similar to `foreign-lambda*` but the code will be executed in a *primitive* CPS context, which means it will not actually return, but call it's continuation on exit. This means that code inside this form may allocate Scheme data on the C stack (the *nursery*) with `C_alloc` (see below). If the `RETURNTYPE` is omitted it defaults to `void`. You can return multiple values inside the body of the `foreign-primitive` form by calling this C function:

```
C_values(N + 2, C_SCHEME_UNDEFINED, C_k, X1, ...)
```

where `N` is the number of values to be returned, and `X1, ...` are the results, which should be Scheme data objects. When returning multiple values, the return-type should be omitted.

Previous: Interface to external functions and variables

Next: Foreign type specifiers

35 Foreign type specifiers

Here is a list of valid foreign type specifiers:

35.1 scheme-object

An arbitrary Scheme data object (immediate or non-immediate).

35.2 bool

As argument: any value (`#f` is false, anything else is true). As result: anything different from 0 and the `NULL`-pointer is `#t`.

35.3 byte unsigned-byte

A byte.

35.4 char unsigned-char

A character.

35.5 short unsigned-short

A short integer number.

35.6 int unsigned-int int32 unsigned-int32

An small integer number in fixnum range (at least 30 bit).

35.7 integer unsigned-integer integer32 unsigned-integer32 integer64

Either a fixnum or a flonum in the range of a (unsigned) machine *int* or with 32/64 bit width.

35.8 long unsigned-long

Either a fixnum or a flonum in the range of a (unsigned) machine *long* or with 32 bit width.

35.9 float double

A floating-point number. If an exact integer is passed as an argument, then it is automatically converted to a float.

35.10 number

A floating-point number. Similar to `double`, but when used as a result type, then either an exact integer or a floating-point number is returned, depending on whether the result fits into an exact integer or not.

35.11 symbol

A symbol, which will be passed to foreign code as a zero-terminated string. When declared as the result of foreign code, the result should be a string and a symbol with the same name will be interned in the symbol table (and returned to the caller).

35.12 scheme-pointer

An untyped pointer to the contents of a non-immediate Scheme object (not allowed as return type). The value `#f` is also allowed and is passed as a `NULL` pointer. Don't confuse this type with `(pointer ...)` which means something different (a machine-pointer object).

35.13 nonnull-scheme-pointer

As `pointer`, but guaranteed not to be `#f`. Don't confuse this type with `(nonnull-pointer ...)` which means something different (a machine-pointer object).

35.14 c-pointer

An untyped operating-system pointer or a locative. The value `#f` is also allowed and is passed as a `NULL` pointer. If uses as the type of a return value, a `NULL` pointer will be returned as `#f`.

35.15 nonnull-c-pointer

As `c-pointer`, but guaranteed not to be `#f/NULL`.

35.16 scheme-or-c-pointer

An untyped pointer. If the argument is a machine-pointer object, the wrapped C pointer will be passed. If it is any other object, then this type is treated as `scheme-pointer`.

Not allowed as a result type.

35.17 [nonnull-] byte-vector

A byte-vector object, passed as a pointer to its contents. Arguments of type `byte-vector` may optionally be `#f`, which is passed as a NULL pointer. This is not allowed as a return type.

35.18 [nonnull-] u8vector [nonnull-] u16vector [nonnull-] u32vector [nonnull-] s8vector [nonnull-] s16vector [nonnull-] s32vector [nonnull-] f32vector [nonnull-] f64vector

A SRFI-4 number-vector object, passed as a pointer to its contents. Arguments of type `byte-vector` may optionally be `#f`, which is passed as a NULL pointer. These are not allowed as return types.

35.19 c-string

A C string (zero-terminated). The value `#f` is also allowed and is passed as a `NULL` pointer. If uses as the type of a return value, a `NULL` pointer will be returned as `#f`. Note that the string is copied (with a zero-byte appended) when passed as an argument to a foreign function. Also a return value of this type is copied into garbage collected memory.

35.20 nonnull-c-string

As `c-string`, but guaranteed not to be `#f/NULL`.

35.21 [nonnull-] c-string*

Similar to `[nonnull-]c-string`, but if used as a result-type, the pointer returned by the foreign code will be freed (using the C-libraries `free()`) after copying. This type specifier is not valid as a result type for callbacks defined with `define-external`.

35.22 void

Specifies an undefined return value. Not allowed as argument type.

35.23 (const TYPE)

The foreign type `TYPE` with an additional `const` specifier.

35.24 (enum NAME)

An enumeration type. Handled internally as an `integer`.

35.25 (pointer TYPE) (c-pointer TYPE)

An operating-system pointer or a locative to an object of `TYPE`.

35.26 (nonnull-pointer TYPE) (nonnull-c-pointer TYPE)

As `(pointer TYPE)`, but guaranteed not to be `#f/NULL`.

35.27 (ref TYPE)

A C++ reference type. Reference types are handled the same way as pointers inside Scheme code.

35.28 (struct NAME)

A struct of the name `NAME`, which should be a string. Structs can not be directly passed as arguments to foreign function, neither can they be result values. Pointers to structs are allowed, though.

35.29 (template TYPE ARGTYPE ...)

A C++ template type. For example `vector<int>` would be specified as `(template "vector" int)`. Template types can not be directly passed as arguments or returned as results.

35.30 (union NAME)

A union of the name `NAME`, which should be a string. Unions can not be directly passed as arguments to foreign function, neither can they be result values. Pointers to unions are allowed, though.

35.31 (instance CNAME SCHEMECLASS)

A pointer to a C++ class instance. `CNAME` should designate the name of the C++ class, and `SCHEMECLASS` should be the class that wraps the instance pointer. Normally `SCHEMECLASS` should be a subclass of `<c++-object>`.

35.32 (instance-ref CNAME SCHEMECLASS)

A reference to a C++ class instance.

35.33 (function RESULTTYPE (ARGUMENTTYPE1 ... [...]) [CALLCONV])

A function pointer. `CALLCONV` specifies an optional calling convention and should be a string. The meaning of this string is entirely platform dependent. The value `#f` is also allowed and is passed as a `NULL` pointer.

36 Mappings

Foreign types are mapped to C types in the following manner:

bool	int
[unsigned-]char	[unsigned] char
[unsigned-]short	[unsigned] short
[unsigned-]int	[unsigned] int
[unsigned-]integer	[unsigned] int
[unsigned-]long	[unsigned] long
float	float
double	double
number	double
[nonnull-]pointer	void *
[nonnull-]c-pointer	void *
[nonnull-]byte-vector	unsigned char *
[nonnull-]u8vector	unsigned char *
[nonnull-]s8vector	char *
[nonnull-]u16vector	unsigned short *
[nonnull-]s16vector	short *
[nonnull-]u32vector	uint32_t *
[nonnull-]s32vector	int32_t *
[nonnull-]f32vector	float *
[nonnull-]f64vector	double *
[nonnull-]c-string	char *
symbol	char *
void	void
([nonnull-]pointer TYPE)	TYPE *
(enum NAME)	enum NAME
(struct NAME)	struct NAME
(ref TYPE)	TYPE &
(template T1 T2 ...)	T1<T2, ...>
(union NAME)	union NAME
(function RTYPE (ATYPE ...) [CALLCONV])	[CALLCONV] RTYPE (*)(ATYPE, ...)
(instance CNAME SNAME)	CNAME *
(instance-ref CNAME SNAME)	CNAME &

Previous: Accessing external objects

Next: Embedding

37 Embedding

Compiled Scheme files can be linked with C code, provided the Scheme code was compiled in *embedded* mode by passing `-DC_EMBEDDED` to the C compiler (this will disable generation of a `main()` function). `csc` will do this, when given the `-embedded` option. Alternatively pass `-embedded` to `csc`.

The following C API is available:

37.1 CHICKEN_parse_command_line

```
[C function] void CHICKEN_parse_command_line (int argc, char *argv[], int *heap
```

Parse the programs command-line contained in `argc` and `argv` and return the heap-, stack- and symbol table limits given by runtime options of the form `-:...`, or choose default limits. The library procedure `argv` can access the command-line only if this function has been called by the containing application.

37.2 CHICKEN_initialize

```
[C function] int CHICKEN_initialize (int heap, int stack, int symbols, void *to
```

Initializes the Scheme execution context and memory. `heap` holds the number of bytes that are to be allocated for the secondary heap. `stack` holds the number of bytes for the primary heap. `symbols` contains the size of the symbol table. Passing `0` to one or more of these parameters will select a default size. `toplevel` should be a pointer to the toplevel entry point procedure. You should pass `C_toplevel` here. In any subsequent call to `CHICKEN_run` you can simply pass `NULL`. Calling this function more than once has no effect. If enough memory is available and initialization was successful, then `1` is returned, otherwise this function returns `0`.

37.3 CHICKEN_run

```
[C function] C_word CHICKEN_run (void *toplevel)
```

Starts the Scheme program. Call this function once to execute all toplevel expressions in your compiled Scheme program. If the runtime system was not initialized before, then `CHICKEN_initialize` is called with default sizes. `toplevel` is the toplevel entry-point procedure, you usually pass `C_toplevel` here. The result value is the continuation that can be used to re-invoke the Scheme code from the point after it called `return-to-host` (see below).

If you just need a Scheme interpreter, you can also pass `CHICKEN_default_toplevel` as the toplevel procedure, which just uses the default library units.

Once `CHICKEN_run` has been called, Scheme code is executing until all toplevel expressions have been evaluated or until `return-to-host` is called inside the Scheme program.

37.4 return-to-host

```
[procedure] (return-to-host)
```

Exits the Scheme code and returns to the invoking context that called `CHICKEN_run` or `CHICKEN_continue`.

After `return-to-host` has been executed and once `CHICKEN_run` returns, you can invoke callbacks which have been defined with `define-external`. The `eval` library unit also provides *boilerplate* callbacks, that simplify invoking Scheme code embedded in a C or C++ application a lot.

37.5 CHICKEN_eval

```
[C macro] int CHICKEN_eval (C_word exp, C_word *result)
```

Evaluates the Scheme object passed in `exp`, writing the result value to `result`. The return value is 1 if the operation succeeded, or 0 if an error occurred. Call `CHICKEN_get_error_message` to obtain a description of the error.

