reference-titlechicken-setup reference reference-pg160 reference-sntSection 7.6

CHICKEN

A practical and portable Scheme system
User's Manual
Version 1, Build 944

Felix L. Winkelmann

Table of Contents

1 Introduction

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. This principle, which is used by several existing compilers, achieves high portability because C is implemented on nearly all available platforms.

This package is distributed under the BSD license and as such is free to use and modify. An interpreter is also available and can be used as a scripting environment or for testing programs before compilation.

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, 7, 8, 9, 10, 13, 14, 16, 18, 23, 28, 30, 39 and 55.
- Lightweight threads based on first-class continuations
- Pattern matching with Andrew Wright's `match` package
- Record structures
- An object system with multiple inheritance, multimethods and a meta-object protocol
- Extended comment- and string-literal syntaxes
- Libraries for regular expressions, string handling, Common LISP style `format`, 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.

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 Section 5.4.4 [Substitution forms and macros], page 29).

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.

3 Using the compiler

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`. On UNIX-compatible systems, a shell script named `chicken-config` is supplied that emits the correct options for the host system's C compiler. Enter

```
chicken-config -help
```

on the command line for a list of available options. 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.

3.1 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 `-debug-level 0 -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-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
e	lists all exported toplevel bindings
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
F	show output of “easy” FFI parser
P	show execution of outer partitioning
Q	show execution of middle partitioning
R	show execution of inner partitioning
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`.
- `-debug-level 1` does nothing.

`-disable-c-syntax-checks`
: Disable basic syntax checking of embedded C code fragments.

`-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.

`-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-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.

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

`-ffi`
: Parse C/C++ code and generate Scheme bindings. This is effectively equivalent to wrapping the code in `#>! ... <#`.

`-ffi-custom`
: Parse C/C++ code and embed it into a `(foreign-parse/spec ...)` form. This is effectively equivalent to wrapping the code in `#>% ... <#`). Use the `-extend` or `-require-extension` options to provide a definition for the `foreign-parse/spec` macro.

`-ffi-parse`
: Parse C/C++ code and embed as if wrapped inside `#>? ... <#`.

`-ffi-define SYMBOL`
: Defines a macro that will be accessible in `foreign-parse` declarations.

`-ffi-include-path PATH`
: Set include path for “easy” FFI parser.

`-ffi-no-include`
: Don’t parse include files when encountered by the FFI parser.

`-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.

`-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`.

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

`-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 equivalent to `-optimize-leaf-routines`
- `-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.

`-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.

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

`-split NUMBER`
: Splits output into multiple C files that can be compiled seperately. The generated C files will be named `filename0`, ..., `filename<NUMBER-1>` with as many files as given in `NUMBER`.

`-split-level NUMBER`
: Specifies how hard the partitioning algorithm should work:

  - 0 Exit after first iteration (quickest)
  - 1 Exit when cost does not decrease by at least one-half (the default)
  - 2 Exit when cost does not change

`-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.

3.2 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 during startup.

`-: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.

`-: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` or `-debug-level 0` 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.

3.3 An example

To compile a Scheme program (assuming a UNIX-like environment) perform the following steps:

- Consider this Scheme source file, named `foo.scm`

  ```
  ;;; foo.scm

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

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

- Compile the file `foo.scm`

  ```
  % chicken foo.scm
  ```

- Compile the generated C file `foo.c`

  ```
  % gcc foo.c -o foo ‘chicken-config -cflags -libs‘
  ```

- Start the compiled program

  ```
  % foo
  3628800
  ```

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:

- Consider these two Scheme source files, named `foo.scm` and `bar.scm`

  ```
  ;;; foo.scm

  (declare (uses bar))

  (write (fac 10)) (newline)


  ;;; bar.scm

  (declare (unit bar))

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

- Compile the files `foo.scm` and `bar.scm`

  ```
  % chicken foo.scm
  % chicken bar.scm
  ```

- Compile the generated C files `foo.c` and `bar.c`

  ```
  % gcc -c foo.c ‘chicken-config -cflags‘
  % gcc -c bar.c ‘chicken-config -cflags‘
  ```

- Link the object files `foo.o` and `bar.o`

  ```
  % gcc foo.o bar.o -o foo ‘chicken-config -libs‘
  ```

- Start the compiled program

  ```
  % foo
  3628800
  ```

The declarations 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.

3.4 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: Section 5.6 [Parameters], page 36.

user-options-pass [parameter]

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.

user-read-pass [parameter]

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.

user-preprocessor-pass [parameter]

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.

user-pass [parameter]

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.

user-pass-2 [parameter]

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.

user-post-analysis-pass [parameter]

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`.

3.5 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 `library.c`, the compiled form of `library.scm`, which contains the core library. Note the use of the `-explicit-use -uses library` option in the invocation of `chicken` to make sure we only link in the core library - the `eval` and `extras` library units are not used in this example:

```
% chicken hello.scm -explicit-use -uses hello.c -optimize-level 2 -debug-level 0
```

A simple makefile is needed as well:

```
# Makefile for UNIX systems

hello: hello.o runtime.o library.o
```

```
       $(CC) -o hello hello.o runtime.o library.o -lm

hello.o: chicken.h
runtime.o: chicken.h
library.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 chicken.h
% 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 autoconf/automake/libtool.

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

4 Using the interpreter

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

4.1 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 Section 3.1 [Compiler command line format], page 4 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.

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

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

4.2 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).
- Windows:

  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.

4.3 Toplevel commands

The toplevel loop understands a number of special commands:

`,?`
: Show summary of available toplevel commands.

`,l FILENAME`
: Load file with given `FILENAME` (may be a symbol or string).

`,ln FILENAME`
: Load file 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`.

`,q`
: Quit the interpreter.

`,r`
: Show system information.

`,s STRING-OR-SYMBOL`
: Execute shell-command.

`,t EXP`
: Evaluate form and print elapsed time.

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

4.4 Macros and procedures implemented in the interpreter

Additional macros and procedures available in the interpreter are:

trace [syntax]

```
(trace NAME ...)
```

Switches tracing on for the procedures with the given names.

```
#;1> (fac 10)                       ==> 3628800
#;2> (trace fac)
#;3> (fac 3)
|(fac 3)
| (fac 2)
|  (fac 1)
|   (fac 0)
|   fac -> 1
|  fac -> 1
| fac -> 2
|fac -> 6                          ==> 6
#;4> (untrace fac)
#;5> (fac 3)                        ==> 6
```

untrace [syntax]

```
(untrace NAME ...)
```

Switches tracing of the given procedures off.

#[INDEX] [read syntax]

```
#
#INDEX
```

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 this facility is a reader macro and is implicitly quoted.

5 Supported language

5.1 Deviations from the standard

[2] Identifiers are by default case-sensitive.

[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:

1. 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.
2. 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, and otherwise to `#f`. The initializer is evaluated in an environment in which all previous formal arguments have been bound.
3. 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 keyword-formal-arguments, the it shall be an error if there are any remaining actual arguments.
4. 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 the second is the 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, and otherwise `#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.

[4.3] `syntax-rules` macros are not provided but available separately.

[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)))
```

[6.2.4] 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] The routines `complex?`, `real?` and `rational?` are identical to the standard procedure `number?`. The procedures `numerator`, `denominator` and `rationalize` are not implemented. Also not implemented are all procedures related to complex numbers.

[6.2.6] The procedure `string->number` does not obey read/write invariance on inexact numbers.

[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.

5.2 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 Section 5.3 [Non standard read syntax], page 23.

[4] Numerous non-standard macros are provided. See Section 5.4 [Non-standard macros and special forms], page 25 for more information.

[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) ) )
```

[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` in Section 5.7.16 [User-defined named characters], page 49. 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`, 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`.

5.3 Non standard read syntax

#| ... |# [read syntax]

A multiline “block” comment. May be nested. Implements SRFI-30)

#;EXPRESSION [read syntax]

Treats `EXPRESSION` as a comment.

#,(CONSTRUCTORNAME DATUM ...) [read syntax]

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

#'EXPRESSION [read syntax]

An abbreviation for `(syntax EXPRESSION)`.

#$EXPRESSION [read syntax]

An abbreviation for `(location EXPRESSION)`.

#:SYMBOL [read syntax]

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

#<<TAG [read syntax]

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.")
```

#<#TAG [read syntax]

Similar to `#<<`, but allows substitution of embedded Scheme expressions prefixed with `#` and optionally enclosed in `{ ... }`. 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")
```

#> ... <# [read syntax]

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

#>? ... <# [read syntax]

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

#>! ... <# [read syntax]

Abbreviation for

```
(declare
  (foreign-declare " ... ")
  (foreign-parse " ... ") )
```

#>$ ... <# [read syntax]

An abbreviation for

```
(foreign-parse " ... ")
```

#>% ... <# [read syntax]

An abbreviation for

```
(foreign-parse/spec " ... ")
```

#%... [read syntax]

Reads like a normal symbol.

#!... [read syntax]

Treated as a commment and ignores everything up the end of the current line. The keywords `#!optional`, `#!rest` and `#!key` are handled separately and returned as normal symbols. The special (self-evaluating) symbol `#!eof` is read as the end-of-file object.

#cs... [read syntax]

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

#ci... [read syntax]

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

5.4 Non-standard macros and special forms

5.4.1 Making extra libraries and extensions available

require-extension [syntax]
use [syntax]

```
(require-extension ID ...)
(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 SRFI-55 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`, then nothing is done.
- If `ID` names one of syntactic extensions `chicken-match-macros chicken-more-macros chicken-default-entry-points chicken-entry-points chicken-ffi-macros`, then this extension will be loaded.
- If `ID` names one of the core library units shipped with CHICKEN, then a `(load-library 'ID)` will be performed. If one of those libraries define specific syntax (`match srfi-13`), then the required source file defining the syntax will be loaded.
- If `ID` names an installed extension with the `syntax` or `require-at-runtime` attribute, then the equivalent of `(require-for-syntax 'ID)` is performed.
- 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`, then nothing is done.
- If `ID` names one of syntactic extensions `chicken-match-macros chicken-more-macros chicken-default-entry-points chicken-entry-points 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, then a `(declare (uses ID))` is generated. If one of those libraries define specific syntax (`match srfi-13`), then the required source file defining the syntax will be loaded at compile-time, making the syntactic extension available in compiled code.
- If `ID` names an installed extension with the `syntax` or `require-at-runtime` attribute, then the equivalent of `(require-for-syntax 'ID)` is performed.
- 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).

See also: `set-extension-specifier!`

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 if the form

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

is 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).

5.4.2 Binding forms for optional arguments

: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
```

case-lambda [syntax]

```
(case-lambda (LAMBDA-LIST1 EXP1 ...) ...)
```

SRFI-16. 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

let-optionals [syntax]
let-optionals* [syntax]

```
(let-optionals ARGS ((VAR1 DEFAULT1) ...) BODY ...)
(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 3))
  (list a b c) )                              ==> (one two 3)
(let-optionals* '(one two) ((a 1) (b 2) (c a))
  (list a b c) )                              ==> (one two one)
```

5.4.3 Other binding forms

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 .

cut [syntax]
cute [syntax]

```
(cut SLOT ...)
(cute SLOT ...)
```

Syntactic sugar for specializing parameters.

define-values [syntax]

```
(define-values (NAME ...) EXP)
```

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

fluid-let [syntax]

```
(fluid-let ((VAR1 X1) ...) BODY ...)
```

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

let-values [syntax]

```
(let-values (((NAME ...) EXP) ...) BODY ...)
```

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

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
```

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
```

parameterize [syntax]

```
(parameterize ((PARAMETER1 X1) ...) BODY ...)
```

Binds the parameters `PARAMETER1 ...` dynamically to the values `X1 ...` during execution of `BODY ...`. (see also: `make-parameter` in Section 5.6 [Parameters], page 36). Note that `PARAMETER` may be any expression that evaluates to a parameter procedure.

receive [syntax]

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

set!-values [syntax]

```
(set!-values (NAME ...) EXP)
```

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

5.4.4 Substitution forms and macros

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.

define-inline [syntax]

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

define-macro [syntax]

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

5.4.5 Conditional forms

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.

unless [syntax]

```
(unless TEST EXP1 EXP2 ...)
```

Equivalent to:

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

when [syntax]

```
(when TEST EXP1 EXP2 ...)
```

Equivalent to:

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

5.4.6 Record structures

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
```

define-record-printer [syntax]

```
(define-record-printer (NAME RECORDVAR PORTVAR) BODY ...)
(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.

define-record-type [syntax]

```
(define-record-type NAME (CONSTRUCTOR TAG ...) PREDICATE
                        (FIELD ACCESSOR [MODIFIER]) ...)
```

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

5.4.7 Other forms

assert [syntax]

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

Signal 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.

cond-expand [syntax]

```
(cond-expand FEATURE-CLAUSE ...)
```

SRFI-0. Expands by selecting feature clauses. Predefined feature-identifiers are `srfi-0`, `srfi-2`, `srfi-6`, `srfi-8`, `srfi-9`, `srfi-10`, and `chicken`. If the source file containing this form is currently compiled, the feature `compiling` is defined. For further information, see the documentation for SRFI-0 This form is allowed to appear in non-toplevel expressions.

critical-section [syntax]

```
(critical-section BODY ...)
```

Evaluate `BODY ...` with timer-interrupts temporarily disabled.

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.

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.

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.

nth-value [syntax]

```
(nth-value N EXP)
```

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

time [syntax]

```
(time EXP1 ...)
```

Evaluates `EXP1 ...` and print elapsed time and memory information. The result of the last expression is returned.

5.5 Declarations

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:

always-bound [declaration specifier]

```
(always-bound SYMBOL ...)
```

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

block [declaration specifier]

```
(block)
```

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

block-global [declaration specifier]
hide [declaration specifier]

```
(block-global SYMBOL ...)
```

```
(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.

bound-to-procedure [declaration specifier]

```
(bound-to-procedure SYMBOL ...)
```

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

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.

check-c-syntax [declaration specifier]

```
(check-c-syntax)
(not check-c-syntax)
```

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

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.

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.

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.

foreign-declare [declaration specifier]

```
(foreign-declare STRING ...)
```

Include given strings verbatim into header of generated file.

foreign-parse [declaration specifier]

```
(foreign-parse STRING ...)
```

Parse given strings and generate foreign-interface bindings. See Section 6.7 [The Easy Foreign Function Interface], page 133 for more information.

disable-interrupts [declaration specifier]

```
(disable-interrupts)
(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.

inline [declaration specifier]

```
(inline)
(not inline)
(inline IDENTIFIER ...)
(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.

inline-limit [declaration specifier]

```
(inline-limit THRESHOLD)
```

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

interrupts-enabled [declaration specifier]

```
(interrupts-enabled)
```

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

lambda-lift [declaration specifier]

```
(lambda-lift)
```

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

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.

no-argc-checks [declaration specifier]

```
(no-argc-checks)
```

Disables argument count checking.

no-bound-checks [declaration specifier]

```
(no-bound-checks)
```

Disables the bound-checking of toplevel bindings.

no-procedure-checks [declaration specifier]

```
(no-procedure-checks)
```

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

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.

TYPE [declaration specifier]
fixnum-arithmetic [declaration specifier]

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

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

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.

standard-bindings [declaration specifier]

```
(standard-bindings SYMBOL ...)
(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.

extended-bindings [declaration specifier]

```
(extended-bindings SYMBOL ...)
(not extended-bindings SYMBOL ...)
```

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

usual-integrations [declaration specifier]

```
(usual-integrations SYMBOL ...)
(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.

unit [declaration specifier]

```
(unit SYMBOL)
```

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

unsafe [declaration specifier]

```
(unsafe)
(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))
```

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.

5.6 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

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.

case-sensitive [parameter]

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

dynamic-load-libraries [parameter]

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`.

command-line-arguments [parameter]

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

exit-handler [parameter]

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.

eval-handler [parameter]

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.

force-finalizers [parameter]

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`.

implicit-exit-handler [parameter]

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 do nothing, or, if one or more entry-points were defined (see: Section 6.3 [Entry points], page 125) to enter a loop that waits for callbacks from the host program.

keyword-style [parameter]

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.

load-verbose [parameter]

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

repl-prompt [parameter]

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> ")`.

reset-handler [parameter]

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.

