Gambit-C, version 4.0 beta 7

A portable implementation of Scheme
Edition 4.0 beta 7, September 2004

Marc Feeley

1 Gambit-C: a portable version of Gambit

The Gambit programming system is a full implementation of the Scheme language which conforms to the R4RS and IEEE Scheme standards. It consists of two main programs: `gsi`, the Gambit Scheme interpreter, and `gsc`, the Gambit Scheme compiler.

Gambit-C is a version of the Gambit programming system in which the compiler generates portable C code, making the whole Gambit-C system and the programs compiled with it easily portable to many computer architectures for which a C compiler is available. With appropriate declarations in the source code the executable programs generated by the compiler run roughly as fast as equivalent C programs.

For the most up to date information on Gambit and add-on packages please check the Gambit web page at '`http://www.iro.umontreal.ca/~gambit`'. Bug reports and inquiries should be sent to '`gambit@iro.umontreal.ca`'.

1.1 Accessing the Gambit system files

Unless the default is overridden when the Gambit-C system was built (with the command '`configure --prefix=/my/own/directory`'), all files are installed in '`/usr/local/Gambit-C`' under UNIX and Mac OS X and '`C:\Gambit-C`' under Microsoft Windows. This is the *Gambit installation directory*.

The system's executables including the interpreter '`gsi`' and compiler '`gsc`' are stored in the '`bin`' subdirectory of the Gambit installation directory. It is convenient to put the '`bin`' directory in the shell's '`PATH`' environment variable so that these programs can be invoked simply by entering their name.

The runtime library is located in the '`lib`' subdirectory. When the system's runtime library is built as a shared-library (with the command '`configure --enable-shared`') all programs built with Gambit-C, including the interpreter and compiler, need to find this library when they are executed and consequently this directory must be in the path searched by the system for shared-libraries. This path is normally specified through an environment variable which is '`LD_LIBRARY_PATH`' on most versions of UNIX, '`LIBPATH`' on AIX, '`SHLIB_PATH`' on HPUX, '`DYLD_LIBRARY_PATH`' on Mac OS X, and '`PATH`' on Microsoft Windows. If the shell is of the '`sh`' family, the setting of the path can be made for a single execution by prefixing the program name with the environment variable assignment, as in:

```
% LD_LIBRARY_PATH=/usr/local/Gambit-C/lib gsi
```

A similar problem exists with the Gambit header file '`gambit.h`', located in the '`include`' subdirectory. This header file is needed for compiling Scheme programs with the Gambit-C compiler. If the C compiler is being called explicitly, then it may be necessary to use a '`-I<dir>`' command line option to indicate where to find header files and a '`-L<dir>`' command line option to indicate where to find libraries. Access to both of these files can be simplified by creating a link to them in the appropriate system directories (special privileges may however be required):

```
% ln -s /usr/local/Gambit-C/lib/libgambc.a /usr/lib # name may vary
% ln -s /usr/local/Gambit-C/include/gambit.h /usr/include
```

This is not done by the installation process. Alternatively these files can also be copied or linked in the directory where the C compiler is invoked (this requires no special privileges).

2 The Gambit Scheme interpreter

Synopsis:

```
gsi [-:runtimeoption,...] [-i] [-f] [[-] [-e expressions] [file]]...
```

The interpreter is executed in *interactive mode* when no file or '`-`' or '`-e`' option is given on the command line. When at least one file or '`-`' or '`-e`' option is present the interpreter is executed in *batch mode*. The '`-i`' option is ignored by the interpreter. When the '`-f`' option is absent the interpreter will examine the initialization file (see ⟨undefined⟩ [GSI customization], page ⟨undefined⟩). Runtime options are explained in ⟨undefined⟩ [Runtime options], page ⟨undefined⟩.

2.1 Interactive mode

In interactive mode a read-eval-print loop (REPL) is started for the user to interact with the interpreter. At each iteration of this loop the interpreter displays a prompt, reads a command and executes it. The commands can be Scheme expressions to evaluate (the typical case) or special commands related to debugging, for example '`,q`' to terminate the current thread (for a complete list of commands see ⟨undefined⟩ [Debugging], page ⟨undefined⟩). Most commands produce some output, such as the value or error message resulting from an evaluation.