37.6 CHICKEN_eval_string

```
[C macro] int CHICKEN_eval_string (char *str, C_word *result)
```

Evaluates the Scheme expression passed in the string `str`, writing the result value to `result`.

37.7 CHICKEN_eval_to_string

```
[C macro] int CHICKEN_eval_to_string (C_word exp, char *result, int size)
```

Evaluates the Scheme expression passed in `exp`, writing a textual representation of the result into `result`. `size` should specify the maximal size of the result string.

37.8 CHICKEN_eval_string_to_string

```
[C macro] int CHICKEN_eval_string_to_string (char *str, char *result, int size)
```

Evaluates the Scheme expression passed in the string `str`, writing a textual representation of the result into `result`. `size` should specify the maximal size of the result string.

37.9 CHICKEN_apply

```
[C macro] int CHICKEN_apply (C_word func, C_word args, C_word *result)
```

Applies the procedure passed in `func` to the list of arguments `args`, writing the result value to `result`.

37.10 CHICKEN_apply_to_string

`[C macro] int CHICKEN_apply_to_string (C_word func, C_word args, char *result,`

Applies the procedure passed in `func` to the list of arguments `args`, writing a textual representation of the result into `result`.

37.11 CHICKEN_read

`[C macro] int CHICKEN_read (char *str, C_word *result)`

Reads a Scheme object from the string `str`, writing the result value to `result`.

37.12 CHICKEN_load

`[C macro] int CHICKEN_load (char *filename)`

Loads the Scheme file `filename` (either in source form or compiled).

37.13 CHICKEN_get_error_message

`[C macro] void CHICKEN_get_error_message (char *result, int size)`

Returns a textual description of the most recent error that occurred in executing embedded Scheme code.

37.14 CHICKEN_yield

`[C macro] int CHICKEN_yield (int *status)`

If threads have been spawned during earlier invocations of embedded Scheme code, then this function will run the next scheduled thread for one complete time-slice. This is useful, for example, inside an *idle* handler in a GUI application with background Scheme threads. Note that the `srfi-18` library unit has to be linked in for this.

An example:

```
% cat x.scm
;;; x.scm

(define (bar x) (gc) (* x x))

(define-external (baz (int i)) double
  (sqrt i))
(return-to-host)
```

```
% cat y.c
/* y.c */

#include <chicken.h>
#include <assert.h>

extern double baz(int);

int main() {
  char buffer[ 256 ];
  int status;
  C_word val = C_SCHEME_UNDEFINED;
  C_word *data[ 1 ];

  data[ 0 ] = &val;

  CHICKEN_run(C_toplevel);

  status = CHICKEN_read("(bar 99)", &val);
  assert(status);

  C_gc_protect(data, 1);

  printf("data: %08x\n", val);

  status = CHICKEN_eval_string_to_string("(bar)", buffer, 255);
  assert(!status);

  CHICKEN_get_error_message(buffer, 255);
  printf("ouch: %s\n", buffer);

  status = CHICKEN_eval_string_to_string("(bar 23)", buffer, 255);
  assert(status);

  printf("-> %s\n", buffer);
  printf("data: %08x\n", val);

  status = CHICKEN_eval_to_string(val, buffer, 255);
  assert(status);
  printf("-> %s\n", buffer);

  printf("->` %g\n", baz(22));

  return 0;
}

% csc x.scm y.c -embedded
```

It is also possible to re-enter the computation following the call to `return-to-host` by calling `CHICKEN_continue`:

37.15 CHICKEN_continue

```
[C function] C_word CHICKEN_continue (C_word k)
```

Re-enters Scheme execution. `k` is the continuation received from the previous invocation of `CHICKEN_run` or `CHICKEN_continue`. When `return-to-host` is called again, this function returns another continuation that can be used to restart again.

If you invoke callbacks prior to calling `CHICKEN_continue`, make sure that the continuation is not reclaimed by garbage collection. This can be avoided by using `C_gc_protect` or gc-roots.

Another example:

```
% cat x.scm
(require-extension srfi-18)

(define m (make-mutex))

(define (t)
  (mutex-lock! m)
  (thread-sleep! 1)
  (print (thread-name (current-thread)))
  (mutex-unlock! m)
  (t) )

(thread-start! (make-thread t 'PING!))
(thread-start! (make-thread t 'PONG!))

(let loop ()
  (return-to-host)
  (thread-yield!)
  (loop) )

% cat y.c
#include <chicken.h>

int main()
{
  C_word k = CHICKEN_run(C_toplevel);

  for(;;)
    k = CHICKEN_continue(k);

  return 0;
}

% csc x.scm y.c -embedded
```

It is advisable not to mix repeated uses of `CHICKEN_continue`/`return-to-host` (as in the example above) with callbacks. Once `return-to-host` is invoked, the runtime system and any Scheme code executed prior to the invocation is initialized and can be conveniently used via callbacks.

A simpler interface For handling GC-safe references to Scheme data are the so called *gc-roots*:

37.16 CHICKEN_new_gc_root

```
[C function] void* CHICKEN_new_gc_root ()
```

Returns a pointer to a *GC root*, which is an object that holds a reference to a Scheme value that will always be valid, even after a garbage collection. The content of the gc root is initialized to an unspecified value.

37.17 CHICKEN_delete_gc_root

```
[C function] void CHICKEN_delete_gc_root (void *root)
```

Deletes the gc root.

37.18 CHICKEN_gc_root_ref

```
[C macro] C_word CHICKEN_gc_root_ref (void *root)
```

Returns the value stored in the gc root.

37.19 CHICKEN_gc_root_set

```
[C macro] void CHICKEN_gc_root_set (void *root, C_word value)
```

Sets the content of the GC root to a new value.

Sometimes it is handy to access global variables from C code:

37.20 CHICKEN_global_lookup

```
[C function] void* CHICKEN_global_lookup (char *name)
```

Returns a GC root that holds the global variable with the name `name`. If no such variable exists, `NULL` is returned.

37.21 CHICKEN_global_ref

```
[C function] C_word CHICKEN_global_ref (void *global)
```

Returns the value of the global variable referenced by the GC root `global`.

37.22 CHICKEN_global_set

`[C function] void CHICKEN_global_set (void *global, C_word value)`

Sets the value of the global variable referenced by the GC root `global` to `value`.

Previous: Foreign type specifiers

Next: Callbacks

38 Callbacks

To enable an external C function to call back to Scheme, the form `foreign-safe-lambda` (or `foreign-safe-lambda*`) has to be used. This generates special code to save and restore important state information during execution of C code. There are two ways of calling Scheme procedures from C: the first is to invoke the runtime function `C_callback` with the closure to be called and the number of arguments. The second is to define an externally visible wrapper function around a Scheme procedure with the `define-external` form.

Note: the names of all functions, variables and macros exported by the CHICKEN runtime system start with `C_`. It is advisable to use a different naming scheme for your own code to avoid name clashes. Callbacks (defined by `define-external`) do not capture the lexical environment.

Non-local exits leaving the scope of the invocation of a callback from Scheme into C will not remove the C call-frame from the stack (and will result in a memory leak).

38.1 define-external

```
[syntax] (define-external [QUALIFIERS] (NAME (ARGUMENTTYPE1 VARIABLE1) ...) RET
[syntax] (define-external NAME TYPE [INIT])
```

The first form defines an externally callable Scheme procedure. `NAME` should be a symbol, which, when converted to a string, represents a legal C identifier. `ARGUMENTTYPE1 ...` and `RETURNTYPE` are foreign type specifiers for the argument variables `VAR1 ...` and the result, respectively. `QUALIFIERS` is an optional qualifier for the foreign procedure definition, like `__stdcall`.

```
(define-external (foo (c-string x)) int (string-length x))
```

The second form of `define-external` can be used to define variables that are accessible from foreign code. It declares a global variable named by the symbol `NAME` that has the type `TYPE`. `INIT` can be an arbitrary expression that is used to initialize the variable. `NAME` is accessible from Scheme just like any other foreign variable defined by `define-foreign-variable`.

```
(define-external foo int 42)
((foreign-lambda* int ()
  "C_return(foo);"))           ==> 42
```

Note: don't be tempted to assign strings or bytevectors to external variables. Garbage collection moves those objects around, so it is very bad idea to assign pointers to heap-data. If you have to do so, then copy the data object into statically allocated memory (for example by using `object-evict`).

Results of type `scheme-object` returned by `define-external` are always allocated in the secondary heap, that is, not in the stack.

38.2 C_callback

```
[C function] C_word C_callback (C_word closure, int argc)
```

This function can be used to invoke the Scheme procedure `closure`. `argc` should contain the number of arguments that are passed to the procedure on the temporary stack. Values are put onto the temporary stack with the `C_save` macro.

38.3 C_callback_adjust_stack_limits

```
[C function] void C_callback_adjust_stack_limits (C_word *ptr)
```

The runtime-system uses the stack as a special allocation area and internally holds pointers to estimated limits to distinguish between Scheme data objects inside the stack from objects outside of it. If you invoke callbacks at wildly differing stack-levels, these limits may shift from invocation to invocation. Callbacks defined with `define-external` will perform appropriate adjustments automatically, but if you invoke `C_callback` manually, you should perform a `C_callback_adjust_stack_limits` to make sure the internal limits are set properly. `ptr` should point to some data object on the stack. The call will make sure the limits are adjusted so that the value pointed to by `ptr` is located in the stack.

Previous: Embedding

Next: Locations

39 Locations

It is also possible to define variables containing unboxed C data, so called *locations*. It should be noted that locations may only contain simple data, that is: everything that fits into a machine word, and double-precision floating point values.

39.1 define-location

```
[syntax] (define-location NAME TYPE [INIT])
```

Identical to `(define-external NAME TYPE [INIT])`, but the variable is not accessible from outside of the current compilation unit (it is declared `static`).

39.2 let-location

```
[syntax] (let-location ((NAME TYPE [INIT]) ...) BODY ...)
```

Defines a lexically bound location.