5.7 Unit library

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

5.7.1 Arithmetic

add1 [procedure]
sub1 [procedure]

```
(add1 N)
(sub1 N)
```

Adds/subtracts 1 from `N`.

bitwise-and [procedure]
bitwise-ior [procedure]
bitwise-xor [procedure]
bitwise-not [procedure]
arithmetic-shift [procedure]

```
(bitwise-and N1 ...)
(bitwise-ior N1 ...)
(bitwise-xor N1 ...)
(bitwise-not N)
(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).

fixnum? [procedure]

```
(fixnum? X)
```

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

fx+ [procedure]
fx- [procedure]
fx* [procedure]
fx/ [procedure]
fxmod [procedure]
fxneg [procedure]
fxmin [procedure]
fxmax [procedure]
fx= [procedure]
fx> [procedure]
fx< [procedure]
fx>= [procedure]
fx<= [procedure]
fxand [procedure]
fxior [procedure]
fxxor [procedure]
fxnot [procedure]
fxshl [procedure]
fxshr [procedure]

```
(fx+ N1 N2)
```

```
(fx- N1 N2)
(fx* N1 N2)
(fx/ N1 N2)
(fxmod N1 N2)
(fxneg N)
(fxmin N1 N2)
(fxmax N1 N2)
(fx= N1 N2)
(fx> N1 N2)
(fx< N1 N2)
(fx>= N1 N2)
(fx<= N1 N2)
(fxand N1 N2)
(fxior N1 N2)
(fxxor N1 N2)
(fxnot N)
(fxshl N1 N2)
(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.

fp+ [procedure]
fp- [procedure]
fp* [procedure]
fp/ [procedure]
fpneg [procedure]
fpmin [procedure]
fpmax [procedure]
fp= [procedure]
fp> [procedure]
fp< [procedure]
fp>= [procedure]
fp<= [procedure]

```
(fp+ N1 N2)
(fp- N1 N2)
(fp* N1 N2)
(fp/ N1 N2)
(fpneg N)
(fpmin N1 N2)
(fpmax N1 N2)
(fp= N1 N2)
(fp> N1 N2)
(fp< N1 N2)
(fp>= N1 N2)
(fp<= N1 N2)
```

Arithmetic floating-point operations. These procedures do not check their arguments, so non-flonum parameters will result in incorrect results. On division by zero, `fp/` signals a condition of kind `(exn arithmetic)`.

signum [procedure]

```
(signum N)
```

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

5.7.2 File Input/Output

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.

flush-output [procedure]

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

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

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.

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.

set-port-name! [procedure]

```
(set-port-name! PORT STRING)
```

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

5.7.3 Files

delete-file [procedure]

```
(delete-file STRING)
```

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

file-exists? [procedure]

```
(file-exists? STRING)
```

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

pathname-directory-separator [variable]

Contains the directory-separator character for pathnames on this platform.

pathname-extension-separator [variable]

Contains the extension-separator character for pathnames on this platform.

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.

5.7.4 String ports

get-output-string [procedure]

```
(get-output-string PORT)
```

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

open-input-string [procedure]

```
(open-input-string STRING)
```

Returns a port for reading from `STRING`.

open-output-string [procedure]

```
(open-output-string)
```

Returns a port for accumulating output in a string.

5.7.5 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.

features [procedure]

```
(features)
```

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

test-feature? [procedure]

```
(test-feature? ID ...)
```

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

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.

unregister-feature! [procedure]

```
(unregister-feature! FEATURE ...)
```

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

5.7.6 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.

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.

keyword? [procedure]

```
(keyword? X)
```

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

keyword->string [procedure]

```
(keyword->string KEYWORD)
```

Transforms `KEYWORD` into a string.

string->keyword [procedure]

```
(string->keyword STRING)
```

Returns a keyword with the name `STRING`.

5.7.7 Exceptions

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

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
```

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.

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 error-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 and false, no error is signalled and the accessor returns `#g`.

5.7.8 Environment information and system interface

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.

exit [procedure]

```
(exit [CODE])
```

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

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
unknown
```

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.

errno [procedure]

```
(errno)
```

Returns the error code of the last system call.

getenv [procedure]

```
(getenv STRING)
```

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

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
```

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
```

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
```

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`.

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.

system [procedure]

```
(system STRING)
```

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

5.7.9 Execution time

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.

current-milliseconds [procedure]

```
(current-milliseconds)
```

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

current-seconds [procedure]

```
(current-seconds)
```

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

5.7.10 Interrupts and error-handling

enable-interrupts [procedure]
disable-interrupts [procedure]

```
(enable-interrupts)
(disable-interrupts)
```

Enables/disables processing of timer-interrupts and interrupts caused by signals.

```
(disable-interrupts)
(disable-interrupts)
(enable-interrupts)
; <interrupts still disabled - call enable-interrupts once more>
```

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.

error [procedure]

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

Prints error message, writes all extra arguments to the value of `(current-error-port)` and invokes the current value of `(error-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.

print-backtrace [procedure]

`(print-backtrace [PORT])`

Prints a backtrace of the procedure call history to `PORT`, which defaults to `(current-error-port)`. Backtrace information is only generated in compiled code, with a `-debug-level` >= 1.

print-error-message [procedure]

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

Prints an appropriate error message to `PORT` (which defaults to the value of `(current-error-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:"`.

reset [procedure]

`(reset)`

Reset program (Invokes `reset-handler`).

5.7.11 Garbage collection

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.

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.

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`.

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.

5.7.12 Other control structures

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`.

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`.

5.7.13 String utilities

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.

5.7.14 Generating uninterned symbols

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.

string->uninterned-symbol [procedure]

`(string->uninterned-symbol STRING)`

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

5.7.15 Standard Input/Output

port? [procedure]

```
(port? X)
```

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

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.

print* [procedure]

```
(print* EXP1 ...)
```

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

5.7.16 User-defined named characters

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
```

5.7.17 Vectors

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)`

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.

5.7.18 The *unspecified* value

void [procedure]

```
(void)
```

Returns an unspecified value.

5.7.19 Continuations

call/cc [procedure]

```
(call/cc PROCEDURE)
```

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

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.

continuation? [procedure]

```
(continuation? X)
```

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

continuation-graft [procedure]

```
(continuation-graft CONT THUNK)
```

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

continuation-return [procedure]

```
(continuation-return CONT VALUE ...)
```

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

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

5.8 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.

5.8.1 Loading code

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!"))
% chicken x.scm -quiet -dynamic
% gcc x.c -shared -fPIC ‘chicken-config -cflags -shared -libs‘ -o x.so
% csi -quiet
#;1> (load "x.so")
; loading x.so ...
#;2> (hello)
Hello!
#;3>
```

The second argument to `load` is ignored when loading compiled code. The same compiled object file can not be loaded more than once. If source code is loaded from a port, then that port is closed after all expressions have been read.

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.

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.

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.

5.8.2 Read-eval-print loop

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 `error-handler` to display a message, write any arguments to the value of `(current-error-port)` and reset.

5.8.3 Macros

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.

macro? [procedure]

```
(macro? SYMBOL)
```

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

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.

macroexpand-1 [procedure]

```
(macroexpand-1 X)
```

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

undefine-macro! [procedure]

```
(undefine-macro! SYMBOL)
```

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

syntax-error [procedure]

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

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

5.8.4 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.

repository-path [parameter]

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).

extension-info [procedure]

```
(extension-info 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-info` returns `#f`.

provide [procedure]

```
(provide ID ...)
```

Registers the extension IDs `ID ...` as loaded. This is mainly intended to provide aliases for certain extension identifiers.

provided? [procedure]

```
(provided? ID ...)
```

Returns `#t` if the extension with the IDs `ID ...` are currently loaded, or `#f` otherwise. Works also for feature-ids.

require [procedure]
require-for-syntax [procedure]

```
(require ID ...)
(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:

1. the current include path, which defaults to the pathnames given in `CHICKEN_INCLUDE_PATH` and `CHICKEN_HOME`. In case `ID` is a list, it is interpreted as a (relative) pathname.
2. the current directory

`ID` should be 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.

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-package-
```

Note that the handler has to be registered at compile time, if it is to be visible in compiled code.

5.8.5 Reader extensions

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 .

set-read-syntax! [procedure]

```
(set-read-syntax! 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)
```

set-dispatch-read-syntax! [procedure]

```
(set-dispatch-read-syntax! PROC)
```

Similar to `set-read-syntax!`, but allows defining new `#<CHAR> ...` reader syntax.

5.8.6 Eval

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)`.

5.9 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.

5.9.1 Lists

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`).

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?`.

atom? [procedure]

```
(atom? X)
```

Returns `#t` if `X` is a not list (`X` is not a pair nor the empty list).

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?`.

butlast [procedure]

`(butlast LIST)`

Returns a fresh list with all elements but the last of `LIST`.

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))
```

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)
```

flatten [procedure]

`(flatten LIST1 ...)`

Returns `LIST1 ...` concatenated together, with nested lists removed (flattened).

intersperse [procedure]

`(intersperse LIST X)`

Returns a new list with `X` placed between each element.

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)) )
```

shuffle [procedure]

`(shuffle LIST)`

Returns `LIST` with its elements sorted in a random order.

tail? [procedure]

`(tail? X LIST)`

Returns true if `X` is one of the tails (cdr's) of `LIST`.

5.9.2 String-port extensions

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`.

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.

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`.

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.

5.9.3 Formatted output

fprintf [procedure]
printf [procedure]
sprintf [procedure]

```
(fprintf PORT FORMATSTRING ARG ...)
(printf FORMATSTRING ARG)
(sprintf FORMATSTRING ARG ...)
```

Simple formatted output to a given port (`fprintf`), the value of `(current-output-port)` (`printf`) or a string (`sprintf`). 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

For more powerful output formatting, see the section about the `format` unit.

5.9.4 Hash tables

make-hash-table [procedure]

```
(make-hash-table [PRED [HASHFUNC [SIZE]]])
```

Creates and returns a hash-table with keys compared via `PRED`, which defaults to `eq?`. If `SIZE` is provided it specifies the initial size of the hash-table. If the hash-table fills above a certain size it is automatically resized to accommodate more entries. The `HASHFUNC` argument specifies the hashing function and defaults to `hash`.

hash-table-ref [procedure]

```
(hash-table-ref HASH-TABLE KEY [DEFAULT])
```

Returns the entry in the given hash-table under `KEY`. If no entry is stored in the table, `DEFAULT` is returned, or `#f` if `DEFAULT` is not given.

hash-table-set! [procedure]

```
(hash-table-set! HASH-TABLE KEY VALUE)
```

Adds or changes an entry in the given hash-table.

hash-table-exists? [procedure]

```
(hash-table-exists? HASH-TABLE KEY)
```

Returns `#t` if `HASH-TABLE` has an entry for `KEY` or `#f` otherwise.

clear-hash-table! [procedure]

```
(clear-hash-table! HASH-TABLE)
```

Erases all entries in the hash-table `HASH-TABLE`.

hash-table? [procedure]

```
(hash-table? X)
```

Returns `#t` if the argument is a hash-table.

hash-table->alist [procedure]

```
(hash-table->list HASH-TABLE)
```

Converts `HASH-TABLE` into an association-list.

alist->hash-table [procedure]

```
(alist->hash-table ALIST [PRED [SIZE]])
```

Creates a hash-table (optionally specialized with comparison predicate `PRED` and initial size `SIZE`) from the association-list `ALIST` where each key and value created in the hash-table is taken from the car and cdr of the elements in `ALIST`.

hash-table-keys [procedure]
hash-table-values [procedure]

```
(hash-table-keys HAST-TABLE)
(hash-table-values HAST-TABLE)
```

Return a list of the keys/values contained in `HASH-TABLE`. The order of the elements is unspecified.

hash-table-count [procedure]

```
(hash-table-count HASH-TABLE)
```

Returns the number of entries in the given hash-table.

hash-table-size [procedure]

```
(hash-table-size HASH-TABLE)
```

Returns the size of the hash-table (the actual size of the collection containing the hash-buckets). If you want to retrieve the number of items sotred, use `hash-table-count`.

hash-table-for-each [procedure]

```
(hash-table-for-each HASH-TABLE PROC)
```

Calls `PROC` which should expect two arguments. This procedure is called for each entry in the hash-table with the key and the value as parameters.

hash-table-remove! [procedure]

```
(hash-table-remove! HASH-TABLE KEY)
```

Removes an entry in the given hash-table.

hash-table-update! [procedure]

```
(hash-table-update! HASH-TABLE KEY PROC INIT)
```

If an entry with the given key exists in `HASH-TABLE`, replace it with `(PROC OLDVALUE)`. If no such entry exists, add a new entry with `INIT` as the initial value.

hash [procedure]

```
(hash X [BOUND])
```

Returns a hash-value for the given data object `X` in the bounds specified in `BOUND`, which defaults to some unspecified large fixnum.

get [procedure]

```
(get HASH-TABLE KEY PROP)
```

Returns the value of property `PROP` of the item `KEY` in `HASH-TABLE` . This facility can be used as a kind of "disembodied" property-list. If no entry named `KEY` is stored in the hash-table or if no property `PROP` for that key exists, `#f` is returned.

put! [procedure]

```
(put! HASH-TABLE KEY PROP VALUE)
```

Stores `VALUE` as property `PROP` under the item `KEY` in the given hash-table. Any previously existing value is overwritten.

5.9.5 Queues

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.

make-queue [procedure]

```
(make-queue)
```

Returns a newly created queue.

queue? [procedure]

```
(queue? X)
```

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

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.

queue-add! [procedure]

```
(queue-add! QUEUE X)
```

Adds `X` to the rear of `QUEUE`.

queue-empty? [procedure]

```
(queue-empty? QUEUE)
```

Returns `#t` if `QUEUE` is empty, or `#f` otherwise.

queue-first [procedure]

```
(queue-first QUEUE)
```

Returns the first element of `QUEUE`. If `QUEUE` is empty an error is signaled

queue-last [procedure]

```
(queue-last QUEUE)
```

Returns the last element of `QUEUE`. If `QUEUE` is empty an error is signaled

queue-remove! [procedure]

```
(queue-remove! QUEUE)
```

Removes and returns the first element of `QUEUE`. If `QUEUE` is empty an error is signaled

5.9.6 Sorting

merge [procedure]
merge! [procedure]

```
(merge LIST1 LIST2 LESS?)
(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.

sort [procedure]
sort! [procedure]

```
(sort SEQUENCE LESS?)
(sort! SEQUENCE LESS?)
```

Sort `SEQUENCE`, which should be a list or a vector. `sort!` is the destructive version of sort.

sorted? [procedure]

```
(sorted? SEQUENCE LESS?)
```

Returns true if the list or vector `SEQUENCE` is already sorted.

5.9.7 Random numbers

random [procedure]

```
(random N)
```

Returns an exact random integer from 0 to `N`-1.

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.

5.9.8 Input/Output extensions

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).

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.

pretty-print [procedure]
pp [procedure]

```
(pretty-print EXP [PORT])
(pp EXP [PORT])
```

Print expression nicely formatted. `PORT` defaults to the value of `(current-output-port)`.

pretty-print-width [parameter]
Specifies the maximal line-width for pretty printing, after which line wrap will occur.

read-file [procedure]

```
(read-file [FILE-OR-PORT])
```

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.

read-line [procedure]
write-line [procedure]

```
(read-line [PORT [LIMIT]])
(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.

read-lines [procedure]

```
(read-lines [PORT [MAX]])
```

Read `MAX` or fewer lines from `PORT`. `PORT` defaults to the value of `(current-input-port)`.

read-string [procedure]
write-string [procedure]

```
(read-string [NUM [PORT]])
(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-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.

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`.

with-input-from-port [procedure]

`(with-input-from-port PORT THUNK)`

Call procedure `THUNK` with the current input-port temporarily bound to `PORT`.

with-output-to-port [procedure]

`(with-output-to-port PORT THUNK)`

Call procedure `THUNK` with the current output-port temporarily bound to `PORT`.

5.9.9 Strings

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)) )
```

->string [procedure]

`(->string X)`

Returns a string-representation of `X`.

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")
```

string-compare3 [procedure]

`(string-compare3 STRING1 STRING2)`

string-compare3-ci [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.

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"))
```

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" "")
```

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.

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;"
```

substring=? [procedure]
substring-ci=? [procedure]

```
(substring=? STRING1 STRING2 [START1 [START2 [LENGTH]]])
(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).

substring-index [procedure]
substring-index-ci [procedure]

```
(substring-index WHICH WHERE [START])
```

```
(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`.

5.9.10 Combinators

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)
```

complement [procedure]

```
(complement PROC)
```

Returns a procedure that returns the boolean inverse of `PROC`.

```
(complement PROC) <=> (lambda (x) (not (PROC x)))
```

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))
```

conjoin [procedure]

```
(conjoin PRED ...)
```

Returns a procedure that returns `#t` if its argument satisfies the predicates `PRED ...`.

```
((conjoin odd? positive?) 33)   ==>  #t
((conjoin odd? positive?) -33)  ==>  #f
```

disjoin [procedure]

```
(disjoin PRED ...)
```

Returns a procedure that returns `#t` if its argument satisfies any predicate `PRED ...`.

```
((disjoin odd? positive?) 32)    ==>  #t
((disjoin odd? positive?) -32)   ==>  #f
```

flip [procedure]

```
(flip PROC)
```

Returns a two-argument procedure that calls `PROC` with its arguments swapped:

```
(flip PROC) <=> (lambda (x y) (PROC y x))
```

identity [procedure]

```
(identity X)
```

Returns its sole argument `X`.

project [procedure]

```
(project N)
```

Returns a procedure that returns its `N`th argument.

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
```

noop [procedure]

```
(noop X ...)
```

Ignores it's arguments, does nothing and returns an unspecified value.