The input and output of the interaction is done on the *interaction channel*. The interaction channel can be specified through the runtime options but if none is specified the system uses a reasonable default that depends on the system's configuration. When the system's runtime library was built with support for the IDE (with the command '`configure --enable-ide`') the interaction channel corresponds to the *console window* of the primordial thread (for details see ⟨undefined⟩ [IDE], page ⟨undefined⟩), otherwise the interaction channel is the user's *console*, also known as the *controlling terminal* in the UNIX world. When the REPL first starts, the ports associated with '`(current-input-port)`', '`(current-output-port)`' and '`(current-error-port)`' all refer to the interaction channel.

Expressions are evaluated in the global *interaction environment*. The interpreter adds to this environment any definition entered using the `define` and `define-macro` special forms. Once the evaluation of an expression is completed, the value or values resulting from the evaluation are output to the interaction channel by the pretty printer. The special "void" object is not output. This object is returned by most procedures and special forms which the Scheme standard defines as returning an unspecified value (e.g. `write`, `set!`, `define`).

Here is a sample interaction with `gsi`:

```
% gsi
Gambit Version 4.0 beta 7

> (define (fact n) (if (< n 2) 1 (* n (fact (- n 1)))))
> (map fact '(1 2 3 4 5 6))
(1 2 6 24 120 720)
> (values (fact 10) (fact 40))
3628800
```

```
8159152832478977343456112695961158942720000000000
> ,q
```

What happens when errors occur is explained in ⟨undefined⟩ [Debugging], page ⟨undefined⟩.

2.2 Batch mode

In batch mode the command line arguments denote files to be loaded, REPL interactions to start ('`-`' option), and expressions to be evaluated ('`-e`' option). Note that '`-`' and '`-e`' options can be interspersed with the files on the command line and can occur multiple times. The interpreter processes the command line arguments from left to right, loading files with the `load` procedure and evaluating expressions with the `eval` procedure in the global interaction environment. After this processing the interpreter exits.

Files can have no extension, or the extension '`.scm`' or '`.six`' or '`.on`' where *n* is a positive integer that acts as a version number (the '`.on`' extension is used for object files produced by `gsc`). When the file name has no extension the `load` procedure first attempts to load the file with no extension as a Scheme source file. If that file doesn't exist it completes the file name with a '`.on`' extension with the highest consecutive version number starting with 1, and loads that file as an object file. If that file doesn't exist the file extension '`.scm`' and '`.six`' will be tried in that order.

If the extension of the file loaded is '`.scm`' the content of the file will be parsed using the normal Scheme prefix syntax. If the extension of the file loaded is '`.six`' the content of the file will be parsed using the Scheme infix syntax extension (see ⟨undefined⟩ [Scheme infix syntax extension], page ⟨undefined⟩).

The ports associated with '`(current-input-port)`', '`(current-output-port)`' and '`(current-error-port)`' initially refer respectively to the standard input ('`stdin`'), standard output ('`stdout`') and the standard error ('`stderr`') of the interpreter. This is true even in REPLs started with the '`-`' option. The usual interaction channel (console or IDE's console window) is still used to read expressions and commands and to display results. This makes it possible to use REPLs to debug programs which read the standard input and write to the standard output, even when these have been redirected.

Here is a sample use of the interpreter in batch mode, under UNIX:

```
% cat m1.scm
(display "hello") (newline)
% cat m2.six
display("world"); newline();
% gsi -e "(display 1)" m1 -e "(display 2)" m2 -e "(display 3)"
1hello
2world
3
```

2.3 Customization

There are two ways to customize the interpreter. When the interpreter starts off it tries to execute a '`(load "~~/gambcext")`' (for an explanation of how file names are interpreted

see ⟨undefined⟩ [I/O and ports], page ⟨undefined⟩). An error is not signaled if the file does not exist. Interpreter extensions and patches that are meant to apply to all users and all modes should go in that file.

Extensions which are meant to apply to a single user or to a specific working directory are best placed in the *initialization file*, which is a file containing Scheme code. In all modes, the interpreter first tries to locate the initialization file by searching the following locations: '`gambcini`' and '`~/gambcini`' (with no extension, a '`.scm`' extension, and a '`.six`' extension in that order). The first file that is found is examined as though the expression `(include` *`initialization-file`*`)` had been entered at the read-eval-print loop where *initialization-file* is the file that was found. Note that by using an `include` the macros defined in the initialization file will be visible from the read-eval-print loop (this would not have been the case if `load` had been used). The initialization file is not searched for or examined if the '`-f`' option is specified.

2.4 Process exit status

The status is zero when the interpreter exits normally and is nonzero when the interpreter exits due to an error. Specifically, here is the meaning of each possible exit status:

`0` The execution of the primordial thread (i.e. the main thread) did not encounter any error. It is however possible that other threads terminated abnormally (by default threads other than the primordial thread terminate silently when they raise an exception that is not handled).