39.3 location

```
[syntax] (location NAME)
[syntax] (location X)
```

This form returns a pointer object that contains the address of the variable `NAME`. If the argument to `location` is not a location defined by `define-location`, `define-external` or `let-location`, then

```
(location X)
```

is essentially equivalent to

```
(make-locative X)
```

(See the manual chapter or `locatives` for more information about locatives.

Note that `(location X)` may be abbreviated as `#$X`.

```
(define-external foo int)
((foreign-lambda* void (((pointer int) ip)) "*ip = 123;")
  (location foo))
foo
```

This facility is especially useful in situations, where a C function returns more than one result value:

```
#>
#include <math.h>
<#
```

```
(define modf
  (foreign-lambda double "modf" double (pointer double)) )

(let-location ([i double])
  (let ([f (modf 1.99 (location i))])
    (print "i=" i ", f=" f) ) )
```

`location` returns a value of type `c-pointer`, when given the name of a callback-procedure defined with `define-external`.

Previous: Callbacks

Next: Other support procedures

39.4 Other support procedures

39.5 argc+argv

`[procedure] (argc+argv)`

Returns two values: an integer and a foreign-pointer object representing the `argc` and `argv` arguments passed to the current process.

Previous: Locations

Next: C interface

40 C interface

The following functions and macros are available for C code that invokes Scheme or foreign procedures that are called by Scheme:

40.1 C_save

```
[C macro] void C_save (C_word x) :
```

Saves the Scheme data object `x` on the temporary stack.

40.2 C_restore

```
[C macro] void C_restore
```

Pops and returns the topmost value from the temporary stack.

40.3 C_fix

```
[C macro] C_word C_fix (int integer)
```

40.4 C_make_character

```
[C macro] C_word C_make_character (int char_code)
```

40.5 C_SCHEME_END_OF_LIST

```
[C macro] C_SCHEME_END_OF_LIST
```

40.6 C_word C_SCHEME_END_OF_FILE

```
[C macro] C_SCHEME_END_OF_FILE
```

40.7 C_word C_SCHEME_FALSE

```
[C macro] C_SCHEME_FALSE
```

40.8 C_word C_SCHEME_TRUE

```
[C macro] C_SCHEME_TRUE
```

These macros return immediate Scheme data objects.

40.9 C_string

```
[C function] C_word C_string (C_word **ptr, int length, char *string)
```

40.10 C_string2

```
[C function] C_word C_string2 (C_word **ptr, char *zero_terminated_string)
```

40.11 C_intern2

```
[C function] C_word C_intern2 (C_word **ptr, char *zero_terminated_string)
```

40.12 C_intern3

```
[C function] C_word C_intern3 (C_word **ptr, char *zero_terminated_string, C_wo
```

40.13 C_pair

```
[C function] C_word C_pair (C_word **ptr, C_word car, C_word cdr)
```

40.14 C_flonum

```
[C function] C_word C_flonum (C_word **ptr, double number)
```

40.15 C_int_to_num

```
[C function] C_word C_int_to_num (C_word **ptr, int integer)
```

40.16 C_mpointer

```
[C function] C_word C_mpointer (C_word **ptr, void *pointer)
```

40.17 C_vector

`[C function] C_word C_vector (C_word **ptr, int length, ...)`

40.18 C_list

`[C function] C_word C_list (C_word **ptr, int length, ...)`

These functions allocate memory from `ptr` and initialize a fresh data object. The new data object is returned. `ptr` should be the **address** of an allocation pointer created with `C_alloc`.

40.19 C_alloc

`[C macro] C_word* C_alloc (int words)`

Allocates memory from the C stack (`C_alloc`) and returns a pointer to it. `words` should be the number of words needed for all data objects that are to be created in this function. Note that stack-allocated data objects have to be passed to Scheme callback functions, or they will not be seen by the garbage collector. This is really only usable for callback procedure invocations, make sure not to use it in normal code, because the allocated memory will be re-used after the foreign procedure returns. When invoking Scheme callback procedures a minor garbage collection is performed, so data allocated with `C_alloc` will already have moved to a safe place.

Note that `C_alloc` is really just a wrapper around `alloca`, and can also be simulated by declaring a stack-allocated array of `C_word`s:

40.20 C_SIZEOF_LIST

`[C macro] int C_SIZEOF_LIST (int length)`

40.21 C_SIZEOF_STRING

`[C macro] int C_SIZEOF_STRING (int length)`

40.22 C_SIZEOF_VECTOR

`[C macro] int C_SIZEOF_VECTOR (int length)`

40.23 C_SIZEOF_INTERNED_SYMBOL

```
[C macro] int C_SIZEOF_INTERNED_SYMBOL (int length)
```

40.24 C_SIZEOF_PAIR

```
[C macro] int C_SIZEOF_PAIR
```

40.25 C_SIZEOF_FLONUM

```
[C macro] int C_SIZEOF_FLONUM
```

40.26 C_SIZEOF_POINTER

```
[C macro] int C_SIZEOF_POINTER
```

40.27 C_SIZEOF_LOCATIVE

```
[C macro] int C_SIZEOF_LOCATIVE
```

40.28 C_SIZEOF_TAGGED_POINTER

```
[C macro] int C_SIZEOF_TAGGED_POINTER
```

These are macros that return the size in words needed for a data object of a given type.

40.29 C_character_code

```
[C macro] int C_character_code (C_word character)
```

40.30 C_unfix

```
[C macro] int C_unfix (C_word fixnum)
```

40.31 C_flonum_magnitude

```
[C macro] double C_flonum_magnitude (C_word flonum)
```

40.32 C_c_string

`[C function] char* C_c_string (C_word string)`

40.33 C_num_to_int

`[C function] int C_num_to_int (C_word fixnum_or_flonum)`

40.34 C_pointer_address

`[C function] void* C_pointer_address (C_word pointer)`

These macros and functions can be used to convert Scheme data objects back to C data. Note that `C_c_string()` returns a pointer to the character buffer of the actual Scheme object and is not zero-terminated.

40.35 C_header_size

`[C macro] int C_header_size (C_word x)`

40.36 C_header_bits

`[C macro] int C_header_bits (C_word x)`

Return the number of elements and the type-bits of the non-immediate Scheme data object `x`.

40.37 C_block_item

`[C macro] C_word C_block_item (C_word x, int index)`

This macro can be used to access slots of the non-immediate Scheme data object `x`. `index` specifies the index of the slot to be fetched, starting at 0. Pairs have 2 slots, one for the **car** and one for the **cdr**. Vectors have one slot for each element.

40.38 C_u_i_car

`[C macro] C_word C_u_i_car (C_word x)`

40.39 C_u_i_car

`[C macro] C_word C_u_i_car (C_word x)`

Aliases for `C_block_item(x, 0)` and `C_block_item(x, 1)`, respectively.

40.40 C_data_pointer

`[C macro] void* C_data_pointer (C_word x)`

Returns a pointer to the data-section of a non-immediate Scheme object.

40.41 C_make_header

`[C macro] C_word C_make_header (C_word bits, C_word size)`

A macro to build a Scheme object header from its bits and size parts.

40.42 C_mutate

`[C function] C_word C_mutate (C_word *slot, C_word val)`

Assign the Scheme value `val` to the location specified by `slot`. If the value points to data inside the nursery (the first heap-generation), then the garbage collector will remember to handle the data appropriately. Assigning nursery-pointers directly will otherwise result in lost data. Note that no copying takes place at the moment when `C_mutate` is called, but later - at the next (minor) garbage collection.

40.43 C_symbol_value

`[C macro] C_word C_symbol_value (C_word symbol)`

Returns the global value of the variable with the name `symbol`. If the variable is unbound `C_SCHEME_UNBOUND` is returned. You can set a variable's value with `C_mutate(&C_symbol_value(SYMBOL), VALUE)`.

40.44 C_gc_protect

`[C function] void C_gc_protect (C_word *ptrs[], int n)`

Registers `n` variables at address `ptrs` to be garbage collection roots. The locations should not contain pointers to data allocated in the nursery, only immediate values or pointers to heap-data are valid. Any assignment of potential nursery data into a root-array should be done via `C_mutate()`. The variables have to be initialized to sensible values before the next garbage collection starts (when in doubt, set all locations in `ptrs` to `C_SCHEME_UNDEFINED`) `C_gc_protect` may not called before the runtime system has been initialized (either by `CHICKEN_initialize`, `CHICKEN_run` or `CHICKEN_invoke`.

For a slightly simpler interface to creating and using GC roots see `CHICKEN_new_gc_root`.

40.45 C_gc_unprotect

```
[C function] void C_gc_unprotect (int n)
```

Removes the last `n` registered variables from the set of root variables.

40.46 C_post_gc_hook

```
[C Variable] void (*C_post_gc_hook)(int mode)
```

If not `NULL`, the function pointed to by this variable will be called after each garbage collection with a flag indicating what kind of collection was performed (either `0` for a minor collection or `1` for a major collection). Minor collections happen very frequently, so the hook function should not consume too much time. The hook function may not invoke Scheme callbacks.

An example:

```
% cat foo.scm
#>
extern int callout(int, int, int);
<#

(define callout (foreign-safe-lambda int "callout" int int int))

(define-external (callin (scheme-object xyz)) int
  (print "This is 'callin': " xyz)
  123)

(print (callout 1 2 3))

% cat bar.c
#include <stdio.h>
#include "chicken.h"

extern int callout(int, int, int);
extern int callin(C_word x);

int callout(int x, int y, int z)
{
  C_word *ptr = C_alloc(C_SIZEOF_LIST(3));
  C_word lst;

  printf("This is 'callout': %d, %d, %d\n", x, y, z);
  lst = C_list(&ptr, 3, C_fix(x), C_fix(y), C_fix(z));
  return callin(lst);  /* Note: `callin' will have GC'd the data in `ptr' */
}
```

```
% csc foo.scm bar.c -o foo
% foo
This is 'callout': 1, 2, 3
This is 'callin': (1 2 3)
123
```

Notes:

Scheme procedures can call C functions, and C functions can call Scheme procedures, but for every pending C stack frame, the available size of the first heap generation (the *nursery*) will be decreased, because the C stack is identical to the nursery. On systems with a small nursery this might result in thrashing, since the C code between the invocation of C from Scheme and the actual calling back to Scheme might build up several stack-frames or allocates large amounts of stack data. To prevent this it is advisable to increase the default nursery size, either when compiling the file (using the `-nursery` option) or when running the executable (using the `-:s` runtime option).