5.9.11 Binary searching

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.

5.10 Unit srfi-1

List library, see the documentation for SRFI-1

5.11 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:

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:

u8vector->byte-vector [procedure]
s8vector->byte-vector [procedure]
u16vector->byte-vector [procedure]
s16vector->byte-vector [procedure]
u32vector->byte-vector [procedure]
s32vector->byte-vector [procedure]
f32vector->byte-vector [procedure]
f64vector->byte-vector [procedure]

```
(u8vector->byte-vector U8VECTOR)
(s8vector->byte-vector S8VECTOR)
(u16vector->byte-vector U16VECTOR)
(s16vector->byte-vector S16VECTOR)
(u32vector->byte-vector U32VECTOR)
(s32vector->byte-vector S32VECTOR)
(f32vector->byte-vector F32VECTOR)
(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.

byte-vector->u8vector [procedure]
byte-vector->s8vector [procedure]
byte-vector->u16vector [procedure]
byte-vector->s16vector [procedure]
byte-vector->u32vector [procedure]
byte-vector->s32vector [procedure]
byte-vector->f32vector [procedure]
byte-vector->f64vector [procedure]

```
(byte-vector->u8vector BYTE-VECTOR)
(byte-vector->s8vector BYTE-VECTOR)
(byte-vector->u16vector BYTE-VECTOR)
(byte-vector->s16vector BYTE-VECTOR)
(byte-vector->u32vector BYTE-VECTOR)
(byte-vector->s32vector BYTE-VECTOR)
(byte-vector->f32vector BYTE-VECTOR)
(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.

subu8vector [procedure]
subu16vector [procedure]
subu32vector [procedure]
subs8vector [procedure]
subs16vector [procedure]
subs32vector [procedure]
subf32vector [procedure]
subf64vector [procedure]

```
(subu8vector U8VECTOR FROM TO)
(subu16vector U16VECTOR FROM TO)
(subu32vector U32VECTOR FROM TO)
(subs8vector S8VECTOR FROM TO)
(subs16vector S16VECTOR FROM TO)
(subs32vector S32VECTOR FROM TO)
(subf32vector F32VECTOR FROM TO)
(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`.

5.12 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)
```

5.13 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.

5.14 Unit match

Andrew Wright's pattern matching package. 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.

match [syntax]
match-lambda [syntax]
match-lambda* [syntax]
match-let [syntax]
match-let* [syntax]
match-letrec [syntax]
match-define [syntax]

```
(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-define PAT EXP)
```

Match expression or procedure arguments with pattern and execute associated expressions. A Postscript manual is available .

match-error-control [procedure]

```
(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:

`#:error`
Signal an error. This is the default.

`#:match`
Signal an error and output the offending form.

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

`#: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)`.

Note: 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. The structure facility of the match package (`define-structure`) is not available.

5.15 Unit regex

This library unit provides support for regular expressions. The flavor depends on the particular installation platform:

- On UNIX systems that have PCRE (the Perl Compatible Regular Expression package) installed, PCRE is used.
- If PCRE is not available, and the C library provides regular expressions, these are used instead.
- on Windows (or of PCRE and libc regexes are not available), Dorai Sitaram's portable `pregexp` library is used.

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) )
```

pattern->regexp [procedure]

```
(pattern->regexp PATTERN)
```

Converts the file-pattern `PATTERN` into a regular expression.

```
(pattern->regexp "foo.*") ==> "foo\..*"
```

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.

Notes:

- regex doesn't allow (?:) cloisters (non-capturing groups) Currently this means if you use utf8 matching, individual "." matching will return extra submatches.
- pregexp doesn't allow a # comment w/o a trailing newline.

regexp? [procedure]

```
(regexp? X)
```

Returns `#t` if `X` is a precompiled regular expression, or `#f` otherwise.

string-match [procedure]
string-match-positions [procedure]

```
(string-match REGEXP STRING [START])
(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 ^ ... $.

string-search [procedure]
string-search-positions [procedure]

```
(string-search REGEXP STRING [START [RANGE]])
(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.

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")
```

string-substitute [procedure]

```
(string-substitute REGEXP SUBST STRING [INDEX])
```

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 INDEX 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)"
```

string-substitute* [procedure]

```
(string-substitute* 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 regular expression `MATCH` in `STRING` will be replaced by the string `REPLACEMENT`

```
(string-substitute* "<h1>Hello, world!</h1>"
                    '(("<[/A-Za-z0-9]+>" . "")))

==>  "Hello, world!"
```

Note that back-references like `\NUM` are currently not supported.

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]+:.*$")   ==>  "\\^\\[0-9\\]\\+:.\n.\\*\\$"
```

5.16 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))`.
- 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:

thread-deliver-signal! [procedure]

```
(thread-deliver-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.

thread-quantum [procedure]

`(thread-quantum THREAD)`

Returns the quantum of `THREAD`, which is an exact integer specifying the approximate time-slice of the thread.

thread-quantum-set! [procedure]

`(thread-quantum-set! THREAD QUANTUM)`

Sets the quantum of `THREAD` to `QUANTUM`.

5.17 Unit format

format [procedure]

`(format DESTINATION FORMAT-STRING . ARGUMENTS)`

An almost complete implementation of Common LISP format description according to the CL reference book **Common LISP** from Guy L. Steele, Digital Press. This code was originally part of SLIB. The author is Dirk Lutzebaeck.

Returns `#t`, `#f` or a string; has side effect of printing according to `FORMAT-STRING`. If `DESTINATION` is `#t`, the output is to the current output port and `#t` is returned. If `DESTINATION` is `#f`, a formatted string is returned as the result of the call. If `DESTINATION` is a string, `DESTINATION` is regarded as the format string; `FORMAT-STRING` is then the first argument and the output is returned as a string. If `DESTINATION` is a number, the output is to the value of `(current-error-port)`. Otherwise `DESTINATION` must be an output port and `#t` is returned.

`FORMAT-STRING` must be a string. In case of a formatting error format returns `#f` and prints a message on the value of `(current-error-port)`. Characters are output as if the string were output by the `display` function with the exception of those prefixed by a tilde (~). For a detailed description of the `FORMAT-STRING` syntax please consult a Common LISP format reference manual. A list of all supported, non-supported and extended directives can be found in `format.txt`.

This unit uses definitions from the `extras` unit.

`format` implements SRFI-28

5.18 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)`.

5.18.1 Directories

change-directory [procedure]

`(change-directory NAME)`

Changes the current working directory to `NAME`.

current-directory [procedure]

`(current-directory)`

Returns the name of the current working directory.

create-directory [procedure]

`(create-directory NAME)`

Creates a directory with the pathname `NAME`.

delete-directory [procedure]

`(delete-directory NAME)`

Deletes the directory with the pathname `NAME`. The directory has to be empty.

directory [procedure]

`(directory PATHNAME)`

Returns a list with all files that are contained in the directory with the name `PATHNAME`.

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.

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).

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.

5.18.2 Pipes

call-with-input-pipe [procedure]
call-with-output-pipe [procedure]

```
(call-with-input-pipe CMDLINE PROC [MODE])
(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.

close-input-pipe [procedure]
close-output-pipe [procedure]

`(close-input-pipe PORT)`

```
(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.

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.

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`.

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`.

pipe/buf [limit]

This variable contains the maximal number of bytes that can be written atomically into a pipe or FIFO.

with-input-from-pipe [procedure]
with-output-to-pipe [procedure]

```
(with-input-from-pipe CMDLINE THUNK [MODE])
(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
) ) )
```

5.18.3 Fifos

create-fifo [procedure]

```
(create-fifo FILENAME [MODE])
```

Creates a FIFO with the name `FILENAME` and the permission bits `MODE`, which defaults to

```
(+ perm/irwxu perm/irwxg perm/irwxo)
```

fifo? [procedure]

```
(fifo? FILENAME)
```

Returns `#t` if the file with the name `FILENAME` names a FIFO.

5.18.4 File descriptors and low-level I/O

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.

file-close [procedure]

```
(file-close FILENO)
```

Closes the input/output file with the file-descriptor `FILENO`.

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`.

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)))
```

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.

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 `()`.

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.

fileno/stdin [file descriptor]
fileno/stdout [file descriptor]
fileno/stderr [file descriptor]

These variables contain file-descriptors for the standard I/O files.

open/rdonly [flag]
open/wronly [flag]
open/rdwr [flag]
open/read [flag]
open/write [flag]
open/creat [flag]
open/append [flag]
open/excl [flag]
open/noctty [flag]
open/nonblock [flag]
open/trunc [flag]
open/sync [flag]
open/fsync [flag]
open/binary [flag]
open/text [flag]

Flags for use with `file-open`.

open-input-file* [procedure]
open-output-file* [procedure]

```
(open-input-file* FILENO [OPENMODE])
(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).

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.

5.18.5 Retrieving file attributes

file-access-time [procedure]
file-change-time [procedure]
file-modification-time [procedure]

```
(file-access-time FILE)
(file-change-time FILE)
(file-modification-time FILE)
```

Returns time (in seconds) of the last acces, modification or change of `FILE`. `FILE` may be a filename or a file-descriptor. If the file does not exist, an error is signaled.

file-stat [procedure]

```
(file-stat FILE [LINK])
```

Returns a 9-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`). 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 resolved).

file-position [procedure]

```
(file-position FILE)
```

Returns the current file position of `FILE`, which should be a port or a file-descriptor.

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.

regular-file? [procedure]

```
(regular-file? FILENAME)
```

Returns true, if `FILENAME` names a regular file (not a directory or symbolic link).

5.18.6 Changing file attributes

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.

set-file-position! [procedure]

```
(set-file-position! FILE POSITION [WHENCE])
```

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)`

5.18.7 Processes

current-process-id [procedure]

```
(current-process-id)
```

Returns the process ID of the current process.

parent-process-id [procedure]

```
(parent-process-id)
```

Returns the process ID of the parent of the current process.

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 spplied, 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.

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.

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.

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`.

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.

process [procedure]

```
(process COMMANDLINE [ARGUMENTLIST [ENVIRONMENT]])
```

Passes the string `COMMANDLINE` to the host-system's shell that is invoked as a sub-process 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.

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.

5.18.8 Hard and symbolic links

symbolic-link? [procedure]

```
(symbolic-link? FILENAME)
```

Returns true, if `FILENAME` names a symbolic link.

create-symbolic-link [procedure]

```
(create-symbolic-link OLDNAME NEWNAME)
```

Creates a symbolic link with the filename `NEWNAME` that points to the file named `OLDNAME`.

read-symbolic-link [procedure]

```
(read-symbolic-link FILENAME)
```

Returns the filename to which the symbolic link `FILENAME` points.

file-link [procedure]

```
(file-link OLDNAME NEWNAME)
```

Creates a hard link from `OLDNAME` to `NEWNAME` (both strings).

5.18.9 Permissions, owners, users and groups

file-owner [procedure]

```
(file-owner FILE)
```

Returns the user-id of `FILE`. `FILE` may be a filename or a file-descriptor.

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.

file-read-access? [procedure]
file-write-access? [procedure]
file-execute-access? [procedure]

```
(file-read-access? FILENAME)
(file-write-access? FILENAME)
```

```
(file-execute-access? FILENAME)
```

These procedures return `#t` if the current user has read, write or execute permissions on the file named `FILENAME`.

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.

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.

current-user-id [procedure]
current-group-id [procedure]
current-effective-user-id [procedure]
current-effective-group-id [procedure]

```
(current-user-id)
(current-group-id)
(current-effective-user-id)
(current-effective-group-id)
```

Return the user- and group-ids of the current process.

process-group-id [procedure]

```
(process-group-id PID)
```

Returns the process group ID of the process specified by `PID`.

group-information [procedure]

```
(group-information GROUP)
```

If `GROUP` specifies a valid group-name or group-id, then this procedure returns 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.

get-groups [procedure]

```
(get-groups)
```

Returns a list with the supplementary group IDs of the current user.

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.

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.

perm/irusr [permission bits]
perm/iwusr [permission bits]
perm/ixusr [permission bits]
perm/irgrp [permission bits]
perm/iwgrp [permission bits]
perm/ixgrp [permission bits]
perm/iroth [permission bits]
perm/iwoth [permission bits]
perm/ixoth [permission bits]
perm/irwxu [permission bits]
perm/irwxg [permission bits]
perm/irwxo [permission bits]
perm/isvtx [permission bits]
perm/isuid [permission bits]
perm/isgid [permission bits]

These variables contain permission bits as used in `change-file-mode`.

set-user-id! [procedure]

`(set-user-id! UID)`

Sets the effective user id of the current process to `UID`, which should be a positive integer.

set-group-id! [procedure]

`(set-group-id! GID)`

Sets the effective group id of the current process to `GID`, which should be a positive integer.

set-process-group-id! [procedure]

`(set-user-id! PID PGID)`

Sets the process group ID of the process specifed by `PID` to `PGID`.

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.

create-session [procedure]

`(create-session)`

Creates a new session if the calling process is not a process group leader and returns the session ID.

5.18.10 Record locking

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.

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.

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.

file-unlock [procedure]

```
(file-unlock LOCK)
```

Unlocks the previously locked portion of a file given in `LOCK`.

5.18.11 Signal handling

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.

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 `SIGNAL`. `PROC` is called with the signal number as its sole argument. If the argument `PROC` is `#f` then this signal will be ignored. Note that is is unspecified in which thread of execution the signal handler will be invoked.

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.

signal/term [signal code]
signal/kill [signal code]
signal/int [signal code]
signal/hup [signal code]
signal/fpe [signal code]
signal/ill [signal code]
signal/segv [signal code]
signal/abrt [signal code]
signal/trap [signal code]
signal/quit [signal code]
signal/alrm [signal code]
signal/vtalrm [signal code]
signal/prof [signal code]
signal/io [signal code]
signal/urg [signal code]
signal/chld [signal code]
signal/cont [signal code]
signal/stop [signal code]
signal/tstp [signal code]
signal/pipe [signal code]
signal/xcpu [signal code]
signal/xfsz [signal code]
signal/usr1 [signal code]
signal/usr2 [signal code]
signal/winch [signal code]

These variables contain signal codes for use with `process-signal` or `set-signal-handler!`.

5.18.12 Environment access

current-environment [procedure]

```
(current-environment)
```

Returns a association list of the environment variables and their current values.

Note: Under Mac OS X, this procedure always returns the empty list.

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.

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.

5.18.13 Memory mapped I/O

memory-mapped-file? [pocedure]

`(memory-mapped-file? X)`

Returns `#t`, if `X` is an object representing a memory mapped file, or `#f` otherwise.

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.

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.

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.

5.18.14 Time routines

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: the number of seconds after the minute (0 - 59)
- minutes: the number of minutes after the hour (0 - 59)
- hours: the number of hours past midnight (0 - 23)
- mday: the day of the month (1 - 31)
- month: the number of months since january (0 - 11)
- year: the number of years since 1900
- wday: the number of days since Sunday (0 - 6)
- yday: the number of days since January 1 (0 - 365)

- dstflag: a flag that is true if Daylight Saving Time is in effect at the time described.
- timezone: the difference between UTC and the latest local standard time, in seconds west of UTC.

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"`.

seconds->utc-time [procedure]

```
(seconds->utc-time SECONDS)
```

Similar to `seconds->local-time`, but interpretes `SECONDS` as UTC time.

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"`.

5.18.15 *Raw* exit

_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`.

5.18.16 ERRNO values

errno/perm	[error code]
errno/noent	[error code]
errno/srch	[error code]
errno/intr	[error code]
errno/io	[error code]
errno/noexec	[error code]
errno/badf	[error code]
errno/child	[error code]
errno/nomem	[error code]
errno/acces	[error code]
errno/fault	[error code]
errno/busy	[error code]
errno/notdir	[error code]
errno/isdir	[error code]
errno/inval	[error code]
errno/mfile	[error code]
errno/nospc	[error code]
errno/spipe	[error code]
errno/pipe	[error code]
errno/again	[error code]
errno/rofs	[error code]
errno/exist	[error code]
errno/wouldblock	[error code]

These variables contain error codes as returned by `errno`.

5.18.17 Finding files

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.

5.18.18 Getting the hostname and system information

get-host-name [procedure]

`(get-host-name)`

Returns the hostname of the machine that this process is running on.

system-information [procedure]

`(system-information)`

Invokes the UNIX system call `uname()` and returns 5 values: system-name, node-name, OS release, OS version and machine.

5.18.19 Setting a files buffering mode

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).

5.18.20 Terminal ports

terminal-name [procedure]

`(terminal-name PORT)`

Returns the name of the terminal that is connected to `PORT`.

terminal-port? [procedure]

`(terminal-port? PORT)`

Returns `#t` if `PORT` is connected to a terminal and `#f` otherwise.

5.18.21 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-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
`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

5.18.22 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
signal/...
set-signal-handler!  set-signal-mask!
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-groupd-id
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-mapped-file?
set-alarm!
terminal-port?  terminal-name
process-fork  process-signal
parent-process-id
set-root-directory!
```

Additionally, the following definitions are only available for Windows:

spawn/overlay [spawn mode]
spawn/wait [spawn mode]
spawn/nowait [spawn mode]
spawn/nowaito [spawn mode]
spawn/detach [spawn mode]

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.