Calls to Scheme/C may be nested arbitrarily, and Scheme continuations can be invoked as usual, but keep in mind that C stack frames will not be recovered, when a Scheme procedure call from C does not return normally.

When multiple threads are running concurrently, and control switches from one thread to another, then the continuation of the current thread is captured and saved. Any pending C stack frame still active from a callback will remain on the stack until the threads is re-activated again. This means that in a multithreading situation, when C callbacks are involved, the available nursery space can be smaller than expected. So doing many nested SchemeCScheme calls can reduce the available memory up to the point of thrashing. It is advisable to have only a single thread with pending C stack-frames at any given time.

Pointers to Scheme data objects should not be stored in local or global variables while calling back to Scheme. Any Scheme object not passed back to Scheme will be reclaimed or moved by the garbage collector.

Calls from C to Scheme are never tail-recursive.

Continuations captured via `call-with-current-continuation` and passed to C code can be invoked like any other Scheme procedure.

Previous: Other support procedures

Next: chicken-setup

41 chicken-setup

41.1 Extension libraries

Extension libraries (*eggs*) are extensions to the core functionality provided by the basic CHICKEN system, to be built and installed separately. The mechanism for loading compiled extensions is based on dynamically loadable code and as such is only available on systems on which loading compiled code at runtime is supported. Currently these are most UNIX-compatible platforms that provide the `libdl` functionality like Linux, Solaris, BSD, Mac OS X and Windows using Cygwin.

Note: Extension may also be normal applications or shell scripts, but are usually libraries.

`chicken-setup` will download the source code for extension automatically from the canonical server at http://www.call-with-current-continuation.org/eggs if the requested egg does not exist in the current directory. Various command-line options exist for customizing the process and/or retrieving the egg from other locations or in other formats.

41.2 Installing extensions

To install an extension library, run the `chicken-setup` program with the extension name as argument. The extension archive is downloaded, its contents extracted and the contained *setup* script is executed. This setup script is a normal Scheme source file, which will be interpreted by `chicken-setup`. The complete language supported by `csi` is available, and the library units `srfi-1 regex utils posix tcp` are loaded. Additional libraries can be loaded at run-time.

The setup script should perform all necessary steps to build the new library (or application). After a successful build, the extension can be installed by invoking one of the procedures `install-extension`, `install-program` or `install-script`. These procedures will copy a number of given files into the extension repository or in the path where the CHICKEN executables are located (in the case of executable programs or scripts). Additionally the list of installed files, and user-defined metadata is stored in the repository.

If no extension name is given on the command-line, and if none of the options `-list`, `-version`, `-repository` (without argument), `-program-path` (without argument), `-fetch`, `-fetch-tree` or `-docindex` is given, then all `.setup` scripts in the current directory are processed.

41.3 Creating extensions

Extensions can be created by creating an (optionally gzipped) `tar` archive named `EXTENSION.egg` containing all needed files plus a `.setup` script in the root directory. After `chicken-setup` has extracted the files, the setup script will be invoked. There are no additional constraints on the structure of the archive, but the setup script has to be in the root path of the archive.

41.4 Procedures and macros available in setup scripts

41.4.1 install-extension

```
(install-extension ID FILELIST [INFOLIST])
```

Installs the extension library with the name `ID`. All files given in the list of strings `FILELIST` will be copied to the extension repository. It should be noted here that the extension id has to be identical to the name of the file implementing the extension. The extension may load or include other files, or may load other extensions at runtime specified by the `require-at-runtime` property.

`FILELIST` may be a filename, a list of filenames, or a list of pairs of the form `(SOURCE DEST)` (if you want to copy into a particular sub-directory - the destination directory will be created as needed). If `DEST` is a relative pathname, < it will be copied into the extension repository.

The optional argument `INFOLIST` should be an association list that maps symbols to values, this list will be stored as `ID.setup` at the same location as the extension code. Currently the following properties are used:

41.4.1.1 syntax

```
[extension property] (syntax)
```

Marks the extension as syntax-only. No code is compiled, the extension is intended as a file containing macros to be loaded at compile/macro-expansion time.

41.4.1.2 require-at-runtime

```
[extension property] (require-at-runtime ID ...)
```

Specifies extensions that should be loaded (via `require`) at runtime. This is mostly useful for syntax extensions that need additional support code at runtime.

41.4.1.3 version

```
[extension property] (version STRING)
```

Specifies version string.

41.4.1.4 documentation

```
[extension property] (documentation FILENAME)
```

The filename of a HTML document containing extension-specific documentation. This file should be given in the file-list passed to `install-extension` and a link to it will be automatically included in the index page (accessible via `chicken-setup -docindex`).

41.4.1.5 examples

`[extension property] (examples FILENAME ...)`

Copies the given files into the examples directory, which is usually `$prefix/share/chicken/examples` (equivalent to `$CHICKEN_HOME/examples` or `(make-pathname (chicken-home) "examples")`).

Note that the files listed in this property should not be listed in the normal list of files to install passed to `install-extension`. This is the only exception - other files that are installed in the repository must be given in the file list.

41.4.1.6 exports

`[extension property] (exports EXPORT ...)`

Add export-information to the generated extension-information. `EXPORT` may be a symbol naming an exported toplevel variable or a string designating a file with exported variables, as generated by the `-emit-exports` option or the `emit-exports` declaration specifier.

41.4.1.7 static

`[extension property] (static STRING)`

If the extension also provides a static library, then STRING should contain the name of that library. Used by `csc` when compiling with the `-static-extensions` option.

41.4.1.8 static-options

`[extension property] (static-options STRING)`

Additional options that should be passed to the linker when linking with the static version of an extension (see `static` above). Used by `csc` when compiling with the `-static-extensions` option.

All other properties are currently ignored. The `FILELIST` argument may also be a single string.

41.4.2 install-program

`[procedure] (install-program ID FILELIST [INFOLIST])`

Similar to `install-extension`, but installs an executable program in the executable path (usually `/usr/local/bin`).

41.4.3 install-script

`[procedure] (install-script ID FILELIST [INFOLIST])`

Similar to `install-program`, but additionally changes the file permissions of all files in `FILELIST` to executable (for installing shell-scripts).

41.4.4 run

`[syntax] (run FORM ...)`

Runs the shell command `FORM`, which is wrapped in an implicit `quasiquote`. `(run (csc ...))` is treated specially and passes `-v` (if `-verbose` has been given to `chicken-setup`) and `-feature compiling-extension` options to the compiler.

41.4.5 compile

`[syntax] (compile FORM ...)`

Equivalent to `(run (csc FORM ...))`.

41.4.6 make

`[syntax] (make ((TARGET (DEPENDENT ...) COMMAND ...) ...) ARGUMENTS)`

A *make* macro that executes the expressions `COMMAND ...`, when any of the dependents `DEPENDENT ...` have changed, to build `TARGET`. This is the same as the `make` extension, which is available separately. For more information, see make.

41.4.7 patch

`[procedure] (patch WHICH REGEX SUBST)`

Replaces all occurrences of the regular expression `REGEX` with the string `SUBST`, in the file given in `WHICH`. If `WHICH` is a string, the file will be patched and overwritten. If `WHICH` is a list of the form `OLD NEW`, then a different file named `NEW` will be generated.

41.4.8 copy-file

`[procedure] (copy-file FROM TO)`

Copies the file or directory (recursively) given in the string `FROM` to the destination file or directory `TO`.

41.4.9 move-file

```
[procedure] (move-file FROM TO)
```

Moves the file or directory (recursively) given in the string `FROM` to the destination file or directory `TO`.

41.4.10 remove-file*

```
[procedure] (remove-file* PATH)
```

Removes the file or directory given in the string `PATH`.

41.4.11 find-library

```
[procedure] (find-library NAME PROC)
```

Returns `#t` if the library named `libNAME.[a|so]` (unix) or `NAME.lib` (windows) could be found by compiling and linking a test program. `PROC` should be the name of a C function that must be provided by the library. If no such library was found or the function could not be resolved, `#f` is returned.

41.4.12 find-header

```
[procedure] (find-header NAME)
```

Returns `#t` if a C include-file with the given name is available, or `#f` otherwise.

41.4.13 try-compile

```
[procedure] (try-compile CODE #!key cc cflags ldflags compile-only c++)
```

Returns `#t` if the C code in `CODE` compiles and links successfully, or `#f` otherwise. The keyword parameters `cc` (compiler name, defaults to the C compiler used to build this system), `cflags` and `ldflags` accept additional compilation and linking options. If `compile-only` is true, then no linking step takes place. If the keyword argument `c++` is given and true, then the code will be compiled in C++ mode.

41.4.14 create-directory

```
[procedure] (create-directory PATH)
```

Creates the directory given in the string `PATH`, with all parent directories as needed.

41.4.15 installation-prefix

```
[parameter] installation-prefix
```

Holds the prefix under which CHICKEN executables and libraries have been installed (either the value of the environment variable `CHICKEN_PREFIX` or whatever prefix was specified at the time the system was built.

41.4.16 program-path

```
[parameter] (program-path [PATH])
```

Holds the path where executables are installed and defaults to either `$CHICKEN_PREFIX/bin`, if the environment variable `CHICKEN_PREFIX` is set, `$CHICKEN_HOME` or the path where the CHICKEN binaries (`chicken`, `csi`, etc.) are installed.

41.4.17 setup-root-directory

```
[parameter] (setup-root-directory [PATH])
```

Contains the path of the directory where `chicken-setup` was invoked.

41.4.18 setup-build-directory

```
[parameter] (setup-build-directory [PATH])
```

Contains the path of the directory where the extension is built. This is not necessarily identical to `setup-root-directory`.

41.4.19 setup-verbose-flag

```
[parameter] (setup-verbose-flag [BOOL])
```

Reflects the setting of the `-verbose` option, i.e. is `#t`, if `-verbose` was given.

41.4.20 setup-install-flag

```
[parameter] (setup-install-flag [BOOL])
```

Reflects the setting of the `--no-install` option, i.e. is `#f`, if `-no-install` was given.

41.4.21 required-chicken-version

```
[procedure] (required-chicken-version VERSION)
```

Signals an error if the version of CHICKEN that this script runs under is lexicographically less than `VERSION` (the argument will be converted to a string, first).

41.4.22 required-extension-version

```
[procedure] (required-extension-version EXTENSION1 VERSION1 ...)
```

Checks whether the extensions `EXTENSION1 ...` are installed and at least of version `VERSION1 ...`. The test is made by lexicographically comparing the string-representations of the given version with the version of the installed extension. If one of the listed extensions is not installed, has no associated version information or is of a version older than the one specified.

41.4.23 cross-chicken

```
[procedure] (cross-chicken)
```

Retrusn `#t` if this system is configured for cross-compilation or `#f` otherwise.

41.5 Examples for extensions

The simplest case is a single file that does not export any syntax. For example

```
;;;; hello.scm

(define (hello name)
  (print "Hello, " name " !") )
```

We need a `.setup` script to build and install our nifty extension:

```
;;;; hello.setup

;; compile the code into a dynamically loadable shared object
;; (will generate hello.so or hello.dll on Windows)
(compile -s hello.scm)

;; Install as extension library
(install-extension 'hello "hello.so")
```

(Note: `install-extension` will recognized the `.so` extension and automatically do the right thing on Windows)

After entering

```
$ chicken-setup hello
```

at the shell prompt (and in the same directory where the two files exist), the file `hello.scm` will be compiled into a dynamically loadable library. If the compilation succeeds, `hello.so` will be stored in the repository, together with a file named `hello.setup-info` containing an a-list with metadata. If no extension name is given to `chicken-setup`, it will simply execute the first file with the `.setup` extension it can find.

Use it like any other CHICKEN extension:

```
$ csi -q
#;1> (require-extension hello)
; loading /usr/local/lib/chicken/1/hello.so ...
#;2> (hello "me")
Hello, me!
#;3>
```

Here we create a simple application:

```
;;;; hello2.scm

(print "Hello, ")
(for-each (lambda (x) (printf "~A " x)) (command-line-arguments))
(print "!")
```

We also need a setup script:

```
;;;; hello2.setup

(compile hello2.scm)  ; compile `hello2'
(install-program 'hello2 "hello2") ; name of the extension and files to be inst
```

To use it, just run `chicken-setup` in the same directory:

```
$ chicken-setup
```

(Here we omit the extension name)

Now the program `hello2` will be installed in the same location as the other CHICKEN tools (like `chicken`, `csi`, etc.), which will normally be `/usr/local/bin`. Note that you need write-permissions for those locations and may have to run `chicken-setup` with administrative rights.

Uninstallation is just as easy:

```
$ chicken-setup -uninstall hello2
```

`chicken-setup` provides a `make` macro, so build operations can be of arbitrary complexity. When running `chicken-setup` with an argument `NAME`, for which no associated file `NAME.setup`, `NAME.egg` or `NAME.scm` exists will ask you to download the extension via HTTP from the default URL http://www.call-with-current-continuation.org/eggs. You can use the `-host` option to specify an alternative source location. Extensions that are required to compile and/or use the requested extension are downloaded and installed automatically.

If the given extension name contains a path prefix and the `-host` option is given, then `chicken-setup` can also download and install eggs from an arbitrary HTTP server. Alternatively you can pass a full URL (including the `http://` prefix. Note that no dependency checks are done when downloading eggs directly with the URL syntax.

Finally a somewhat more complex example: We want to package a syntax extension with additional support code that is to be loaded at run-time of any Scheme code that uses that extension. We create a *glass* lambda, a procedure with free variables that can be manipulated from outside:

```
;;;; glass.scm

(define-macro (glass-lambda llist vars . body)
  ;; Low-level macros are fun!
  (let ([lvar (gensym)]
        [svar (gensym)]
        [x (gensym)]
        [y (gensym)]
        [yn (gensym)] )
    `(let ,(map (lambda (v) (list v #f)) vars)
       (define (,svar ,x . ,y)
  let* ([(yn (pair? ,y)]
     and ,yn (car[,yy())] )
    case ,x(
             ,@(map lambda (v)
              if ,yn  `([,v] (
  set! ,v ,y)                    (
                                 ,v) ) )
                    vars)
      else (error "variable not found" ,x)) ) ) )
       (define ,lvar (lambda ,llist ,@body))
       (extend-procedure ,lvar ,svar) ) ) )
```

Here some support code that needs to be loaded at runtime:

```
;;;; glass-support.scm

(require-extension lolevel)

(define glass-lambda-accessor procedure-data)
(define (glass-lambda-ref gl v) ((procedure-data gl) v))
(define (glass-lambda-set! gl v x) ((procedure-data gl) v x))
```

The setup script looks like this:

```
(compile -s glass-support.scm)

(install-extension
  'glass
  '("glass.scm" "glass-support.so")
  '((syntax) (require-at-runtime glass-support)) )
```

The invocation of `install-extension` provides the files that are to be copied into the extension repository, and a metadata list that specifies that the extension `glass` is a syntax extension and that, if it is declared to be used by other code (either with the `require-extension` or `require-for-syntax` form), then client code should perform an implicit `(require 'glass-support)` at startup.

This can be conveniently packaged as an *egg*:

```
$ tar cfz glass.egg glass.setup glass.scm glass-support.scm
```

And now we use it:

```
$ chicken-setup glass
$ csi -quiet
#;1> (require-extension glass)
; loading /usr/local/lib/chicken/1/glass.scm ...
; loading /usr/local/lib/chicken/1/glass-support.so ...
#;2> (define foo (glass-lambda (x) (y) (+ x y)))
#;3> (glass-lambda-set! foo 'y 99)
#;4> (foo 33)
132
```

41.6 chicken-setup reference

Available options:

`-h -help`
: Show usage information and exit.

`-V -version`
: Display version and exit.

`-R -repository [PATHNAME]`
: When used without an argument, the path of the extension repository is displayed on standard output. When given an argument, the repository pathname (and the `repository-path` parameter) will be set to `PATHNAME` for all subsequent operations. The default repository path is the installation library directory (usually `/usr/local/lib/chicken`), or (if set) the directory given in the environment variable `CHICKEN_REPOSITORY`. `PATHNAME` should be an absolute pathname.

`-P -program-path [PATHNAME]`
: When used without an argument, the path for executables is displayed on standard output. When given an argument, the program path for installing executables and scripts will be set to `PATHNAME` for all subsequent operations. `PATHNAME` should be an absolute pathname.

`-h -host HOSTNAME[:PORT]`
: Specifies alternative host for downloading extensions, optionally with a TCP port number (which defaults to 80).

`-u -uninstall EXTENSION`
: Removes all files that were installed for `EXTENSION` from the file-system, together with any metadata that has been stored.

`-l -list [NAME ...]`
: List all installed extensions or show extension information.

`-r -run FILENAME`
: Load and execute given file.

`-s -script FILENAME`
: Executes the given Scheme source file with all remaining arguments and exit. The *she-bang* shell script header is recognized, so you can write Scheme scripts that use `chicken-setup` just as with `csi`.

`-e -eval EXPRESSION`
: Evaluates the given expression(s).

`-v -verbose`
: Display additional debug information.

`-k -keep`
: Keep temporary files and directories.

`-c -csc-option OPTION`
: Passes `OPTION` as an extra argument to invocations of the compiler-driver (`csc`). This works only if `csc` is invoked as `(run (csc ...))`.

`-d -dont-ask`
: Do not ask the user before trying to download required extensions.

`-n -no-install`
: Do not install generated binaries and/or support files. Any invocations of `install-program`, `install-extension` or `install-script` will be be no-ops.

`-i -docindex`
: Displays the path to the index-page of any installed extension-documentation. If the index page does not exist, it is created.

`-C -check`
: Downloads the repository-index and lists locally installed extensions for which a newer release is available for download.

`-t -test EXTENSION ...`
: return success if all given extensions are installed

`-ls EXTENSION`
: List installed files for extension

`-fetch-tree`
: Download and print the repository catalog

`-tree FILENAME`
: Download and show the repository catalog

`-svn URL`
: Fetch extension from Subversion repository

`-revision REV`
: Specifies SVN revision to check out

`-local PATHNAME`
: Fetch extension from local file

`-destdir PATHNAME`
: Specify alternative installation prefix (for packaging)

`--`
: Ignore all following arguments.

Note that the options are processed exactly in the order in which they appear in the command-line.

41.7 Windows notes

`chicken-setup` works on Windows, when compiled with Visual C++, but depends on the `tar` and `gunzip` tools to extract the contents of an egg. The best way is to download an egg either manually (or with `chicken-setup -fetch`) and extract its contents with a separate program (like `winzip`). the `CHICKEN_REPOSITORY` environment variable has to be set (in addition to `CHICKEN_HOME`) to a directory where your compiled extensions should be located.

The `.setup` scripts will not always work under Windows, and the extensions may require libraries that are not provided for Windows or work differently. Under these circumstances it is recommended to perform the required steps to build an extension manually.

The required UNIX tools are also available as Windows binaries. Google or ask on the CHICKEN mailing list if you need help locating them.

41.8 Security

When extensions are downloaded and installed one is executing code from potentially compromised systems. This applies also when `chicken-setup` executes system tests for required extensions. As the code has been retrieved over the network effectively untrusted code is going to be evaluated. When `chicken-setup` is run as *root* the whole system is at the mercy of the build instructions (note that this is also the case every time you install software via `sudo make install`, so this is not specific to the CHICKEN extension mechanism).

Security-conscious users should never run `chicken-setup` as root. A simple remedy is to set the environment variable `CHICKEN_REPOSITORY`, which will transparently place the repository at an arbitrary user-selected location. Alternatively obtain write/execute access to the default location of the repository (usually `/usr/local/lib/chicken`) to avoid running as root.