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.

5.19 Unit utils

This unit contains some utility procedures for Shell scripting and for some file operations.

This unit uses the `extras` and `regex` units.

5.19.1 Pathname operations

absolute-pathname? [procedure]

```
(absolute-pathname? PATHNAME)
```

Returns `#t` if the string `PATHNAME` names an absolute pathname, and returns `#f` otherwise.

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.

make-pathname [procedure]
make-absolute-pathname [procedure]

```
(make-pathname DIRECTORY FILENAME [EXTENSION])
(make-absolute-pathname DIRECTORY FILENAME [EXTENSION])
```

Returns a string that names the file with the components `DIRECTORY, FILENAME` and (optionally) `EXTENSION`. `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.

pathname-directory [procedure]

```
(pathname-directory PATHNAME)
```

pathname-file [procedure]

```
(pathname-file PATHNAME)
```

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.

pathname-replace-directory [procedure]

```
(pathname-replace-directory PATHNAME DIRECTORY)
```

pathname-replace-file [procedure]

```
(pathname-replace-file PATHNAME FILENAME)
```

pathname-replace-extension [procedure]

```
(pathname-replace-extension PATHNAME EXTENSION)
```

Return a new pathname with the specified component of `PATHNAME` replaced by a new value.

pathname-strip-directory [procedure]

```
(pathname-strip-directory PATHNAME)
```

pathname-strip-extension [procedure]

```
(pathname-strip-extension PATHNAME)
```

Return a new pathname with the specified component of `PATHNAME` stripped.

5.19.2 Temporary files

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.

5.19.3 Deleting a file without signalling an error

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.

5.19.4 Iterating over input lines and files

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.

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)`.

This code will act as a simple Unix cat(1) command:

```
(for-each-argv-line print)
```

port-for-each [procedure]
port-map [procedure]

```
(port-for-each FN THUNK)
(port-map FN THUNK)
```

Apply `FN` to successive results of calling the zero argument procedure `THUNK` until it returns `#!eof`. `port-for-each` discards the results, while `port-map` returns a list of the collected results.

5.19.5 Executing shell commands with formatstring and error checking

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.

5.19.6 Reading a file's contents

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)`.

5.19.7 Miscellaneous handy things

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)
```

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)
```

5.20 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)`.

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.

tcp-listener? [procedure]

```
(tcp-listener? X)
```

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

tcp-close [procedure]

```
(tcp-close LISTENER)
```

Reclaims any resources associated with `LISTENER`.

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.

tcp-accept-ready? [procedure]

```
(tcp-accept-ready? LISTENER)
```

Returns `#t` if there are any connections pending on `LISTENER`, or `#f` otherwise.

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).

tcp-listener-fileno [procedure]

```
(tcp-listener-port LISTENER)
```

Returns the file-descriptor associated with `LISTENER`.

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.

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.

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.

A very simple example follows. Say we have the two files `client.scm` and `server.scm`:

```
; client.scm
(define-values (i o) (tcp-connect "localhost" 4242))
(write-line "Good Bye!" o)
(print (read-line i))
```

```
; server.scm
(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)

% csi -script server.scm &
[1] 1409
% csi -script client.scm
Good Bye!
Hello!
```

5.21 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.

5.21.1 Foreign pointers

address->pointer [procedure]

```
(address->pointer ADDRESS)
```

Creates a new foreign pointer object initialized to point to the address given in the integer `ADDRESS`.

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))
```

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))
```

null-pointer [procedure]

```
(null-pointer)
```

Another way to say `(address->pointer 0)`.

null-pointer? [procedure]

```
(null-pointer? PTR)
```

Returns `#t` if `PTR` contains a `NULL` pointer, or `#f` otherwise.

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.

pointer? [procedure]

```
(pointer? X)
```

Returns `#t` if `X` is a foreign pointer object, and `#f` otherwise.

pointer=? [procedure]

```
(pointer=? PTR1 PTR2)
```

Returns `#t` if the pointer-like objects `PTR1` and `PTR2` point to the same address.

pointer->address [procedure]

```
(pointer->address PTR)
```

Returns the address, to which the pointer `PTR` points.

pointer->object [procedure]

```
(pointer->object PTR)
```

Returns the Scheme object pointed to by the pointer `PTR`.

pointer-offset [procedure]

```
(pointer-offset PTR N)
```

Returns a new pointer representing the pointer `PTR` increased by `N`.

pointer-u8-ref [procedure]

```
(pointer-u8-ref PTR)
```

Returns the unsigned byte at the address designated by `PTR`.

pointer-s8-ref [procedure]

```
(pointer-s8-ref PTR)
```

Returns the signed byte at the address designated by `PTR`.

pointer-u16-ref [procedure]

```
(pointer-u16-ref PTR)
```

Returns the unsigned 16-bit integer at the address designated by `PTR`.

pointer-s16-ref [procedure]

```
(pointer-s16-ref PTR)
```

Returns the signed 16-bit integer at the address designated by `PTR`.

pointer-u32-ref [procedure]

```
(pointer-u32-ref PTR)
```

Returns the unsigned 32-bit integer at the address designated by `PTR`.

pointer-s32-ref [procedure]

```
(pointer-s32-ref PTR)
```

Returns the signed 32-bit integer at the address designated by `PTR`.

pointer-f32-ref [procedure]

```
(pointer-f32-ref PTR)
```

Returns the 32-bit float at the address designated by `PTR`.

pointer-f64-ref [procedure]

```
(pointer-f64-ref PTR)
```

Returns the 64-bit double at the address designated by `PTR`.

pointer-u8-set! [procedure]

```
(pointer-u8-set! PTR N)
```

Stores the unsigned byte `N` at the address designated by `PTR`.

pointer-s8-set! [procedure]

```
(pointer-s8-set! PTR N)
```

Stores the signed byte `N` at the address designated by `PTR`.

pointer-u16-set! [procedure]

```
(pointer-u16-set! PTR N)
```

Stores the unsigned 16-bit integer `N` at the address designated by `PTR`.

pointer-s16-set! [procedure]

```
(pointer-s16-set! PTR N)
```

Stores the signed 16-bit integer `N` at the address designated by `PTR`.

pointer-u32-set! [procedure]

```
(pointer-u32-set! PTR N)
```

Stores the unsigned 32-bit integer `N` at the address designated by `PTR`.

pointer-s32-set! [procedure]

```
(pointer-s32-set! PTR N)
```

Stores the 32-bit integer `N` at the address designated by `PTR`.

pointer-f32-set! [procedure]

```
(pointer-f32-set! PTR N)
```

Stores the 32-bit floating-point number `N` at the address designated by `PTR`.

pointer-f64-set! [procedure]

```
(pointer-f64-set! PTR N)
```

Stores the 64-bit floating-point number `N` at the address designated by `PTR`.

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.

5.21.2 Tagged pointers

"Tagged" pointers are foreign pointer objects with an extra tag object.

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.

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.

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.

5.21.3 Extending procedures with data

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.

extended-procedure? [procedure]

    (extended-procedure? PROCEDURE)

Returns `#t` if `PROCEDURE` is an extended procedure, or `#f` otherwise.

procedure-data [procedure]

    (procedure-data PROCEDURE)

Returns the data object contained in the extended procedure `PROCEDURE`.

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
```

5.21.4 Bytevectors

byte-vector [procedure]

    (byte-vector FIXNUM ...)

Returns a freshly allocated byte-vector with `FIXNUM ...` as its initial contents.

byte-vector? [procedure]

    (byte-vector? X)

Returns `#t` if `X` is a byte-vector object, or `#f` otherwise.

byte-vector-fill! [procedure]

    (byte-vector-fill! BYTE-VECTOR N)

Sets each element of `BYTE-VECTOR` to `N`, which should be an exact integer.

byte-vector->list [procedure]

    (byte-vector->list BYTE-VECTOR)

Returns a list with elements taken from `BYTE-VECTOR`.

byte-vector->string [procedure]

```
(byte-vector->string BYTE-VECTOR)
```

Returns a string with the contents of `BYTE-VECTOR`.

byte-vector-length [procedure]

```
(byte-vector-length BYTE-VECTOR)
```

Returns the number of elements in `BYTE-VECTOR`.

byte-vector-ref [procedure]

```
(byte-vector-ref BYTE-VECTOR INDEX)
```

Returns the byte at the `INDEX`th position of `BYTE-VECTOR`.

byte-vector-set! [procedure]

```
(byte-vector-set! BYTE-VECTOR INDEX N)
```

Sets the byte at the `INDEX`th position of `BYTE-VECTOR` to the value of the exact integer `n`.

executable-byte-vector->procedure [procedure]

```
(executable-byte-vector->procedure PBYTE-VECTOR)
```

Returns a procedure that on invocation will execute the code in `PBYTE-VECTOR`, which should have been allocated using `make-executable-byte-vector`. The procedure follows the native C calling convention, and will be called as if declared with the following prototype:

```
void <procedure>(int argc, C_word closure, C_word k, C_word arg1, ...)
```

- `argc` contains the number of arguments `arg1, ...` that are passed plus 2 (including `closure` and `k`).
- `closure` is the procedure object itself.
- `k` is a continuation closure that should be called when the code is about to return.

  ```
  typedef void (*CONTINUATION)(int argc, C_word closure,
                                         C_word result);
  ((CONTINUATION)k[ 1 ])(2, k, result)
  ```

  (`k` is a data object with the second word being the actual code pointer)

An example:

```
(define x (make-executable-byte-vector 17))
(move-memory!
 '#u8(#x8b #x44 #x24 #x0c  ; movl 12(%esp), %eax - 'k'
      #x8b #x5c #x24 #x10  ; movl 16(%esp), %ebx - 'arg1'
      #x53                 ; pushl %ebx          - push result
      #x50                 ; pushl %eax          - push k
      #x6a #x02            ; pushl $2            - push argument count
      #x8b #x40 #x04       ; movl 4(%eax), %eax  - fetch code pointer
      #xff #xd0)           ; call *%eax
 x)
(define y (executable-byte-vector->procedure x))
(y 123)                                         ==> 123
```

The result of calling `executable-byte-vector->procedure` with a non-executable statically allocated byte-vector is undefined.

invoke-executable-byte-vector [procedure]

```
(invoke-executable-byte-vector PBYTE-VECTOR ARG1 ...)
```

Invokes the machine code stored in the executable byte-vector `PBYTE-VECTOR`. The native C calling conventions are used, but the invoked code is passed a single argument containing a pointer to an array of the Scheme objects `ARG1 ...`.

```
(define v (make-executable-byte-vector 7))
(move-memory!
 '#u8(#x8b #x44 #x24 #x04        ; movl 4(%esp), %eax
      #x8b #x00                  ; movl 0(%eax), %eax
      #xc3)                      ; ret
 v)
(invoke-executable-byte-vector v "hello!") ==> "hello!"
```

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.

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.

make-executable-byte-vector [procedure]

```
(make-executable-byte-vector SIZE [INIT])
```

As `make-static-byte-vector`, but the code is suitable for execution. **Note:** this feature is currently only available on **x86** platforms.

Exceptions: `(exn bounds)`, `(exn runtime)`

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)`

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`.

string->byte-vector [procedure]

```
(string->byte-vector STRING)
```

Returns a byte-vector with the contents of `STRING`.

5.21.5 Data in unmanaged memory

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`.

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. Two values are returned: the evicted object and a new pointer pointing to the first free address after the evicted object.

object-evicted? [procedure]

```
(object-evicted? X)
```

Returns `#t` if `X` is a non-immediate evicted data object, or `#f` otherwise.

object-size [procedure]

```
(object-size X)
```

Returns the number of bytes that would be needed to evict the data object `X`.

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`.

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`.

5.21.6 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.

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, symbol, pair, string, byte-vector, SRFI-4 number-vector, or record. `INDEX` should be a fixnum. `INDEX` defaults to 0.

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.

locative? [procedure]

`(locative? X)`

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

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.

locative-set! [procedure]

`(locative-set! 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.

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.

5.21.7 Accessing toplevel variables

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.

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.

global-set! [procedure]

```
(global-set! SYMBOL X)
```

Sets the global variable named `SYMBOL` to the value `X`.

5.21.8 Low-level data access

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.

block-set! [procedure]

```
(block-set! 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.

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.

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)) )
```

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: Section 5.11 [Unit srfi-4], page 66), 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.

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.

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.

record-instance? [procedure]

`(record-instance? X)`

Returns `#t` if X is an instance of a record type. See also: `make-record-instance`.

record->vector [procedure]

`(record->vector BLOCK)`

Returns a new vector with the type and the elements of the record BLOCK.

5.21.9 Procedure-call- and variable reference hooks

invalid-procedure-call-handler [procedure]

`(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:

(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.

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.

5.21.10 Magic

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.

5.22 Unit tinyclos

This unit is a port of Gregor Kiczales **TinyCLOS** with numerous modifications.

This unit uses the `extras` unit.

5.22.1 Defining forms

define-class [syntax]

```
(define-class NAME (SUPERCLASS1 ...) (SLOTNAME1 ...) [METACLASS])
```

Sets the variable `NAME` to a new class (a new instance of the class `<class>`). `SUPERCLASS1 ...` is a list of superclasses of the newly created class. If no superclasses are given, then `<object>` is assumed. `SLOTNAME1 ...` are the names of the direct slots of the class. if `METACLASS` is provided, then the new class-instance is an instance of `METACLASS` instead of `<class>`.

```
(define-class NAME (SUPER) (SLOT1 SLOT2) META)
```

is equivalent to

```
(define NAME
  (make META
    'name 'NAME
    'direct-supers (list SUPER)
    'direct-slots (list 'SLOT1 'SLOT2)) )
```

Note that slots-names are not required to be symbols, so the following is perfectly valid:

```
(define hidden-slot (list 'hidden))
(define <myclass>
  (make <class>
     'direct-supers (list <object>)
     'direct-slots (list hidden-slot) ) )
(define x1 (make <myclass>)
(slot-set! x1 hidden-slot 99)
```

define-generic [syntax]

```
(define-generic NAME [CLASS])
```

Sets the variable `NAME` to contain a fresh generic function object without associated methods. If the optional argument `CLASS` is given, then the generic function will be an instance of that class.

define-method [syntax]

```
(define-method (NAME (VARIABLE1 CLASS1) ... PARAMETERS ...) BODY ...)
```

Adds a new method with the code `BODY ...` to the generic function that was assigned to the variable `name`. `CLASS1 ...` is a list if classes that specialize this particular method. The method can have additional parameters `PARAMETERS`, which do not specialize the method any further. Extended lambda-lists are allowed (`#!optional`, `#!key` or `#!rest` argument lists), but can not be specialized. Inside the body of the method the identifier `call-next-method` names a procedure of zero arguments that can be invoked to call the next applicable method with the same arguments. If no generic function is defined under this name, then a fresh generic function object is created and assigned to `NAME`. Note that only `define-generic` expands into a valid definition, so for internal lexically scoped definitions use `define-generic`.

5.22.2 Base language

add-method [procedure]

```
(add-method GENERIC METHOD)
```

Adds the method object `METHOD` to the list of applicable methods for the generic function `GENERIC`.

instance? [procedure]

```
(instance? X)
```

Returns `#t` if `X` is an instance of a non-primitive class.

make [procedure]

```
(make CLASS INITARG ...)
```

Creates a new instance of `CLASS` and passes `INITARG ...` to the `initialize` method of this class.

make-class [procedure]

```
(make-class SUPERCLASSES SLOTNAMES)
```

Creates a new class object, where `SUPERCLASSES` should be the list of direct superclass objects and `SLOTNAMES` should be a list of symbols naming the slots of this class.

make-generic [procedure]

```
(make-generic [NAME])
```

Creates a new generic function object. If `NAME` is specified, then it should be a string.

make-method [procedure]

```
(make-method SPECIALIZERS PROC)
```

Creates a new method object specialized to the list of classes in `SPECIALIZERS`.