41.9 Other modes if installation

It is possible to install extensions directly from a Subversion repository or from a local checkout by using the `-svn` or `-local` options. By using either the `svn` client program (which must be installed) or file-system operations all necessary files will be copied into the current directory (creating a subdirectory named `EXTENSIONNAME.egg-dir`), built and subsequently installed.

Dependency information which is necessary to ensure required extensions are also installed is downloaded automatically. If you have no internet connection or don't want to connect, you can als use a local file containing the necessary dependency information. The `-fetch-tree` option retrieves the canonical *repository file* at http://www.call-with-current-continuation.org/eggs/repository, writing it to stdout. Redirecting this output into a file and passing the file via the `-tree` option to `chicken-setup` allows you now to use the local repository file:

Retrieve complete extension repository (big):

```
% cd /opt
% svn co https://galinha.ucpel.tche.br/svn/chicken-eggs
```

Get your own copy of the repository file:

```
% chicken-setup -fetch-tree >~/my-repository-file
```

Now you can install eggs from your local checkout, with full dependency tracking and without being connected to the internet:

```
% cd ~/tmp
% chicken-setup -local /opt/eggs/chicken-eggs -tree ~/my-repository-file opengl
```

41.10 Linking extensions statically

The compiler and chicken-setup support statically linked eggs. The general approach is to generate an object file or static library (in addition to the usual shared library) in your `.setup` script and install it along with the dynamically loadable extension. The setup properties `static` should contain the name of the object file (or static library) to be linked, when `csc` gets passed the `-static-extensions` option:

```
(compile -s -O2 -d1 my-ext.scm)   ; dynamically loadable "normal" version
```

```
(compile -c -O2 -d1 my-ext -unit my-ext)  ; statically linkable version
(install-extension
  'my-ext
  '("my-ext.so" "my-ext.o")
  '((static "my-ext.o")) )
```

Note the use of the `-unit` option in the second compilation step: static linking must use static library units. `chicken-setup` will perform platform-dependent file-extension translation for the file list, but does currently not do that for the `static` extension property.

To actually link with the static version of `my-ext`, do:

```
% csc -static-extensions my-program.scm -uses my-ext
```

The compiler will try to do the right thing, but can not handle all extensions, since the ability to statically link eggs is relatively new. Eggs that support static linking are designated as being able to do so. If you require a statically linkable version of an egg that has not been converted yet, contact the extension author or the CHICKEN mailing list.

Previous: Interface to external functions and variables

Next: Data representation

42 Data Representation

Note: In all cases below, bits are numbered starting at 1 and beginning with the lowest-order bit.

There exist two different kinds of data objects in the CHICKEN system: immediate and non-immediate objects. Immediate objects are represented by a single machine word, which is usually of 32 bits length, or 64 bits on 64-bit architectures. The immediate objects come in four different flavors:

fixnums, that is, small exact integers, where bit 1 is set to 1. This gives fixnums a range of 31 bits for the actual numeric value (63 bits on 64-bit architectures).

characters, where bits 1-4 are equal to `C_CHARACTER_BITS`. The Unicode code point of the character is encoded in bits 9 to 32.

booleans, where bits 1-4 are equal to `C_BOOLEAN_BITS`. Bit 5 is one for #t and zero for #f.

other values: the empty list, the value of unbound identifiers, the undefined value (void), and end-of-file. Bits 1-4 are equal to `C_SPECIAL_BITS`; bits 5 to 8 contain an identifying number for this type of object. The following constants are defined: `C_SCHEME_END_OF_LIST` `C_SCHEME_UNDEFINED` `C_SCHEME_UNBOUND` `C_SCHEME_END_OF_FILE`

Non-immediate objects are blocks of data represented by a pointer into the heap. The first word of the data block contains a header, which gives information about the type of the object. The header has the size of a machine word, usually 32 bits (64 bits on 64 bit architectures).

bits 1 to 24 contain the length of the data object, which is either the number of bytes in a string (or byte-vector) or the the number of elements for a vector or for a structure type.

bits 25 to 28 contain the type code of the object.

bits 29 to 32 contain miscellaneous flags used for garbage collection or internal data type dispatching. These flags are:

C_GC_FORWARDING_BIT
: Flag used for forwarding garbage collected object pointers.

C_BYTEBLOCK_BIT
: Flag that specifies whether this data object contains raw bytes (a string or byte-vector) or pointers to other data objects.

C_SPECIALBLOCK_BIT
: Flag that specifies whether this object contains a *special* non-object pointer value in its first slot. An example for this kind of objects are closures, which are a vector-type object with the code-pointer as the first item.

C_8ALIGN_BIT
: Flag that specifies whether the data area of this block should be aligned on an 8-byte boundary (floating-point numbers, for example).

The actual data follows immediately after the header. Note that block-addresses are always aligned to the native machine-word boundary. Scheme data objects map to blocks in the following manner:

pairs: vector-like object (type bits `C_PAIR_TYPE`), where the car and the cdr are contained in the first and second slots, respectively.

vectors: vector object (type bits `C_VECTOR_TYPE`).

strings: byte-vector object (type bits `C_STRING_TYPE`).

procedures: special vector object (type bits `C_CLOSURE_TYPE`). The first slot contains a pointer to a compiled C function. Any extra slots contain the free variables (since a flat closure representation is used).

flonum: a byte-vector object (type bits `C_FLONUM_BITS`). Slots one and two (or a single slot on 64 bit architectures) contain a 64-bit floating-point number, in the representation used by the host systems C compiler.

symbol: a vector object (type bits `C_SYMBOL_TYPE`). Slots one and two contain the toplevel variable value and the print-name (a string) of the symbol, respectively.

port: a special vector object (type bits `C_PORT_TYPE`). The first slot contains a pointer to a file- stream, if this is a file-pointer, or NULL if not. The other slots contain housekeeping data used for this port.

structure: a vector object (type bits `C_STRUCTURE_TYPE`). The first slot contains a symbol that specifies the kind of structure this record is an instance of. The other slots contain the actual record items.

pointer: a special vector object (type bits `C_POINTER_TYPE`). The single slot contains a machine pointer.

tagged pointer: similar to a pointer (type bits `C_TAGGED_POINTER_TYPE`), but the object contains an additional slot with a tag (an arbitrary data object) that identifies the type of the pointer.

Data objects may be allocated outside of the garbage collected heap, as long as their layout follows the above mentioned scheme. But care has to be taken not to mutate these objects with heap-data (i.e. non-immediate objects), because this will confuse the garbage collector.

For more information see the header file `chicken.h`.

Previous: chicken-setup

Next: Bugs and limitations

43 Bugs and limitations

- Compiling large files takes too much time.
- If a known procedure has unused arguments, but is always called without those parameters, then the optimizer *repairs* the procedure in certain situations and removes the parameter from the lambda-list.
- `port-position` currently works only for input ports.
- Leaf routine optimization can theoretically result in code that thrashes, if tight loops perform excessively many mutations.

Previous: Data representation

Next: FAQ

44 Introduction

This is the list of Frequently Asked Questions about Chicken Scheme. If you have a question not answered here, feel free to post to the chicken-users mailing list; if you consider your question general enough, feel free to add it to this list.

45 General

45.1 Why yet another Scheme implementation?

Since Scheme is a relatively simple language, a large number of implementations exist and each has its specific advantages and disadvantages. Some are fast, some provide a rich programming environment. Some are free, others are tailored to specific domains, and so on. The reasons for the existence of CHICKEN are:

- CHICKEN is portable because it generates C code that runs on a large number of platforms.
- CHICKEN is extendable, since its code generation scheme and runtime system/garbage collector fits neatly into a C environment.
- CHICKEN is free and can be freely distributed, including its source code.
- CHICKEN offers better performance than nearly all interpreter based implementations, but still provides full Scheme semantics.
- As far as we know, CHICKEN is the first implementation of Scheme that uses Henry Baker's Cheney on the M.T.A concept.

45.2 What should I do if I find a bug?

Send e-mail to `felix@@call-with-current-continuation.org` with some hints about the problem, like version/build of the compiler, platform, system configuration, code that causes the bug, etc.

45.3 Why are values defined with {{define-foreign-variable}} or {{define-constant}} or {{define-inline}} not seen outside of the containing source file?

Accesses to foreign variables are translated directly into C constructs that access the variable, so the Scheme name given to that variable does only exist during compile-time. The same goes for constant- and inline-definitions: The name is only there to tell the compiler that this reference is to be replaced with the actual value.

45.4 How does {{cond-expand}} know which features are registered in used units?

Each unit used via `(declare (uses ...))` is registered as a feature and so a symbol with the unit-name can be tested by `cond-expand` during macro-expansion-time. Features registered using the `register-feature!` procedure are only available during run-time of the compiled file. You can use the `eval-when` form to register features at compile time.

45.5 Why are constants defined by {{define-constant}} not honoured in {{case}} constructs?

`case` expands into a cascaded `if` expression, where the first item in each arm is treated as a quoted list. So the `case` macro can not infer wether a symbol is to be treated as a constant-name (defined via define-constant) or a literal symbol.

45.6 How can I enable case sensitive reading/writing in user code?

To enable the read procedure to read symbols and identifiers case sensitive, you can set the parameter case-sensitivity to #t.

45.7 How can I change {{match-error-control}} during compilation?

Use eval-when, like this:

```
(eval-when (compile)
(match-error-control #:unspecified) )
```

45.8 Why doesn't CHICKEN support the full numeric tower by default?

The short answer:

```
% chicken-setup numbers
% csi -q
#;1> (use numbers)
```

The long answer:

There are a number of reasons for this:

- For most applications of Scheme fixnums (exact word-sized integers) and flonums (64-bit floating-point numbers) are more than sufficient;

- Interfacing to C is simpler;

- Dispatching of arithmetic operations is more efficient.

There is an extension based on the GNU Multiprecision Package that implements most of the full numeric tower, see http://www.call-with-current-continuation.org/eggs/numbers.html.