```
(define-method (foo (x <bar>)) 123)
   <=> (add-method foo
                   (make-method
                      (list <bar>)
                      (lambda (call-next-method x) 123)))
```

slot-ref [procedure]

```
(slot-ref INSTANCE SLOTNAME)
```

Returns the value of the slot `SLOTNAME` of the object `INSTANCE`.

slot-set! [procedure]

```
(slot-set! INSTANCE SLOTNAME VALUE)
```

Sets the value of the slot `SLOTNAME` of the object `INSTANCE` to `VALUE`.

5.22.3 Introspection

class-cpl [procedure]

```
(class-cpl CLASS)
```

Returns the class-precedence-list of `CLASS` as a list of classes.

class-direct-slots [procedure]

```
(class-direct-slots CLASS)
```

Returns the list of direct slots of `CLASS` as a list of lists, where each sublist contains the name of the slot.

class-direct-supers [procedure]

```
(class-direct-supers CLASS)
```

Returns the list of direct superclasses of `CLASS`.

class-of [procedure]

```
(class-of X)
```

Returns the class that the object `X` is an instance of.

class-name [procedure]

```
(class-name CLASS)
```

Returns name of `CLASS`.

class-slots [procedure]

`(class-slots CLASS)`

Returns the list of all slots of `CLASS` and its superclasses as a list of lists, where each sublist contains the name of the slot.

generic-methods [procedure]

`(generic-methods GENERIC)`

Returns the list of all methods associated with the generic function `GENERIC`.

method-specializers [procedure]

`(method-specializers METHOD)`

Returns the list of classes that specialize `METHOD`.

method-procedure [procedure]

`(method-procedure METHOD)`

Returns the procedure that contains the body of `METHOD`.

subclass? [procedure]

`(subclass? CLASS1 CLASS2)`

Returns `#t` is `CLASS1` is a subclass of `CLASS2`, or `#f` otherwise. Note that the following holds:

```
(subclass? X X) ==> #t
```

instance-of? [procedure]

`(instance-of? X CLASS)`

Returns `#t` if `X` is an instance of `CLASS` (or one of its subclasses).

5.22.4 Intercessory protocol

These definitions allow interfacing to the Meta Object Protocol of TinyCLOS. For serious use, it is recommended to consult the source code (`tinyclos.scm`).

allocate-instance [generic]

`(allocate-instance CLASS)`

Allocates storage for an instance of `CLASS` and returns the instance.

compute-apply-generic [generic]

`(compute-apply-generic GENERIC)`

Returns a procedure that will be called to apply the generic function methods to the arguments.

compute-apply-methods [generic]

`(compute-apply-methods GENERIC)`

Returns a procedure of two arguments, a list of applicable methods and a list of arguments and applies the methods.

compute-methods [generic]

`(compute-methods GENERIC)`

Returns a procedure of one argument. The procedure is called with the list of actual arguments passed to the generic function and should return a list of applicable methods, sorted by precedence.

compute-cpl [generic]

`(compute-cpl CLASS)`

Computes and returns the class-precedence-list of `CLASS`.

compute-getter-and-setter [generic]

`(compute-getter-and-setter CLASS SLOT ALLOCATOR)`

Returns two values, the procedures that get and set the contents of the slot `SLOT`. `ALLOCATOR` is a procedure of one argument that gets an initalizer function and returns the getter and setter procedures for the allocated slot.

compute-method-more-specific? [generic]

`(compute-method-more-specific? GENERIC)`

Returns a procedure of three arguments (two methods and a list of arguments) that returns `#t` if the first method is more specific than the second one with respect to the list of arguments. Otherwise the returned predicate returns `#f`.

compute-slots [generic]

`(compute-slots CLASS)`

Computes and returns the list of slots of `CLASS`.

initialize [generic]

`(initialize INSTANCE INITARGS)`

Initializes the object `INSTANCE`. `INITARGS` is the list of initialization arguments that were passed to the `make` procedure.

5.22.5 Additional protocol

describe-object [generic]

`(describe-object INSTANCE [PORT])`

Writes a description of `INSTANCE` to `PORT`. Execution of the interpreter command `,d` will invoke this generic function. If `PORT` is not given it defaults to the value of `(current-output-port)`.

print-object [generic]

`(print-object INSTANCE [PORT])`

Writes a textual representation of `INSTANCE` to `PORT`. Any output of an instance with `display`, `write` and `print` will invoke this generic function. If `PORT` is not given it defaults to the value of `(current-output-port)`.

5.22.6 Utility procedures

initialize-slots [procedure]

```
(initialize-slots INSTANCE INITARGS)
```

This procedure takes a sequence of alternating slot-names and initialization values in `INITARGS` and initializes the corresponding slots in `INSTANCE`.

```
(define-class <pos> () (x y))

(define-method (initialize (pos <pos>) initargs)
  (call-next-method)
  (initialize-slots pos initargs))

(define p1 (make <pos> 'x 1 'y 2))
(define p2 (make <pos> 'x 3 'y 5))
```

5.22.7 Builtin classes

The class hierarchy of builtin classes looks like this:

```
<top>
  <object>
    <class>
      <procedure-class>
        <procedure>
        <entity-class>
          <generic>
      <primitive-class>
    <c++-object>
  <primitive>
    <void>
    <boolean>
    <symbol>
    <char>
    <vector>
    <pair>
    <number>
      <integer>
        <exact>
      <inexact>
    <string>
    <port>
      <input-port>
      <output-port>
    <pointer>
      <tagged-pointer>
      <swig-pointer>
    <locative>
    <byte-vector>
      <u8vector>
      <s8vector>
      <u16vector>
      <s16vector>
      <u32vector>
      <s32vector>
      <f32vector>
      <f64vector>
    <structure>
      <array>
      <char-set>
      <condition>
      <environment>
      <hash-table>
      <lock>
      <mmap>
      <promise>
      <queue>
      <tcp-listener>
      <time>
    <end-of-file>
```

<primitive> → <top> [class]

The parent class of the classes of all primitive Scheme objects.

<boolean> → <primitive> [class]
<symbol> → <primitive> [class]
<char> → <primitive> [class]
<vector> → <primitive> [class]
<null> → <primitive> [class]
<pair> → <primitive> [class]
<number> → <primitive> [class]
<integer> → <primitive> [class]
<exact> → <integer> [class]
<inexact> → <number> [class]
<string> → <primitive> [class]
<port> → <primitive> [class]
<environment> → <structure> [class]
<end-of-file> → <primitive> [class]
<input-port> → <port> [class]
<output-port> → <port> [class]
<procedure> → <procedure-class> [class]

The classes of primitive Scheme objects.

<byte-vector> → <primitive> [class]
<structure> → <primitive> [class]
<hash-table> → <structure> [class]
<queue> → <structure> [class]

The classes of extended data types provided by the various library units.

<class> → <object> [class]

The parent class of all class objects.

<entity-class> → <class> [class]

The parent class of objects that can be invoked as a procedure and have slots.

<generic> → <entity-class> [class]

The parent class of generic function objects.

<method> → <class> [class]

The parent class of method objects.

<object> → <class> [class]

The parent class of all objects.

<procedure-class> → <class> [class]

The parent class of objects that can be invoked as a procedure.

<condition> → <structure> [class]

Class of condition objects.

<array> → <structure> [class]
<char-set> → <structure> [class]
<time> → <structure> [class]
<u8vector> → <byte-vector> [class]
<s8vector> → <byte-vector> [class]
<u16vector> → <byte-vector> [class]
<s16vector> → <byte-vector> [class]
<u32vector> → <byte-vector> [class]
<s32vector> → <byte-vector> [class]
<f32vector> → <byte-vector> [class]
<f64vector> → <byte-vector> [class]

The classes of data objects provided by the various supported SRFIs.

<lock> → <structure> [class]
<mmap> → <structure> [class]

Classes of objects used in the `posix` library unit.

<pointer> → <primitive> [class]
<tagged-pointer> → <pointer> [class]
<swig-pointer> → <pointer> [class]

A machine pointer (untagged, tagged or pointing to SWIG-wrapped data).

<locative> → <primitive> [class]

A locative.

<promise> → `<structure>` [class]

The class of objects returned by `delay`.

<tcp-listener> → `<structure>` [class]

The class of an object returned by `tcp-listen`.

<c++-class> → `<object>` [class]

The class of generated wrappers for C++ classes parsed by the "easy" Foreign Function interface.

The CHICKEN distribution provides several examples in the file `tinyclos-examples.scm`.

6 Interface to external functions and variables

6.1 Accessing external objects

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>
```

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))
```

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.

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-foreign-enum . items)
  `(begin
     ,@(map (match-lambda
              [(name realname) `(define-foreign-variable ,name int ,realname)]
              [name `(define-foreign-variable ,name int)] )
     items) ) )

(define-foreign-enum abc def ghi)

ghi                                ==> 5
```

define-foreign-record [syntax]

`(define-foreign-record NAME 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. 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:

TYPENAME-SLOTNAME [procedure]

`(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.

TYPENAME-SLOTNAME-set! [procedure]

`(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.

Note that no constructor or release procedure will be generated.

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.

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;
     return(n);") )

(my-strlen "one two three")             ==> 13
```

For obscure technical reasons any use of the return statement should enclose the result value in parentheses. For the same reasons return without an argument is not allowed.

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 Section 6.4 [Callbacks], page 131.

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 Section 6.4 [Callbacks], page 131.

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.

6.2 Foreign type specifiers

Here is a list of valid foreign type specifiers:

`scheme-object`
: An arbitrary Scheme data object (immediate or non-immediate).

`bool`
: As argument: any value (`#f` is false, anything else is true). As result: anything different from 0 and the `NULL`-pointer is `#t`.

`byte unsigned-byte`
: A byte.

`char unsigned-char`
: A character.

`short unsigned-short`
: A short integer number.

`int unsigned-int`
: An small integer number in fixnum range (at least 30 bit).

`integer unsigned-integer`
: Either a fixnum or a flonum in the range of a (unsigned) machine “int”.

`long unsigned-long`
: Either a fixnum or a flonum in the range of a (unsigned) machine “long”.

`float double`
: A floating-point number. If an exact integer is passed as an argument, then it is automatically converted to a float.

`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 int or not.

`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 is treated like a string and a symbol with the same name will be interned in the symbol table.

`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).

`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).

`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`.

`nonnull-c-pointer`
: As `c-pointer`, but guaranteed not to be `#f/NULL`.

`[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.

`[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.

`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.

`nonnull-c-string`
: As `c-string`, but guaranteed not to be `#f/NULL`.

`[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.

`void`
: Specifies an undefined return value. Not allowed as argument type.

`(const TYPE)`
: The foreign type `TYPE` with an additional `const` specifier.

`(enum NAME)`
: An enumeration type. Handled internally as an `integer`.

`(pointer TYPE)`
`(c-pointer TYPE)`
: An operating-system pointer or a locative to an object of `TYPE`.

`(nonnull-pointer TYPE)`
`(nonnull-c-pointer TYPE)`
: As `(pointer TYPE)`, but guaranteed not to be `#f/NULL`.

`(ref TYPE)`
: A C++ reference type. Reference types are handled the same way as pointers inside Scheme code.

`(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.

`(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.

`(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.

`(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>`.

`(instance-ref CNAME SCHEMECLASS)`
: A reference to a C++ class instance.

`(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.

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 &

6.3 Entry points

To simplify embedding compiled Scheme code into arbitrary programs, one can define so called "entry points", which provide a uniform interface and parameter conversion facilities. To use this facility, add

```
(include "chicken-entry-points")
```

to the beginning of your code.

define-entry-point [syntax]

```
(define-entry-point INDEX ((VAR1 TYPE1) ...)
                          (RTYPE1 ...)
                          EXP1 EXP2 ...)
```

Defines a new entry-point with index `INDEX` which should evaluate to an exact integer. During execution of the body `EXP1 EXP2 ...` the variables `VAR1 ...` are bound to the parameters passed from the host program to the invoked entry point. The parameters passed are converted according to the foreign type specifiers `TYPE1 ...`. The expressions should return as many values as foreign type specifiers are given in `RTYPE1 ...`, with the exception that if the list of return types is empty, a single value is expected to be returned from the body (there is no need to add a `(values)` form at the end). The results are then transformed into values that can be used in the host program.

Note: if one or more of the result types `RTYPE ...` specify the type `c-string`, then the parameter types at the same positions in `TYPE1 ...` have to be `c-string`s as well, because the result strings are copied into the same area in memory. You should also take care that the passed buffer is long enough to hold the result string or unpredictable things will happen.

If entry points were defined then the program will not terminate after execution of the last toplevel expression, but instead it will enter a loop that waits for the host to invoke one of the defined entry points.

define-embedded [syntax]

```
(define-embedded [QUALIFIER ... ] ([CCONV] NAME (TYPE1 VAR1) ...) RTYPE BODY ...)
```

Defines a named entry-point that can be accessed from external code with a normal C/C++ function call. `QUALIFIER` may be a string for special function declarations (like `__declspec(dllexport)` on Windows, for example) and `CCONV` may be a string designating a calling convention (like `__cdecl`). During the execution of the `BODY` the variables `VAR1 ...` are bound to the arguments passed to the function, which should be of types compatible to the type specifiers `TYPE1 ...`. `RTYPE` specifies the result type of the entry-point. The return type specifiers `[nonnull-]c-string` and `[nonnull-]c-string*` are handled specially: if the return type is `c-string`, a heap-allocated pointer to a zero-terminated string will be returned, which will be valid until the next invocation of an entry-point into the Scheme code. If the return type is `c-string*`, a buffer allocated with `malloc` will be returned, and freeing the string is the responsibility of the caller.

All entry-point definitions with `define-embedded` should be put into the same source file. `define-embedded` expands into a `define-entry-point` form and the entry-point index is provided by the macro (starting from 1). If `define-embedded` is used in multiple source files then the index is counted from 1 in each of the files, resulting in multiple entry-points with the same index.

The following C functions and data types are provided:

`void` **CHICKEN_parse_command_line** (`int argc, char *argv[], int *heap, int *stack int *symbols`) [C function]

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.

`int` **CHICKEN_initialize** (`int heap, int stack, int symbols, void *toplevel`) [C function]

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` or `CHICKEN_invoke` 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`.

`void` **CHICKEN_run** (`void **data, int *bytes, int *maxlen, void *toplevel`) [C function]

Starts the Scheme program. `data, bytes` and `maxlen` contain invocation parameters in raw form. Pass `NULL` here. 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.

void **CHICKEN_invoke** (int index, C_parameter *params, int count, void *toplevel) [C function]

Invoke the entry point with index `index`. `count` should contain the maximum number of arguments or results (whatever is higher). `params` is a pointer to parameter data:

```
typedef union
{
  C_word x;           /* parameter type scheme-object */
  long i;             /* parameter type bool, [unsigned] int/short/long */
  long c;             /* parameter type [unsigned] char */
  double f;           /* parameter type float/double */
  void *p;            /* any pointer parameter type and C strings */
} C_parameter;
```

This function calls `CHICKEN_run` if it was not called at least once before.

int **CHICKEN_is_running** () [C function]

Returns `1`, if called inside a dynamic context invoked via `CHICKEN_run` or `CHICKEN_invoke` (i.e. if running inside a call from Scheme to C). If no Scheme stack frame is currently active, then this function returns `0`.

Here is a simple example (assuming a UNIX-like environment):

```
% cat foo.c
#include <stdio.h>
#include "chicken.h"

int main(void)
{
  C_parameter p[ 3 ];
  char str[ 32 ] = "hello!";  /* We need some space for the result string! */

  memset(p, 0, sizeof(p));
  p[ 0 ].i = -99;
  p[ 1 ].p = str;
  p[ 2 ].f = 3.14;
  CHICKEN_invoke(1, p, 3, C_toplevel);
  printf("->\n%d\n%s\n", p[ 0 ].i, p[ 1 ].p);
  return 0;
}

% cat bar.scm
(include "chicken-entry-points")

(define-entry-point 1
    ((a integer) (b c-string) (c double))
    (int c-string)
  (print (list a b c))
  (values 123 "good bye!") )
```

```
% chicken bar.scm -quiet
% gcc foo.c bar.c -o foo ‘chicken-config -cflags -libs -embedded‘
% foo
(-99 "hello!" 3.14)
->
123
good bye!
```

Note the use of `-embedded`. We have to compile with additional compiler options, because the host program provides the `main` function.

Here another example that uses named entry-points defined with `define-embedded`:

```
% cat foo.c
extern int foo(int, char *);
extern unsigned int square(double);

int main() { foo(square(9), "yo!"); return 0; }

% cat bar.scm
(include "chicken-entry-points")

(define-embedded (foo (int x) (c-string y)) int
  (print x ": " y)
  x)

(define-embedded (square (double x)) unsigned-int
  (* x x))

% chicken bar.scm
compiling ‘bar.scm’ ...
% gcc foo.c bar.c ‘chicken-config -cflags -libs -embedded‘
% a.out
81: yo!
```

CHICKEN also provides “boilerplate” entry points, that simplify invoking Scheme code embedded in a C or C++ application a lot. The include file `default-entry-points.scm` will define entry-points for common usage patterns, like loading a file, evaluating an expression or calling a procedure.

`void` **CHICKEN_eval** (`C_word exp, C_word *result, int *status`) [C macro]

Evaluates the Scheme object passed in `exp`, writing the result value to `result`. `status` is set to 1 if the operation succeeded, or 0 if an error occurred. Call `CHICKEN_get_error_message` to obtain a description of the error.

`void` **CHICKEN_eval_string** (`char *str, C_word *result, int *status`) [C macro]

Evaluates the Scheme expression passed in the string `str`, writing the result value to `result`.

`void` **CHICKEN_eval_to_string** `(C_word exp, char *result, int size, int *status)` [C macro]

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.

`void` **CHICKEN_eval_string_to_string** `(char *str, char *result, int size, int *status)` [C macro]

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.

`void` **CHICKEN_apply** `(C_word func, C_word args, C_word *result, int *status)` [C macro]

Applies the procedure passed in `func` to the list of arguments `args`, writing the result value to `result`.

`void` **CHICKEN_apply_to_string** `(C_word func, C_word args, char *result, int size, int *status)` [C macro]

Applies the procedure passed in `func` to the list of arguments `args`, writing a textual representation of the result into `result`.

`void` **CHICKEN_read** `(char *str, C_word *result, int *status)` [C macro]

Reads a Scheme object from the string `str`, writing the result value to `result`.

`void` **CHICKEN_load** `(char *filename, int *status)` [C macro]

Loads the Scheme file `filename` (either in source form or compiled).

`void` **CHICKEN_get_error_message** `(char *result, int size)` [C macro]

Returns a textual description of the most recent error that occurred in executing embedded Scheme code.

`void` **CHICKEN_yield** `(int *status)` [C macro]

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.

An example:

```
% cat x.scm
;;; x.scm

(include "chicken-default-entry-points")
(define (bar x) (gc) (* x x))

% cat y.c
/* y.c */

#include "chicken.h"
#include <assert.h>
```

```
int main() {
  char buffer[ 256 ];
  int status;
  C_word val = C_SCHEME_UNDEFINED;
  C_word *data[ 1 ];

  data[ 0 ] = &val;

  CHICKEN_read("(bar 99)", &val, &status);
  assert(status);

  C_gc_protect(data, 1);

  printf("data: %08x\n", val);

  CHICKEN_eval_string_to_string("(bar)", buffer, 255, &status);
  assert(!status);

  CHICKEN_get_error_message(buffer, 255);
  printf("ouch: %s\n", buffer);

  CHICKEN_eval_string_to_string("(bar 23)", buffer, 255, &status);
  assert(status);

  printf("-> %s\n", buffer);
  printf("data: %08x\n", val);

  CHICKEN_eval_to_string(val, buffer, 255, &status);
  assert(status);
  printf("-> %s\n", buffer);

  return 0;
}
```

```
% csc x.scm y.c -embedded
```

A simpler interface For handling GC-safe references to Scheme data are the so called “gc-roots”:

`void*` **CHICKEN_new_gc_root** () [C function]

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.

`void` **CHICKEN_delete_gc_root** (`void *root`) [C function]

Deletes the gc root.

`C_word` **CHICKEN_gc_root_ref** (`void *root`) [C macro]

Returns the value stored in the gc root.

`void` **CHICKEN_gc_root_set** (`void *root, C_word value`) [C macro]

Sets the content of the GC root to a new value.

Sometimes it is handy to access global variables from C code:

`void*` **CHICKEN_global_lookup** (`char *name`) [C function]

Returns a GC root that holds the global variable with the name `name`. If no such variable exists, `NULL` is returned.

`C_word` **CHICKEN_global_ref** (`void *global`) [C function]

Returns the value of the global variable referenced by the GC root `global`.

`void` **CHICKEN_global_set** (`void *global, C_word value`) [C function]

Sets the value of the global variable referenced by the GC root `global` to `value`.

6.4 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` or `foreign-safe-wrapper` forms.

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 (either defined by `define-external` or `foreign-safe-wrapper` 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).

define-external [syntax]

```
(define-external [QUALIFIERS] (NAME (ARGUMENTTYPE1 VARIABLE1) ...) RETURNTYPE BOD
(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))
```

is equivalent to

```
(define foo
  (foreign-safe-wrapper int "foo"
    (c-string) (lambda (x) (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 ()
  "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`).

foreign-safe-wrapper [syntax]

```
(foreign-safe-wrapper RETURNTYPE NAME [QUALIFIERS] (ARGUMENTTYPE1 ...) EXP)
```

Defines an externally callable wrapper around the procedure `EXP`. `EXP` **must** be a lambda expression of the form `(lambda ...)`. The wrapper will have the name `NAME` and will have a signature as specified in the return- and argument-types given in `RETURNTYPE` and `ARGUMENTTYPE1 ...`. `QUALIFIERS` is a qualifier string for the function definition (see `define-external`).

`C_word` **C_callback** (`C_word closure, int argc`) [C function]

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.

6.5 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.

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`).

let-location [syntax]

```
(let-location ((NAME TYPE [INIT]) ...) BODY ...)
```

Defines a lexically bound location.

location [syntax]

```
(location NAME)
(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`.

6.6 Other support procedures

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.

6.7 The *Easy* Foreign Function Interface

The compiler contains a builtin parser for a restricted subset of C and C++ that allows the easy generation of foreign variable declarations, procedure bindings and C++ class wrappers. The parser is invoked via the declaration-specifier `foreign-parse`, which extracts binding information and generates the necessary code. An example:

```
(declare
  (foreign-declare "
#include <math.h>

#define my_pi 3.14
")
```

```
  (foreign-parse "extern double sin(double);") )

(print (sin 3.14))
```

The parser would generate code that is equivalent to

```
(declare
  (foreign-declare "
#include <math.h>

#define my_pi 3.14
")

(define-foreign-variable my_pi float "my_pi")
(define sin (foreign-lambda double "sin" double))
```

Note that the read syntax `#>[SPEC] ... <#` provides a somewhat simpler way of using the parser. The example above could alternatively be expressed as

```
#>!
#define my_pi 3.14

extern double sin(double);
<#

(print (sin 3.14))
```

Another example, here using C++. Consider the following class:

```
// file: foo.h

class Foo {
 private:
  int x_;
 public:
  Foo(int x);
  void setX(int x);
  int getX();
};
```

To generate a wrapper class that provides generic functions for the constructor and the `setX` and `getX` methods, we can use the following class definition:

```
; file: test-foo.scm

(require-extension tinyclos)

#>!
#include "Foo.h"
<#

(define x (make <Foo> 99))
(print (getX x))              ; prints ‘‘99’’
```

```
(setX x 42)
(print (getX x))              ; prints ‘‘42’’
(destroy x)
```

Provided the file `foo.o` contains the implementation of the class `Foo`, the given example could be compiled like this (assuming a UNIX like environment):

```
% csc test-foo.scm foo.o -c++
```

Here is another example, a minimal “Hello world” application for QT. We can see the three different ways of embedding C/C++ code in Scheme:

```
; compile like this:
; csc hello.scm -c++ -C -IQTDIR/include -L "-LQTDIR/lib -lqt"

(require-extension tinyclos)

; Include into generated code, but don’t parse:
#>
#include <qapplication.h>
#include <qpushbutton.h>
<#

; Parse but don’t embed: we only want wrappers for a few classes:
#>?
class QWidget
{
public:
  void resize(int, int);
  void show();
};

class QApplication
{
public:
  QApplication(int, char **);
  ~QApplication();
  void setMainWidget(QWidget *);
  void exec();
};

class QPushButton : public QWidget
{
public:
  QPushButton(char *, QWidget *);
  ~QPushButton();
}
<#

(define a (apply make <QApplication> (receive (argc+argv))))
```

```
(define hello (make <QPushButton> "hello world!" #f))
(resize hello 100 30)
(setMainWidget a hello)
(show hello)
(exec a)
(destroy hello)
(destroy a)
```

6.7.1 #> ... <# Syntax

Occurrences of the special read syntax `#>[SPEC ...] ...<#` will be handled according to `SPEC`:

If `SPEC` is the `?` character, the text following up to the next `<#` will be processed as a `(declare (foreign-parse "..."))` declaration (the code will be processed by the FFI parser described in this section).

If `SPEC` is the `!` character, the text will be embedded as

```
(declare
  (foreign-declare "...")
  (foreign-parse "...") )
```

It will be both included verbatim in the declaration section of the generated C/C++ file and processed by the FFI parser.

If `SPEC` is the `:` character, the text will be so it will be executed at the location where it appears.

If `SPEC` is the `$` character, the text will be parsed and the generated Scheme code is returned as a `(foreign-parse CODE)` form.

If `SPEC` is the `%` character, the text will be parsed into an s-expression that specifies the parsed entities, wrapped in a `(foreign-parse/spec ...)` form. See below for details of the specification format.

If `SPEC` is a list of the form `(TAG ...)`, then each `TAG` (which should be a symbol) specifies what should be done with the text:

`declare`	(declare (foreign-declare "..."))
`parse`	(declare (foreign-parse "..."))
`execute`	(foreign-code "...")
`code`	(foreign-parse "...")
`spec`	(foreign-parse/spec "...")

If any other character follows the `#>`, then the complete text will be included verbatim in the declaration part of the generated file (as in a `foreign-declare` declaration).

6.7.2 General operation

The parser will generally perform the following functions

1) Translate macro, enum-definitions and constants into `define-foreign-variable` or `define-constant` forms

2) Translate function prototypes into `foreign-lambda` forms

3) Translate variable declarations into accessor procedures

4) Handle basic preprocessor operations

5) Translate simple C++ class definitions into TinyCLOS wrapper classes and methods

Basic token-substitution of macros defined via `#define` is performed. The preprocessor commands `#ifdef`, `#ifndef`, `#else`, `#endif`, `#undef` and `#error` are handled. The preprocessor commands `#if` and `#elif` are not supported and will signal an error when encountered by the parser, because C expressions (even if constant) are not parsed. The preprocessor command `#pragma` is allowed but will be ignored.

During processing of `foreign-parse` declarations the macro `CHICKEN` is defined (similar to the C compiler option `-DCHICKEN`).

Macro- and type-definitions are available in subsequent `foreign-parse` forms. C variables declared generate a procedure with zero or one argument with the same name as the variable. When called with no arguments, the procedure returns the current value of the variable. When called with an argument, then the variable is set to the value of that argument. C and C++ style comments are supported. Variables declared as `const` will generate normal Scheme variables, bound to the initial value of the variable.

Function-, member-function and constructor/destructor definitions may be preceded by the `___safe` qualifier, which marks the function as (possibly) performing a callback into Scheme. If a wrapped function calls back into Scheme code, and `___safe` has not been given very strange and hard to debug problems will occur. For backward compatibilty, `___callback` is also allowed in place of `___safe`.

Functions and member functions prefixed with `___discard` and a result type that maps to a Scheme string (`c-string`), will have their result type changed to `c-string*` instead.

Constants (as declared by `#define` or `enum`) are not visible outside of the current Compilation units unless the `export_constants` pseudo declaration has been used.

When given the option `-ffi`, CHICKEN will compile a C/C++ file in “Scheme” mode, that is, it wraps the C/C++ source inside `#>! ... <#` and compiles it while generating Scheme bindings for exported definitions.

Function-arguments may be preceded by `___in`, `___out` and `___inout` qualifiers to specify values that are passed by reference to a function, or returned by reference. Only basic types (booleans, numbers and characters) can be passed using this method. During the call a pointer to a temporary piece of storage containing the initial value (or a random value, for `___out` parameters) will be allocated and passed to the wrapped function. This piece of storage is subject to garbage collection and will move, should a callback into Scheme occur that triggers a garbage collection. Multiple `__out` and `___inout` parameters will be returned as multiple values, preceded by the normal return value of thhe function (if not `void`). Here is a simple example:

```
#>!
#ifndef CHICKEN
#include <math.h>
#endif
```

```
double modf(double x, ___out double *iptr);
<#

(let-values ([(frac int) (modf 33.44)])
  ...)
```

Function-arguments may be preceded by `___length(ID)`, where `ID` designates the name of another argument that must refer to a number vector or string argument. The value of the former argument will be computed at run-time and thus can be omitted:

```
#>!
(require-extension srfi-4)

double sumarray(double *arr, ___length(arr) int len)
{
  double sum = 0;

  while(len--) sum += *(arr++);

  return sum;
}
<#

(print (sumarray (f64vector 33 44 55.66)))
```

The length variable may be positioned anywhere in the argument list. Length markers may only be specified for arguments passed as SRFI-4 byte-vectors, byte-vectors (as provided by the `lolevel` library unit) or strings.

Structure and union definitions containing actual field declarations generate getter procedures (and setter procedures when declared `___mutable` or the `mutable_fields` pseudo declaration has been used) The names of these procedures are computed by concatenating the struct (or union) name, a hyphen (`"-"`) and the field name, and the string `"-set!"`, in the case of setters. Structure definitions with fields may not be used in positions where a type specifier is normally expected. The field accessors operate on struct/union pointers only. Additionally a zero-argument procedure named `make-<structname>` will be generated that allocates enough storage to hold an instance of the structure (or union). Prefixing the definition with `___abstract` will omit the creation procedure.

```
#>!
struct My_struct { int x; ___mutable float y; };

typedef struct My_struct My_struct;

My_struct *make_struct(int x, float y)
{
  My_struct *s = (My_struct *)malloc(sizeof(My_struct));
  s->x = x;
  s->y = y;
  return s;
}
<#
```

will generate the following definitions:

```
(make-My_struct) -> PTR
(My_struct-x PTR) -> INT
(My_struct-y PTR) -> FLOAT
(My_struct-y-set! PTR FLOAT)
(make_struct INT FLOAT) -> PTR
```

Nested structs or unions are not supported (but pointers to nested structs/unions are).

All specially handled tokens preceded with `___` are defined as C macros in the headerfile `chicken.h` and will usually expand into nothing, so they don't invalidate the processed source code.

C++ `namespace` declarations of the form `namespace NAME { ... }` recognized but will be completely ignored.

Keep in mind that this is not a fully general C/C++ parser. Taking an arbitrary headerfile and feeding it to CHICKEN will in most cases not work or generate riduculuous amounts of code. This FFI facility is for carefully written headerfiles, and for declarations directly embedded into Scheme code.

6.7.3 Pseudo declarations

Using the `___declare(DECL, VALUE)` form, pseudo declarations can be embedded into processed C/C++ code to provide additional control over the wrapper generation. Pseudo declarations will be ignored when processed by the system's C/C++ compiler.

- abstract [values: <string>] Marks the C++ class given in `<string>` as being abstract, i.e. no constructor will be defined. Alternatively, a class definition may be prefixed with `___abstract`.
- class_finalizers [values: yes, no] Automatically generates calls to `set-finalizer!` so that any unused references to instances of subsequently defined C++ class wrappers will be destroyed. This should be used with care: if the embedded C++ object which is represented by the reclaimed TinyCLOS instance is still in use in foreign code, then unpredictable things will happen.
- mutable_fields [values: yes, no] Specifies that all struct or union fields should generate setter procedures (the default is to generate only setter procedures for fields declared `___mutable`).
- destructor_name [values: <string>] Specifies an alternative name for destructor methods (the default is `destroy`.
- export_constants [values: yes (default), no] Define a global variable for constant-declarations (as with `#define` or `enum`), making the constant available outside the current compilation unit. Use the values `yes/1` for switching constant export on, or `no/0` for switching it off.
- exception_handler [values: <string>] Defines C++ code to be executed when an exception is triggered inside a C++ class member function. The code should be one or more `catch` forms that perform any actions that should be taken in case an exception is thrown by the wrapped member function:

```
#>!
___declare(exception_handler, "catch(...) { return 0; }")

class Foo {
 public:
  Foo *bar(bool f) { if(f) throw 123; else return this; }
};
<#

(define f1 (make <Foo>))
(print (bar f1 #f))
(print (bar f1 #t))
```

will print `<Foo>` and `#f`, respectively.

- full_specialization [values: yes, no] Enables "full specialization" mode. In this mode all wrappers for functions, member functions and static member functions are created as fully specialized TinyCLOS methods. This can be used to handle overloaded C++ functions properly. Only a certain set of foreign argument types can be mapped to TinyCLOS classes, as listed in the following table:

`char`	<char>
`bool`	<bool>
`c-string`	<string>
`unsigned-char`	<exact>
`byte`	<exact>
`unsigned-byte`	<exact>
`[unsigned-]int`	<exact>
`[unsigned-]short`	<exact>
`[unsigned-]long`	<integer>
`[unsigned-]integer`	<integer>
`float`	<inexact>
`double`	<inexact>
`number`	<number>
`(enum _)char`	<exact>
`(const T)char`	(as T)

`(function ...)`	<pointer>
`c-pointer`	<pointer>
`(pointer _)`	<pointer>
`(c-pointer _)`	<pointer>
`u8vector`	<u8vector>
`s8vector`	<s8vector>
`u16vector`	<u16vector>
`s16vector`	<s16vector>
`u32vector`	<u32vector>
`s32vector`	<s32vector>
`f32vector`	<f32vector>
`f64vector`	<f64vector>

All other foreign types are specialized as `<top>`.

Full specialization can be enabled globally, or only for sections of code by enclosing it in

```
___declare(full_specialization, yes)
...
int foo(int x);
int foo(char *x);
...
___declare(full_specialization, no)
```

Alternatively, member function definitions may be prefixed by `___specialize` for specializing only specific members.

- prefix [values: <string>]

  Sets a prefix that should be be added to all generated Scheme identifiers. For example

  ```
  ___declare(prefix, "mylib:")
  #define SOME_CONST     42
  ```

  would generate the following code:

  ```
  (define-constant mylib:SOME_CONST 42)
  ```

  To switch prefixing off, use the values `no` or `0`. Prefixes are not applied to Class names.

- rename [value: <string>]

  Defines to what a certain C/C++ name should be renamed. The value for this declaration should have the form `"<c-name>;<scheme-name>"`, where `<c-name>` specifies the C/C++ identifier occurring in the parsed text and `<scheme-name>` gives the name used in generated wrapper code.
- scheme [value: <string>]

  Embeds the Scheme expression `<string>` in the generated Scheme code.
- substitute [value: <string>]

  Declares a name-substitution for all generated Scheme identifiers. The value for this declaration should be a string containing a regular expression and a replacement string (seperated by the `;` character):

  ```
  ___declare(substitute, "^SDL_;sdl:")

  extern void SDL_Quit();
  ```

  generates

  ```
  (define sdl:Quit
    (foreign-lambda integer "SDL_Quit") )
  ```
- transform [values: <string>]

  Defines an arbitrary transformation procedure for names that match a given regular expression. The value should be a string containing a regular expression and a Scheme expression that evaluates to a procedure of one argument. If the regex matches, the procedure will be called at compile time with the match-result (as returned by `string-match`) and should return a string with the desired transformations applied:

  ```
  (require-for-syntax 'srfi-13)

  #>!
  ___declare(transform, "([A-Z]+)_(.*);(lambda (x) (string-append (cadr x) \"-\" (string-d

  void FOO_Bar(int x) { return x * 2; }
  <#

  (print (FOO-bar 33))
  ```
- default_renaming [value: <string>]

  Chooses a standard name-transformation, converting underscores (`_`) to hyphens (`-`) and transforming “CamelCase” into “camel-case”. All uppercase characters are also converted to lowercase. The result is prefixed with the argument string (equivalent to the `prefix` pseudo declaration).
- type [value: <string>]

  Declares a foreign type transformation, similar to `define-foreign-type`. The value should be a list of two to four items, separated by the `;` character: a C typename, a Scheme foreign type specifier and optional argument- and result-value conversion procedures.

  ```
  ;;;; foreign type that converts to unicode (assumes 4-byte wchar_t):
  ```

```
;
; - Note: this is rather kludgy is only meant to demonstrate the ‘type’
;         pseudo-declaration

(require-extension srfi-4)

(define mbstowcs (foreign-lambda int "mbstowcs" nonnull-u32vector c-string int))

(define (str->ustr str)
  (let* ([len (string-length str)]
         [us (make-u32vector (add1 len) 0)] )
    (mbstowcs us str len)
    us) )

#>!
___declare(type, "unicode;nonnull-u32vector;str->ustr")

static void foo(unicode ws)
{
  printf("\"%ls\"\n", ws);
}
<#

(foo "this is a test!")
```

- opaque [value: <string>]

  Similar to `type`, but provides automatic argument- and result conversions to wrap a value into a structure:

  ```
  #>?
  ___declare(opaque, "myfile;(pointer \"FILE\")")

  myfile fopen(char *, char *);
  <#

  (fopen "somefile" "r")   ==> <myfile>
  ```

  `___declare(opaque, "TYPENAME;TYPE")` is basically equivalent to `___declare(type, "TYPENAME;TYPE;TYPE->RECORD;RECORD->TYPE")` where `TYPE->RECORD` and `RECORD->TYPE` are compiler-generated conversion functions that wrap objects of type `TYPE` into a record and back.

6.7.4 Grammar

The parser understand the following grammar:

```
PROGRAM = PPCOMMAND
        | DECLARATION ";"

PPCOMMAND = "#define" ID [TOKEN ...]
```

```
          | "#ifdef" ID
          | "#ifndef" ID
          | "#else"
          | "#endif"
          | "#undef" ID
          | "#error" TOKEN ...
          | "#include" INCLUDEFILE
          | "#pragma" TOKEN ...

DECLARATION = FUNCTION
            | VARIABLE
            | ENUM
            | TYPEDEF
            | CLASS
            | CONSTANT
            | STRUCT
            | NAMESPACE
            | "___declare" "(" PSEUDODECL "," <tokens> ")"

STRUCT = ("struct" | "union") ID ["{" {["___mutable"] TYPE {"*"} ID {"," {"*"} ID}} "}]

NAMESPACE = "namespace" ID "{" DECLARATION ... "}"

INCLUDEFILE = "\"" ... "\""
            | "<" ... ">"

FUNCTION = {"___callback" | "___safe" | "___specialize" | "___discard"} [STORAGE] TYPE ID "
         | {"___callback" | "___safe" | "___specialize" | "___discard"} [STORAGE] TYPE ID "

ARGTYPE = [IOQUALIFIER] TYPE [ID ["[" ... "]"]]

IOQUALIFIER = "___in" | "___out" | "___inout" | LENQUALIFIER

LENQUALIFIER = "___length" "(" ID ")"

VARIABLE = [STORAGE] ENTITY ["=" INITDATA]

ENTITY = TYPE ID ["[" ... "]"]

STORAGE = "extern" | "static" | "volatile" | "inline"

CONSTANT = "const" TYPE ID "=" INITDATA

PSEUDODECL = "export_constants"
           | "prefix"
           | "substitute"
           | "abstract"
           | "type"
           | "scheme"
           | "rename"
           | "transform"
           | "full_specialization"
           | "destructor_name"
           | "class_finalizers"
           | "exception_handler"
           | "mutable_fields"

ENUM = "enum" "{" ID ["=" NUMBER] "," ... "}"

TYPEDEF = "typedef" TYPE ["*" ...] [ID]

TYPE = ["const"] BASICTYPE [("*" ... | "&" | "<" TYPE "," ... ">" | "(" "*" [ID] ")" "(" TY

BASICTYPE = ["unsigned" | "signed"] "int"
          | ["unsigned" | "signed"] "char"
          | ["unsigned" | "signed"] "short" ["int"]
          | ["unsigned" | "signed"] "long" ["int"]
          | ["unsigned" | "signed"] "___byte"
          | "size_t"
          | "float"
          | "double"
          | "void"
          | "bool"
          | "___bool"
          | "___scheme_value"
          | "___scheme_pointer"
          | "___byte_vector"
          | "___pointer" TYPE "*"
          | "C_word"
          | "___fixnum"
          | "___number"
          | "___symbol"
          | "struct" ID
          | "union" ID
          | "enum" ID
          | ID

CLASS = ["___abstract"] "class" ID [":" [QUALIFIER] ID "," ...] "{" MEMBER ... "}"█

MEMBER = [QUALIFIER ":"] ["virtual"] (MEMBERVARIABLE | CONSTRUCTOR | DESTRUCTOR | MEMBERFUN

MEMBERVARIABLE = TYPE ID ["=" INITDATA]

MEMBERFUNCTION = {"___callback" | "static" | "___specialize" | "___discard"} TYPE ID "(" AR
               | {"___callback" | "static" | "___specialize" | "___discard"} TYPE ID "(" "v

CONSTRUCTOR = ["___callback" | "___safe"] ["explicit"] ID "(" ARGTYPE "," ... ")" [BASECONS

DESTRUCTOR = ["___callback" | "___safe"] "~" ID "(" ["void"] ")" [CODE]

QUALIFIER = ("public" | "private" | "protected")

NUMBER = <a C integer or floating-point number, in decimal, octal or hexadecimal notation>

INITDATA = <everything up to end of chunk>

BASECONSTRUCTORS = <everything up to end of chunk>

CODE = <everything up to end of chunk>
```

The following table shows how argument-types are translated:

`[unsigned] char`	char
`[unsigned] short`	[unsigned-]short
`[unsigned] int`	[unsigned-]integer
`[unsigned] long`	[unsigned-]long
`float`	float
`double`	double
`size_t`	unsigned-integer
`bool`	int
`___bool`	int
`___fixnum`	int
`___number`	number
`___symbol`	symbol
`___scheme_value`	scheme-object
`C_word`	scheme-object
`___scheme_pointer`	scheme-pointer

`char *`	c-string
`signed char *`	s8vector
`[signed] short *`	s16vector
`[signed] int *`	s32vector
`[signed] long *`	s32vector
`unsigned char *`	u8vector
`unsigned short *`	u16vector
`unsigned int *`	u32vector
`unsigned long *`	u32vector
`float *`	f32vector
`double *`	f64vector
`___byte_vector`	byte-vector
`CLASS *`	(instance CLASS <CLASS>)
`CLASS &`	(instance-ref CLASS <CLASS>)
`TYPE *`	(pointer TYPE)
`TYPE`	&(ref TYPE)
`TYPE<T1, ...>`	(template TYPE T1 ...)
`TYPE1 (*)(TYPE2, ...)`	(function TYPE1 (TYPE2 ...))

The following table shows how result-types are translated:

`void`	void
`[unsigned] char`	char
`[unsigned] short`	[unsigned-]short
`[unsigned] int`	[unsigned-]integer

`[unsigned] long`	[unsigned-]long
`float`	float
`double`	double
`size_t`	unsigned-integer
`bool`	bool
`___bool`	bool
`___fixnum`	int
`___number`	number
`___symbol`	symbol
`___scheme_value`	scheme-object
`char *`	c-string
`TYPE *`	(pointer TYPE)
`TYPE`	&(ref TYPE)
`TYPE<T1, ...>`	(template TYPE T1 ...)
`TYPE1 (*)(TYPE2, ...)`	(function TYPE1 (TYPE2 ...))
`CLASS *`	(instance CLASS <CLASS>)
`CLASS &`	(instance-ref CLASS <CLASS>)

The `___pointer` argument marker disables automatic simplification of pointers to numbers: normally arguments of type `int *` are handled as SRFI-4 `s32vector` number vectors. To force treatment as a pointer argument, precede the argument type with `___pointer`.

6.7.5 C notes

Foreign variable definitions for macros are not exported from the current compilation unit, but definitions for C variables and functions are.

`foreign-parse` does not embed the text into the generated C file, use `foreign-declare` for that (or even better, use the `#>! ... <#` syntax which does both).

Functions with variable number of arguments are not supported.

6.7.6 C++ notes

Each C++ class defines a TinyCLOS class, which is a subclass of `<c++-object>`. Instances of this class contain a single slot named `this`, which holds a pointer to a heap-allocated C++ instance. The name of the TinyCLOS class is obtained by putting the C++ classname between angled brackets (`<...>`). TinyCLOS classes are not seen by C++ code.

The C++ constructor is invoked by the `initialize` generic, which accepts as many arguments as the constructor. If no constructor is defined, a default-constructor will be provided taking no arguments. To allow creating class instances from pointers created in foreign code, the `initialize` generic will optionally accept an arguments list of the form `'this POINTER`, where `POINTER` is a foreign pointer object. This will create a TinyCLOS instance for the given C++ object.

To release the storage allocated for a C++ instance invoke the `destroy` generic (the name can be changed by using the `destructor_name` pseudo declaration).

Static member functions are wrapped in a Scheme procedure named `<class>::<member>`.

Member variables and non-public member functions are ignored.

Virtual member functions are not seen by C++ code. Overriding a virtual member function with a TinyCLOS method will not work when the member function is called by C++.

Operator functions and default arguments are not supported.

Exceptions must be explicitly handled by user code and may not be thrown beyond an invocation of C++ by Scheme code.

6.7.7 Using the builtin parser

There are two macros that can be used (in addition to the `#>$ ... <#` reader syntax) to access the parsed C/C++ code directly. These macros are by default undefined. To perform custom processing of the parsed Scheme code or FFI specification, define the macros as you please.

syntax [foreign-parse]

```
(foreign-parse EXP)
```

Should be defined to perform custom processing of Scheme code wrapped in a `#>$ ... <#` construct.

syntax [foreign-parse/spec]

```
(foreign-parse/spec EXP)
```

Should be defined to perform custom processing of an FFI specification wrapped in a `#>% ... <#` construct.

For example:

```
(define-macro (foreign-parse/spec x)
  `(pp ',x) )

#>%
```

```
int foo(double n) { return n * 2; }

typedef short *mytype;

class Bar: public Yo, Foo {
public:
  Bar(int, char *);
  ~Bar();

  void *myfun(mytype x);
};
<#
```

will expand into:

```
(pp '(begin
       (function (foo "foo") integer double)
       (class (<Bar> "Bar") (<c++-object>))
       (destructor (<Bar> "Bar") destroy)
       (constructor (<Bar> "Bar") integer c-string)
       (imethod
         (myfun "myfun")
         (<Bar> "Bar")
         (pointer void)
         (<s16vector> s16vector))))
```

6.7.8 Specification grammar

The following table describes the grammar of specifications parsed by the `#>% ... <#` construct.

`IDs` is a symbol, naming a Scheme identifier. `IDc` is a string naming a C/C++ entity. `VALUE` is a Scheme numeric constant. `TYPE` is a foreign type specifier. Most identifiers are given in pairs of the form `(IDs IDc)`, where `IDs` is a version of the C identifier `IDc` with all renamings performed (and in the case of class-names enclosed in in `<...>`).

`(begin SPEC ...)`
: Contains zero or more specification items.

`(alias (IDs IDc) TYPE)`
: Defines an alias of the name `IDs` for the C/C++ entity `IDc` (usually a variable) with the type `TYPE`.

`(constant IDs VALUE)`
: Defines a named constant.

`(type IDs IDs [EXP1 [EXP2]]))`
: A user defined foreign type (just like `define-foreign-type`).

`(variable (IDs IDc) TYPE)`
: A variable.

`(cvariable (IDs IDc) TYPE)`
: A constant (immutable) variable.

`(function (IDs IDc) TYPEr TYPEa ...)`
: A function with the result type `TYPEr` and the argument types `TYPEa ...`.

`(function* (IDs IDc) TYPEr TYPEa ...)`
: A function that may invoke callbacks.

`(method (IDs IDc) TYPEr (CLASS TYPEa) ...)`
: A generic function method. The corresponding TinyCLOS classes for each argument type are also provided.

`(method* (IDs IDc) TYPEr (CLASS TYPEa) ...)`
: A generic function method that may invoke callbacks.

`(imethod (ID1s ID1c) (ID1s ID1c) TYPEr (CLASS TYPEa) ...)`
: An instance method. The first identifier-pair names the method and the second names the class.

`(class (IDs IDc) (IDs ...))`
: A class definition. The list `(IDs ...)` names the super-classes and defaults to `<c++-object>`.

`(constructor (IDs IDc) TYPEa ...)`
: A class constructor with a given list of argument types.

`(constructor* (IDs IDc) TYPEa ...)`
: A class constructor that should generate instances which are automatically finalized.

`(destructor (IDs IDc) IDs)`
: A class destructor. The second identifier specifies how the destructor method should be named.

`(export IDs)`
: Make the Scheme identifier `IDs` be globally visible (it is assumed constants and aliases are not globally visible by default).

`(struct-field IDc FIELDIDs TYPE GETTER [SETTER])`
: A structure field with the name `FIELDIDs` of the structure named `IDc` and with the type `TYPE`. `GETTER` and `SETTER` (when given) specify the names of any accessor procedures.

`(union-field IDc FIELDIDs TYPE GETTER [SETTER])`
: As above, but for a union field.

`(struct-maker IDc CONSIDs (TYPE FIELDIDc) ...)`
: A structure constructor for the C struct `IDc` field the name `CONSIDs` and the field-initializer arguments and types given in `(TYPE FIELDIDc) ...`.

`(union-maker IDc CONSIDs (TYPE FIELDIDc) ...)`
: A union-constructor.

6.8 C interface

The following functions and macros are available for C code that invokes Scheme or foreign procedures that are called by Scheme:

`void` **C_save** (`C_word x`) [C function]

Saves the Scheme data object `x` on the temporary stack.

`C_word` **C_fix** (`int integer`) [C macro]
`C_word` **C_make_character** (`int char_code`) [C macro]
`C_word` **C_SCHEME_END_OF_LIST** [C macro]
`C_word` **C_SCHEME_END_OF_FILE** [C macro]
`C_word` **C_SCHEME_FALSE** [C macro]
`C_word` **C_SCHEME_TRUE** [C macro]

These macros return immediate Scheme data objects.

`C_word` **C_string** (`C_word **ptr, int length, char *string`) [C function]
`C_word` **C_string2** (`C_word **ptr, char *zero_terminated_string`) [C function]
`C_word` **C_intern2** (`C_word **ptr, char *zero_terminated_string`) [C function]
`C_word` **C_intern3** (`C_word **ptr, char *zero_terminated_string, C_word initial_value`) [C function]
`C_word` **C_pair** (`C_word **ptr, C_word car, C_word cdr`) [C function]
`C_word` **C_flonum** (`C_word **ptr, double number`) [C function]
`C_word` **C_int_to_num** (`C_word **ptr, int integer`) [C function]
`C_word` **C_mpointer** (`C_word **ptr, void *pointer`) [C function]
`C_word` **C_vector** (`C_word **ptr, int length, ...`) [C function]
`C_word` **C_list** (`C_word **ptr, int length, ...`) [C function]

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`.

`C_word*` **C_alloc** (`int words`) [C macro]

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.

`int` **C_SIZEOF_LIST** (`int length`) [C macro]
`int` **C_SIZEOF_STRING** (`int length`) [C macro]
`int` **C_SIZEOF_VECTOR** (`int length`) [C macro]
`int` **C_SIZEOF_INTERNED_SYMBOL** (`int length`) [C macro]
`int` **C_SIZEOF_PAIR** [C macro]
`int` **C_SIZEOF_FLONUM** [C macro]
`int` **C_SIZEOF_POINTER** [C macro]
`int` **C_SIZEOF_LOCATIVE** [C macro]
`int` **C_SIZEOF_TAGGED_POINTER** [C macro]

These are macros that return the size in words needed for a data object of a given type.

`int` **C_character_code** (`C_word character`) [C macro]
`int` **C_unfix** (`C_word fixnum`) [C macro]
`double` **C_flonum_magnitude** (`C_word flonum`) [C macro]
`char*` **C_c_string** (`C_word string`) [C function]
`int` **C_num_to_int** (`C_word fixnum_or_flonum`) [C function]
`void*` **C_pointer_address** (`C_word pointer`) [C function]

These macros and functions can be used to convert Scheme data objects back to C data.

`int` **C_header_size** (`C_word x`) [C macro]
`int` **C_header_bits** (`C_word x`) [C macro]

Return the number of elements and the type-bits of the non-immediate Scheme data object `x`.

`C_word` **C_block_item** (`C_word x, int index`) [C macro]

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.

`C_word` **C_u_i_car** (`C_word x`) [C macro]
`C_word` **C_u_i_car** (`C_word x`) [C macro]

Aliases for `C_block_item(x, 0)` and `C_block_item(x, 1)`, respectively.

`void*` **C_data_pointer** (`C_word x`) [C macro]

Returns a pointer to the data-section of a non-immediate Scheme object.

`C_word` **C_make_header** (`C_word bits, C_word size`) [C macro]

A macro to build a Scheme object header from its bits and size parts.

`C_word` **C_mutate** (`C_word *slot, C_word val`) [C function]

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.

`C_word` **C_symbol_value** (`C_word symbol`) [C macro]

Returns the global value of the variable with the name `symbol`.

`void` **C_gc_protect** (`C_word *ptrs[]`, `int n`) [C function]

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 iniitalized (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`.

`void` **C_gc_unprotect** (`int n`) [C function]

Removes the last `n` registered variables from the set of root variables.

`void` **(*C_post_gc_hook)(int mode)** [C Variable]

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' */
}

% chicken foo.scm -quiet
% gcc foo.c bar.c -o foo `chicken-config -cflags -libs`
% 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 Scheme->C->Scheme 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.

7 chicken-setup

7.1 Extension libraries

Extension libraries 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 this are most UNIX-compatible platforms that provide the `libdl` functionality like Linux, Solaris, BSD or Mac OS X. Windows with Cygwin is currently not supported. Windows with the Microsoft tools is partially supported.

Note: Extension may also be normal applications or shell scripts.

7.2 Installing extensions

To install an extension library, run the `chicken-setup` program with the extension name as argument. If the extension consists of a single Scheme file, then it is compiled and installed in the extension *repository*. If it is an archive containing addition files, then the files are 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 of course 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.

7.3 Creating extensions

Extensions can be created by creating an 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.

7.4 Procedures and macros available in setup scripts

install-extension [procedure]