45.9 How can I specialize a generic function method to match instances of every class?

Specializing a method on <object> doesn't work on primitive data objects like numbers, strings, etc. so for example

```
(define-method (foo (x <my-class>)) ...)
(define-method (foo (x <object>)) ...)
```

```
(foo 123)
```

will signal an error, because to applicable method can be found. To specialize a method for primitive objects, use `<top>`:

```
(define-method (foo (x <top>)) ...)
```

45.10 Does CHICKEN support native threads?

Currently native threads are not supported. The runtime system is not reentrant, and the garbage-collection algorithm would be made much more complicated, since the location of every object (whether it is allocated on the stack or on the heap or completely outside the GC-able data space) has to be checked - this would be rather complex and inefficient in a situation where multiple threads are involved.

46 Platform specific

46.1 How do I generate a DLL under MS Windows (tm) ?

Use `csc` in combination with the `-dll` option:

```
C:\> csc foo.scm -dll
```

46.2 How do I generate a GUI application under Windows(tm)?

Invoke `csc` with the `-windows` option. Or pass the `-DC_WINDOWS_GUI` option to the C compiler and link with the GUI version of the runtime system (that's `libchicken-gui[-static].lib`. The GUI runtime displays error messages in a message box and does some rudimentary command-line parsing.

46.3 Compiling very large files under Windows with the Microsoft C compiler fails with a message indicating insufficient heap space.

It seems that the Microsoft C compiler can only handle files up to a certain size, and it doesn't utilize virtual memory as well as the GNU C compiler, for example. Try closing running applications. If that fails, try to break up the Scheme code into several library units.

46.4 When I run {{csi}} inside an emacs buffer under Windows, nothing happens.

Invoke `csi` with the `-:c` runtime option. Under Windows the interpreter thinks it is not running under control of a terminal and doesn't print the prompt and does not flush the output stream properly.

46.5 I load compiled code dynamically in a Windows GUI application and it crashes.

Code compiled into a DLL to be loaded dynamically must be linked with the same runtime system as the loading application. That means that all dynamically loaded entities (including extensions built and installed with `chicken-setup`) must be compiled with the `-windows` `csc` option.

46.6 On Windows, {{csc.exe}} seems to be doing something wrong.

The Windows development tools include a C# compiler with the same name. Either invoke `csc.exe` with a full pathname, or put the directory where you installed CHICKEN in front of the MS development tool path in the `PATH` environment variable.

47 Customization

47.1 How do I run custom startup code before the runtime-system is invoked?

When you invoke the C compiler for your translated Scheme source program, add the C compiler option `-DC_EMBEDDED`, or pass `-embedded` to the `csc` driver program, so no entry-point function will be generated (`main()`). When your are finished with your startup processing, invoke:

```
CHICKEN_main(argc, argv, C_toplevel);
```

where `C_toplevel` is the entry-point into the compiled Scheme code. You should add the following declarations at the head of your code:

```
#include "chicken.h"
extern void C_toplevel(C_word,C_word,C_word) C_noret;
```

47.2 How can I add compiled user passes?

To add a compiled user pass instead of an interpreted one, create a library unit and recompile the main unit of the compiler (in the file `chicken.scm`) with an additional `uses` declaration. Then link all compiler modules and your (compiled) extension to create a new version of the compiler, like this (assuming all sources are in the current directory):

```
% cat userpass.scm
;;;; userpass.scm - My very own compiler pass

(declare (unit userpass))

;; Perhaps more user passes/extensions are added:
(let ([old (user-pass)])
(user-pass
(lambda (x)
(let ([x2 (do-something-with x)])
   if old (
              (old x2)
              x2) ) ) ) )

% csc -c -x userpass.scm
% csc chicken.scm -c -o chicken-extended.o -uses userpass
% gcc chicken-extended.o support.o easyffi.o compiler.o optimizer.o batch-drive
c-backend.o userpass.o `csc -ldflags -libs` -o chicken-extended
```

On platforms that support it (Linux ELF, Solaris, Windows + VC++), compiled code can be loaded via `-extend` just like source files (see `load` in the User's Manual).

48 Macros

48.1 Why doesn't my fancy macro work in compiled code?

Macro bodies that are defined and used in a compiled source-file are evaluated during compilation and so have no access to definitions in the compiled file. Note also that during compile-time macros are only available in the same source file in which they are defined. Files included via `include` are considered part of the containing file.

48.2 Why are macros not visible outside of the compilation unit in which they are defined?

Macros are defined during compile time, so when a file has been compiled, the definitions are gone. An exception to this rule are macros defined with `define-macro`, which are also visible at run-time, i.e. in `eval`. To use macros defined in other files, use the `include` special form.

49 Warnings and errors

49.1 Why does my program crash when I use callback functions (from Scheme to C and back to Scheme again)?

There are two reasons why code involving callbacks can crash out of no apparent reason:

1. It is important to use `foreign-safe-lambda/foreign-safe-lambda*` for the C code that is to call back into Scheme. If this is not done than sooner or later the available stack space will be exhausted.
2. If the C code uses a large amount of stack storage, or if Scheme-to-C-to-Scheme calls are nested deeply, then the available nursery space on the stack will run low. To avoid this it might be advisable to run the compiled code with a larger nursery setting, i.e. run the code with `-:s...` and a larger value than the default (for example `-:s300k`), or use the `-nursery` compiler option. Note that this can decrease runtime performance on some platforms.

49.2 Why does the linker complain about a missing function {{_C_..._toplevel}}?

This message indicates that your program uses a library-unit, but that the object-file or library was not supplied to the linker. If you have the unit `foo`, which is contained in `foo.o` than you have to supply it to the linker like this (assuming a GCC environment):

```
% csc program.scm foo.o -o program
```

49.3 Why does the linker complain about a missing function {{_C_toplevel}}?

This means you have compiled a library unit as an application. When a unit-declaration (as in `(declare (unit ...))`) is given, then this file has a specially named toplevel entry procedure. Just remove the declaration, or compile this file to an object-module and link it to your application code.

49.4 Why does my program crash when I compile a file with {{-unsafe}} or unsafe declarations?

The compiler option `-unsafe` or the declaration `(declare (unsafe))` disable certain safety-checks to improve performance, so code that would normally trigger an error will work unexpectedly or even crash the running application. It is advisable to develop and debug a program in safe mode (without unsafe declarations) and use this feature only if the application works properly.

49.5 Why do I get a warning when I define a global variable named {{match}}?

Even when the `match` unit is not used, the macros from that package are visible in the compiler. The reason for this is that macros can not be accessed from library units (only when explicitly evaluated in running code). To speed up macro-expansion time, the compiler and the interpreter both already provide the compiled `match-...` macro definitions. Macros shadowed lexically are no problem, but global definitions of variables named identically to (global) macros are useless - the macro definition shadows the global variable.

This problem can be solved using a different name or undefining the macro, like this:

```
(eval-when (compile eval) (undefine-macro! 'match))
```

49.6 Why don't toplevel-continuations captured in interpreted code work?

Consider the following piece of code:

```
(define k (call-with-current-continuation (lambda (k) k)))
(k k)
```

When compiled, this will loop endlessly. But when interpreted, `(k k)` will return to the read-eval-print loop! This happens because the continuation captured will eventually read the next toplevel expression from the standard-input (or an input-file if loading from a file). At the moment `k` was defined, the next expression was `(k k)`. But when `k` is invoked, the next expression will be whatever follows after `(k k)`. In other words, invoking a captured continuation will not rewind the file-position of the input source. A solution is to wrap the whole code into a `(begin ...)` expression, so all toplevel expressions will be loaded together.

49.7 Why does {{define-reader-ctor}} not work in my compiled program?

The following piece of code does not work as expected:

```
(eval-when (compile)
(define-reader-ctor 'integer->char integer->char) )
(print #,(integer->char 33))
```

The problem is that the compiler reads the complete source-file before doing any processing on it, so the sharp-comma form is encountered before the reader-ctor is defined. A possible solution is to include the file containing the sharp-comma form, like this:

```
(eval-when (compile)
(define-reader-ctor 'integer->char integer->char) )

(include "other-file")

;;; other-file.scm:
(print #,(integer->char 33))
```

49.8 Why do built-in units, such as srfi-1, srfi-18, and posix fail to load?

When you try to `use` a built-in unit such as `srfi-18`, you may get the following error:

```
#;1> (use srfi-18)
; loading library srfi-18 ...
Error: (load-library) unable to load library
srfi-18
"dlopen(libchicken.dylib, 9): image not found"                ;; on a Mac
"libchicken.so: cannot open shared object file: No such file or directory"  ;;
```

Another symptom is that `(require 'srfi-18)` will silently fail.

This typically happens because the Chicken libraries have been installed in a non-standard location, such as your home directory. The workaround is to explicitly tell the dynamic linker where to look for your libraries:

```
export DYLD_LIBRARY_PATH=~/scheme/chicken/lib:$DYLD_LIBRARY_PATH ;; Mac
export LD_LIBRARY_PATH=~/scheme/chicken/lib:$LD_LIBRARY_PATH    ;; Linux
```

49.9 How can I increase the size of the trace shown when runtime errors are detected?

When a runtime error is detected, Chicken will print the last entries from the trace of functions called (unless your executable was compiled with the `-no-trace` option. By default, only 16 entries will be shown. To increase this number pass the `-:aN` parameter to your executable.

50 Optimizations

50.1 How can I obtain smaller executables?

If you don't need `eval` or the stuff in the `extras` library unit, you can just use the `library` unit:

```
        (declare (uses library))
        (display "Hello world!\n")
```

(Don't forget to compile with the `-explicit-use` option) Compiled with Visual C++ this generates an executable of around 240 kilobytes. It is theoretically possible to compile something without the library, but a program would have to implement quite a lot of support code on its own.

50.2 How can I obtain faster executables?

There are a number of declaration specifiers that should be used to speed up compiled files: declaring `(standard-bindings)` is mandatory, since this enables most optimizations. Even if some standard procedures should be redefined, you can list untouched bindings in the declaration. Declaring `(extended-bindings)` lets the compiler choose faster versions of certain internal library functions. This might give another speedup. You can also use the the `usual-integrations` declaration, which is identical to declaring `standard-bindings` and `extended-bindings` (note that `usual-integrations` is set by default). Declaring `(block)` tells the compiler that global procedures are not changed outside the current compilation unit, this gives the compiler some more opportunities for optimization. If no floating point arithmetic is required, then declaring `(number-type fixnum)` can give a big performance improvement, because the compiler can now inline most arithmetic operations. Declaring `(unsafe)` will switch off most safety checks. If threads are not used, you can declare `(disable-interrupts)`. You should always use maximum optimizations settings for your C compiler. Good GCC compiler options on Pentium (and compatible) hardware are: `-Os -fomit-frame-pointer -fno-strict-aliasing` Some programs are very sensitive to the setting of the nursery (the first heap-generation). You should experiment with different nursery settings (either by compiling with the `-nursery` option or by using the `-:s...` runtime option).