```
(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.

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:

syntax [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.

require-at-runtime [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.

version [property]

```
(version STRING)
```

Specifies version string.

All other properties are currently ignored. The `FILELIST` argument may also be a single string.

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`).

install-script [procedure]

```
(install-program ID FILELIST [INFOLIST])
```

Similar to `install-program`, but additionally changes the file permissions of all files in `FILELIST` to executable (for installing shell-scripts).

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.

compile [syntax]

```
(compile FORM ...)
```

Equivalent to `(run (csc FORM ...))`.

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.

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.

7.5 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 " !") )
```

After entering

```
$ chicken-setup hello
```

at the shell prompt, the file `hello.scm` will be compiled into a dynamically loadable library, with the default compiler options `-optimize-level 2 -debug-level 0 -shared`. If the compilation succeeds, `hello.so` will be stored in the repository, together with a file named `hello.setup` (not to be confused with a setup script - this `.setup` file just contains an a-list with metadata).

Use it like any other CHICKEN extension:

```
$ csi -quiet
#;1> (require-extension hello)
; loading /usr/local/lib/chicken/hello.so ...
#;2> (hello "me")
Hello, me!
#;3>
```

For more elaborate build operations, when installing applications or scripts, or when additional metadata should be stored for an extension, a `setup` script is required and the script and all additional files should be packaged in a gzipped `tar` archive.

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

(run (csc hello2.scm))  ; compile ‘hello2’
(install-program ’hello2 "hello2") ; name of the extension and files to be installed
```

To use it, just run `chicken-setup` in the same directory:

```
$ chicken-setup hello2
```

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.

Uninstallation is just as easy:

```
$ chicken-setup -uninstall hello2
```

`chicken-setup` provides a `make` tool, so building 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.

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)]
[,y (and ,yn (car ,y))] )
   (case ,x
     ,@(map (lambda (v)
      `([,v] (if ,yn
 (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:

```
(run (csc -s -O2 -d0 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:

```
$ csi -quiet
#;1> (require-extension glass)
; loading /usr/local/lib/chicken/glass.scm ...
; loading /usr/local/lib/chicken/glass-support.so ...
#;2> (define foo (glass-lambda (x) (y) (+ x y)))
#;3> (glass-lambda-set! foo 'y 99)
#;4> (foo 33)
132
```

7.6 `chicken-setup` reference

Available options:

- `-help` Show usage information and exit.
- `-version` Display version and exit.
- `-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.
- `-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.
- `-host HOSTNAME[:PORT]` Specifies alternative host for downloading extensions, optionally with a TCP port number (which defaults to 80).
- `-uninstall EXTENSION` Removes all files that were installed for `EXTENSION` from the file-system, together with any metadata that has been stored.

- `-list` List all installed extensions and exit.
- `-run FILENAME` Load and execute given file.
- `-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`.
- `-verbose` Display additional debug information.
- `-keep` Keep temporary files and directories.
- `-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 ...))`.
- `-dont-ask` Do not ask the user before trying to download required extensions.
- `-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.
- `--` Ignore all following arguments.

Note that the options are processed exactly in the order in which they appear in the command-line.

7.7 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.

8 Additional files

In addition to library units the following files are provided. Use them by including the file in your code with the `include` special form.

8.1 `chicken-more-macros.scm`

This file contains the definitions of all non-standard syntax forms. You normally don't use this file directly, unless you have the following situation: you use non-standard macros at run-time (in evaluated code) in a compiled program and you want non-standard syntax to be available. In this case, add

```
(require-extension chicken-more-macros)
```

to your code. This will load the definitions for non-standard macros available in code evaluated by the program.

See also the FAQ for a discussion about the different macro systems and their ideosyncrasies.

8.2 `chicken-ffi-macros.scm`

This file contains the definitions of macros for interfacing to foreign code, and the definitions contained in this file are automatically made available in compiled code.

8.3 `chicken-entry-points.scm`

This file contains the definition of the macros `define-entry-point` and `define-embedded`. See the section "Entry points" earlier in this manual.

8.4 `chicken-default-entry-points.scm`

This file contains boilerplate entry point definitions. See the section "Entry points". This file automatically includes `entry-points.scm`.

9 Data Representation

There exist two different kinds of data objects in the CHICKEN system: immediate and non-immediate objects. Immediate objects are represented by a tagged machine word, which is usually of 32 bits length (64 bits on 64-bit architectures). The immediate objects come in four different flavors:

- **fixnums**, that is, small exact integers, distinguished by the lowest order bit in the machine word set to 1. This gives fixnums a range of 31 bits for the actual numeric value (63 bit on 64 bit architectures).
- **characters**, where the lowest four bits of machine words containing characters are equal to `C_CHARACTER_BITS`. The ASCII code of the character is encoded in bits 9 to 16, counting from 1 and starting at the lowest order position.
- **booleans**, where the lowest four bits of machine words containing booleans are equal to `C_BOOLEAN_BITS`. Bit 5 (counting from 0 and starting at the lowest order position) is one if the boolean designates true, or 0 if it is false.
- other values: the empty list, void and end-of-file. The lowest four bits of machine words containing these values 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_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 (starting at the lowest order position) 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-points 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`.

10 Bugs and limitations

- Compiling large files takes too much time.
- There is no support for rationals, complex numbers or extended-precision integers (bignums).
- The maximal number of arguments that may be passed to a compiled procedure or macro is 126. 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 be configured at build time to take advantage of this. See the `README` file for more details.
- The maximum number of values that can be passed to continuations captured using `call-with-current-continuation` is 126.
- Some numeric procedures are missing since CHICKEN does not support the full numeric tower. Support for extended numbers (bignums, exact rationals and complex numbers) is available as a separate package, provided the GNU multiprecision library is installed.
- 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.
- Building CHICKEN on RS/6000 systems under AIX is currently not possible, due to strange assembler errors during compilation of the compiler sources.
- `format` is not reentrant. This means that recursive invocation of this procedure (either inside `print-object` methods or record-printer defined with `define-record-printer` will not work.
- User-defined types are currently not supported in the argument and result specifications for entry-points defined with `define-entry-point`.

11 FAQ

11.1 General

- 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 existance 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 I know, CHICKEN is the first implementation of Scheme that uses Henry Baker's "Cheney on the M.T.A" concept.
- What to 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.
- 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.
- 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.
- How can I cut down the size of an executable?

  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 excutable 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.

- 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)

- 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: `-O3 -fomit-frame-pointer -f-nostrict-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).

- 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):

  ```
  % chicken program.scm -output-file program.c
  % gcc program.c foo.o ‘chicken-config -cflags -libs‘ -o program
  ```

- 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 (see Q23). Just remove the declaration, or compile this file to an object-module and link it to your application code.

- 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.

- When I compile a file with `-unsafe` or unsafe declarations, it crashes during execution.

  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.

- Which non-standard procedures are treated specially when the `extended-bindings` or `usual-integrations` declaration or compiler option is used?

  The following extended bindings are handled specially:

  `bitwise-and bitwise-ior bitwise-xor bitwise-not add1 sub1 fx+ fx- fx* fx/ fxmod fx= fx> fx>= fixnum? fxneg fxmax fxmin fxand fxior fxxor fxnot fxshl fxshr 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 u8vector-set! s8vector-set! u16vector-set! s16vector-set! u32vector-set! s32vector-set! 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`

- 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))
  ```

- Why does my program abort with an "out of memory" error when I compile with extended debug information?

The compiler instruments the source file with a lot of support code to provide things like arbitrary restarting/returning from activation frames. This will transform calls in tail-position into non-tail calls in all situations but the most trivial ones (`do` loops and named `let`). This will result in a much higher memory usage for allocating continuation frames. Try passing the `-:hXXX` option when executing the program, with a large value for `XXX`.

- 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 in one of three ways:

  - Use a different name

  - Undefine the macro, like this:

  ```
  (eval-when (compile eval) (undefine-macro! 'match))
  ```

- 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`.

- When I use callback functions (from Scheme to C and back to Scheme again), I get weird crashes.

  There are two reasons why code involving callbacks can crash out of know apparent reason. The first is that 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. The second reason is that 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.

- How can I change `match-error-control` during compilation?

  Use `eval-when`, like this:

  ```
  (eval-when (compile)
    (match-error-control #:unspecified) )
  ```

- Why doesn't CHICKEN support the full numeric tower?

  There are a number of reasons for this:

  - I haven't found a decent library for multiprecision arithmetic, yet (the GNU multi-precision library is distributed under a different license than CHICKEN);

  - 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.

- Toplevel-continuations captured in interpreted code don't seem to 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.

- 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>)) ...)
  ```

- Why does a chicken have both dark and white meats?

  Short answer is chickens don't fly long distance. As a result the muscles in the wings are designed for short bursts of activity and don't have as many of the proteins that facilitate oxygen distribution in those muscles. These proteins are coloured red and cause the dark colour of the meat. Ducks fly a lot hand have dark meat in their wings. Chickens do walk a lot so their leg and thighs are dark meat.

  (Thanks to Chris Double)

11.2 Platform specific

- How do I generate a DLL under MS Windows (tm) ?

  Use `csc` in combination with the `-dll` option:

  ```
  C:\> csc foo.scm -dll
  ```

  You can use the `define-entry-point` facility to interface to the Scheme code.

- 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.

- 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.
- 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.
- 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.
- 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.

11.3 Customization

- 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;
  ```
- 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 a UNIX like environment and also 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) ) ) ) )
...
% chicken userpass.scm -output-file userpass.c -explicit-use -quiet
% chicken chicken.scm -output-file chicken-extended.c -quiet -postlude "(declare (uses
% gcc -c userpass.c ‘chicken-config -cflags‘ -o userpass.o
% gcc -c chicken-extended.c ‘chicken-config -cflags‘ -o chicken-extended.o
% gcc chicken-extended.o support.o easyffi.o compiler.o optimizer.o batch-driver.o c-p
  c-backend.o userpass.o ‘chicken-config -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).

11.4 Macros

- Why does `(define-macro foo bar)` not work?

  Consider this code snippet:

  ```
  (define (double x) (list '* x x))

  (define-macro twice double)
  ```

  Because the macro `twice` is defined at compile time (following forms may refer to it), functions defined at runtime (as `double` in this case) are not available. The alternative syntax of `define-macro`:

  ```
  (define-macro twice (lambda (x) (list '* x x)))
  ```

  does just exist to make porting other Scheme code easier. It is not able to assign procedures computed at runtime as macro expanders.
- 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.
- 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.

12 Acknowledgements

Many thanks to:

William Annis, Marc Baily, Peter Barabas, Jonah Beckford, Peter Bex, Dave Bodenstab, Fabian Bhlke, T. Kurt Bond, Terence Brannon, Roy Bryant, Category 5, Taylor Campbell, Franklin Chen, Gian Paolo Ciceri, Grzegorz Chrupala, James Crippen, Tollef Fog Heen, Alejandro Forero Cuervo, Linh Dang, Brian Denheyer, Chris Double, Petter Egesund, Steve Elkins, Daniel B. Faken, Graham Fawcett, Fizzie, Kimura Fuyuki, Tony Garnock-Jones, Martin Gasbichler, Joey Gibson, Johannes Groedem, Andreas Gustafsson, Sven Hartrumpf, Jun-ichiro itojun Hagino, Matthias Heiler, Karl M. Hegbloom, William P. Heinemann, Bruce Hoult, Hans Hbner, Christian Jaeger, Dale Jordan, Valentin Kamyshenko, Peter Keller, Ron Kneusel, Matthias Koeppe, Krysztof Kowalczyk, Todd R. Kueny Sr, Micky Latowicki, Kirill Lisovsky, Kon Lovett, Dennis Marti, Charles Martin, Alain Mellan, Eric Merrit, Perry Metzger, Scott G. Miller, Mikael, Bruce Mitchener, Chris Moline, Eric E. Moore, Julian Morrison, Lars Nilsson, o.t., Davide Puricelli, Doug Quale, Eric Raible, Joel Reymont, David Rush, Lars Rustemeier, Oskar Schirmer, Burton Samograd, Reed Sheridan, Ronald Schr&ouml;der, Spencer Schumann, Shmul, Jeffrey B. Siegal, Michele Simionato, Dorai Sitaram, Robert Skeels, Jason Songhurst, Clifford Stein, Zbigniew Szadkowski, Mike Thomas, Christian Tismer, Andre van Tonder, John Tobey, Vladimir Tsichevsky, Neil van Dyke, Sander Vesik, Panagiotis Vossos, Shawn Wagner, Peter Wang, Ed Watkeys, Thomas Weidner, Matthew Welland, Joerg Wittenberger, Mark Wutka, Richard Zidlicky and Houman Zolfaghari for bug-fixes, tips and suggestions.

Special thanks to Benedikt Rosenau for his continuous moral support.

CHICKEN contains code from several people:

- Eli Barzilay: some performance tweaks used in TinyCLOS.
- Anthony Carrico: the option parsing facility.
- Mikael Djurfeldt: topological sort used by compiler.
- Marc Feeley: pretty-printer.
- Aubrey Jaffer: implementation of `dynamic-wind`.
- Gregor Kiczales: original implementation of TinyCLOS.
- Dirk Lutzebaeck: Common LISP `format`.
- Richard O'Keefe: sorting routines.
- Olin Shivers: implementation of `let-optionals[*]` and reference implementations of SRFI-1, SRFI-13 and SRFI-14.
- Dorai Sitaram: the PREGEXP regular expression package and TeX2page, which was used to generate the HTML documentation.
- Andrew Wilcox: queues.
- Andrew Wright: pattern matcher.

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`

Index