50.3 Which non-standard procedures are treated specially when the {{extended-bindings}} or {{usual-integrations}} declaration or compiler option is used?

The following standard bindings are handled specially, depending on optimization options and compiler settings:

```
+ * - / quotient eq? eqv? equal? apply c...r values call-with-values
list-ref null? length not char? string? symbol? vector? pair? procedure?
boolean? number? complex? rational? real? exact? inexact? list? eof-object?
string-ref string-set! vector-ref vector-set! char=? char<? char>? char<=? char
char-numeric? char-alphabetic? char-whitespace? char-upper-case?
char-lower-case? char-upcae char-downcase list-tail assv memv memq assoc
member set-car! set-cdr! abs exp sin cos tan log asin acos atan sqrt
zero? positive? negative? vector-length string-length char->integer
integer->char inexact->exact = > < >= <= for-each map substring
string-append gcd lcm list exact->inexact string->number number->string
```

even? odd? remainder floor ceiling truncate round cons vector string string=? string-ci=? make-vector call-with-current-continuation

The following extended bindings are handled specially:

bitwise-and bitwise-ior bitwise-xor bitwise-not bit-set? add1 sub1 fx+ fx- fx* fx/ fxmod fx= fx> fx>= fixnum? fxneg fxmax fxmin fxand fxior fxxor fxnot fxshl fxshr flonum? fp+ fp- fp* fp/ atom? fp= fp> fp>= fpneg fpmax fpmin arithmetic-shift signum flush-output thread-specific thread-specific-set! not-pair? null-list? print print* u8vector->bytevector s8vector->bytevector u16vector->bytevector s16vector->bytevector u32vector->bytevector s32vector->bytevector f32vector->bytevector f64vector->bytevector block-ref byte-vector-length u8vector-length s8vector-length u16vector-length s16vector-length u32vector-length s32vector-length f32vector-length f64vector-length u8vector-ref s8vector-ref u16vector-ref s16vector-ref u32vector-ref s32vector-ref f32vector-ref f64vector-ref u8vector-set! s8vector-set! u16vector-set! s16vector-set! u32vector-set! s32vector-set! hash-table-ref block-set! number-of-slots first second third fourth null-pointer? pointer->object make-record-instance locative-ref locative-set! locative? locative->object identity cpu-time error call/cc any?

51 Garbage collection

51.1 Why does a loop that doesn't {{cons}} still trigger garbage collections?

Under CHICKENs implementation policy, tail recursion is achieved simply by avoiding to return from a function call. Since the programs is CPS converted, a continuous sequence of nested procedure calls is performed. At some stage the stack-space has to run out and the current procedure and its parameters (including the current continuation) are stored somewhere in the runtime system. Now a minor garbage collection occurs and rescues all live data from the stack (the first heap generation) and moves it into the the second heap generation. Than the stack is cleared (using a `longjmp`) and execution can continue from the saved state. With this method arbitrary recursion (in tail- or non-tail position) can happen, provided the application doesn't run out of heap-space. (The difference between a tail- and a non-tail call is that the tail-call has no live data after it invokes its continuation - and so the amount of heap-space needed stays constant)

51.2 Why do finalizers not seem to work in simple cases in the interpeter?

Consider the following interaction in CSI:

```
#;1> (define x '(1 2 3))
#;2> (define (yammer x) (print x " is dead"))
#;3> (set-finalizer! x yammer)
(1 2 3)
#;4> (gc #t)
157812
#;5> (define x #f)
#;6> (gc #t)
157812
#;7>
```

While you might expect objects to be reclaimed and "*(1 2 3) is dead*" printed, it won't happen: the literal list gets held in the interpreter history, because it is the result value of the set-finalizer! call. Running this in a normal program will work fine.

When testing finalizers from the interpreter, you might want to define a trivial macro such as

```
(define-macro (v x) `(begin (print ,x) (void)))
```

and wrap calls to `set-finalizer!` in it.

52 Interpreter

52.1 Does CSI support history and autocompletion?

CSI doesn't support it natively but it can be activated with the http://galinha.ucpel.tche.br/readline egg. After installing the egg, add the following to your `~/.csirc` or equivalent file:

```
; The regex egg is required for ##csi#name-of-symbols-starting-with
(require 'readline)
(current-input-port (make-gnu-readline-port))
(gnu-history-install-file-manager (string-append (or (getenv "HOME") ".") "/.cs
```

53 Extensions

53.1 How can I install Chicken eggs to a non-default location?

You can just set the `CHICKEN_REPOSITORY` environment variable. It should contain the path where you want eggs to be installed:

```
$ export CHICKEN_REPOSITORY=~/chicken/
$ chicken-setup extensionname
```

In order to make programs (including csi) see these eggs, you should set this variable when you run them. Alternatively, you can call the `repository-path` Scheme procedure before loading the eggs, as in:

```
(repository-path "/home/azul/chicken")
(use format-modular)
```

Note, however, that using `repository-path` as above hard-codes the location of your eggs in your source files. While this might not be an issue in your case, it might be safe to keep this configuration outside of the source code (that is, specifying it as an environment variable) to make it easier to maintain.

53.2 Can I install chicken eggs as a non-root user?

Yes, just install them in a directory you can write.

Previous: Bugs and limitations

Next: Acknowledgements

54 Acknowledgements

Many thanks to Nico Amtsberg, William Annis, Marc Baily, Peter Barabas, Jonah Beckford, Arto Bendiken, Peter Bex, Jean-FranÃ§ois Bignolles, Dave Bodenstab, Fabian BÃ¶hlke, T. Kurt Bond, Ashley Bone, Dominique Boucher, Terence Brannon, Roy Bryant, Adam Buchbinder, Hans Bulfone, Category 5, Taylor Campbell, Franklin Chen, Thomas Chust, Gian Paolo Ciceri, John Cowan, Grzegorz Chrupa&#322;a, James Crippen, Tollef Fog Heen, Alejandro Forero Cuervo, Linh Dang, Brian Denheyer, dgym, Chris Double, Jarod Eells, Petter Egesund, Steve Elkins, Daniel B. Faken, WIll Farr, Graham Fawcett, Fizzie, Kimura Fuyuki, Tony Garnock-Jones, Martin Gasbichler, Joey Gibson, Stephen C. Gilardi, Joshua Griffith, Johannes GrÃ¸dem, Damian Gryski, Mario Domenech Goulart, Andreas Gustafsson, Sven Hartrumpf, Jun-ichiro itojun Hagino, Matthias Heiler, Karl M. Hegbloom, William P. Heinemann, Bill Hoffman, Bruce Hoult, Hans HÃ¼bner, Markus HÃ¼lsmann, Goetz Isenmann, Paulo Jabardo, David Janssens, Christian Jaeger, Dale Jordan, Valentin Kamyshenko, Daishi Kato, Peter Keller, Brad Kind, Ron Kneusel, Matthias Koeppe, Krysztof Kowa&#322;czyk, Todd R. Kueny Sr, Goran Krampe, Micky Latowicki, John Lenz, Kirill Lisovsky, Kon Lovett, Dennis Marti, Charles Martin, Bob McIsaac, Alain Mellan, Eric Merrit, Perry Metzger, Scott G. Miller, Mikael, Bruce Mitchener, Chris Moline, Eric E. Moore, Julian Morrison, Dan Muresan, Lars Nilsson, Ian Oversby, o.t., Gene Pavlovsky, Levi Pearson, Nicolas Pelletier, Carlos Pita, Robin Lee Powell, Pupeno, Davide Puricelli, Doug Quale, Eric Raible, Joel Reymont, Andreas Rottman, David Rush, Lars Rustemeier, Daniel Sadilek, Oskar Schirmer, Burton Samograd, Reed Sheridan, Ronald SchrÃ¶der, Spencer Schumann, Alex Shinn, Shmul, Tony Sidaway, Jeffrey B. Siegal, Andrey Sidorenko, Michele Simionato, Volker Stolz, Dorai Sitaram, Robert Skeels, Jason Songhurst, Clifford Stein, Sunnan, Zbigniew Szadkowski, Mike Thomas, Minh Thu, Christian Tismer, Andre van Tonder, John Tobey, Henrik Tramberend, Vladimir Tsichevsky, Neil van Dyke, Sander Vesik, Panagiotis Vossos, Shawn Wagner, Peter Wang, Ed Watkeys, Thomas Weidner, Goeran Weinholt, Matthew Welland, Joerg Wittenberger, Peter Wright, Mark Wutka, Richard Zidlicky and Houman Zolfaghari for bug-fixes, tips and suggestions.

CHICKEN uses the PCRE regular expression package (http://www.pcre.org), which is written by Philip Hazel.

Special thanks to Brandon van Every for contributing the CMake support and for helping with Windows build issues.

Also special thanks to Benedikt Rosenau for his constant encouragement.

CHICKEN contains code from several people:

Eli Barzilay
: some performance tweaks used in TinyCLOS.

Mikael Djurfeldt
: topological sort used by compiler.

Marc Feeley
: pretty-printer.

Aubrey Jaffer
: implementation of `dynamic-wind`.

Richard O'Keefe
: sorting routines.

Olin Shivers
: implementation of `let-optionals[*]` and reference implementations of SRFI-1, SRFI-13 and SRFI-14.

Andrew Wilcox
: queues.

Andrew Wright
: pattern matcher.

Previous: FAQ

Next: Bibliography

55 Bibliography

Henry Baker: *CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A.*
http://home.pipeline.com/~hbaker1/CheneyMTA.html

Revised^5 Report on the Algorithmic Language Scheme
http://www.schemers.org/Documents/Standards/R5RS

Previous: Acknowledgements