`64` The runtime options or the environment variable '`GAMBCOPT`' contained a syntax error or were invalid.

`70` Change this: There was a problem initializing the runtime system.

`71` There was a problem initializing the runtime system, for example insufficient memory to allocate critical tables.

For example, if the shell is `sh`:

```
% gsi nonexistent.scm
*** ERROR IN ##main -- No such file or directory
(load "nonexistent.scm")
% echo $?
1
```

2.5 Scheme scripts

Gambit's `load` procedure treats specially files that begin with the two characters '`#!`' and '`@;`'. Such files are called *script files*. In addition to indicating that the file is a script, the first line provides information about the source code language to be used by the `load` procedure. After the two characters '`#!`' and '`@;`' the system will search for the first substring matching one of the following language specifying tokens:

`scheme-r4rs`
R4RS language with prefix syntax, case insensitivity, keyword syntax not supported

`scheme-r5rs`
: R5RS language with prefix syntax, case insensitivity, keyword syntax not supported

`scheme-ieee-1178-1990`
: IEEE 1178-1990 language with prefix syntax, case insensitivity, keyword syntax not supported

`scheme-srfi-0`
: R5RS language with prefix syntax and SRFI 0 support (i.e. `cond-expand` special form), case insensitivity, keyword syntax not supported

`gsi-script`
: Full Gambit Scheme language with prefix syntax, case sensitivity, keyword syntax supported

`six-script`
: Full Gambit Scheme language with infix syntax, case sensitivity, keyword syntax supported

If a language specifying token is not found, `load` will use the same language as a nonscript file (i.e. it uses the file extension and runtime system options to determine the language).

After processing the first line, `load` will read the rest of the file and then execute it. If this file is being loaded because it is an argument on the interpreter's command line, the interpreter will

- Setup the `command-line` procedure so that it returns a list containing the expanded file name of the script file and the arguments following the script file on the command line. This is done before the script is executed. The expanded file name of the script file can be used to determine the directory that contains the script (i.e. `(path-directory (car (command-line)))`).
- After the script is loaded the procedure `main` is called with the command-line arguments. The way this is done depends on the language specifying token. For `scheme-r4rs`, `scheme-r5rs`, `scheme-ieee-1178-1990`, and `scheme-srfi-0`, the `main` procedure is called with the equivalent of `(main (cdr (command-line)))` and `main` is expected to return a process exit status code in the range 0 to 255. This conforms to SRFI 22. For `gsi-script` and `six-script` the `main` procedure is called with the equivalent of `(apply main (cdr (command-line)))` and the process exit status code is 0 (`main`'s result is ignored). The Gambit-C system has a predefined `main` procedure which accepts any number of arguments and returns 0, so it is perfectly valid for a script to not define `main` and to do all its processing with top-level expressions.
- When `main` returns, the interpreter exits. The command-line arguments after a script file are consequently not processed.

2.5.1 Scripts under UNIX and Mac OS X

Under UNIX and Mac OS X, the Gambit-C installation process creates the executable '`gsi`' and also the executables '`gsi-script`', '`six-script`', '`scheme-r5rs`', '`scheme-srfi-0`', etc as links to '`gsi`'. A Scheme script need only start with the name

of the desired Scheme language variant prefixed with '`#!`' and the directory where the Gambit-C executables are stored. This script should be made executable by setting the execute permission bits (with a '`chmod +x` *`script`*'. Here is an example of a script which lists on standard output the files in the current directory:

```
#!/usr/local/Gambit-C/bin/gsi-script
(for-each pretty-print (directory-files (current-directory)))
```

Here is another UNIX script, using the Scheme infix syntax extension, which takes a single integer argument and prints on standard output the numbers from 1 to that integer:

```
#!/usr/local/Gambit-C/bin/six-script

void main (obj n_str)
{
  int n = \string->number(n_str);
  for (int i=1; i<=n; i++)
    \pretty-print(i);
}
```

For maximal portability it is a good idea to start scripts indirectly through the '`/usr/bin/env`' program, so that the executable of the interpreter will be searched in the user's '`PATH`'. This is what SRFI 22 recommends. For example here is a script that mimics the UNIX '`cat`' utility for text files:

```
#!/usr/bin/env gsi-script

(define (display-file filename)
  (display (call-with-input-file filename
             (lambda (port)
               (read-line port #f)))))

(for-each display-file (cdr (command-line)))
```

2.5.2 Scripts under Microsoft Windows

Under Microsoft Windows, the Gambit-C installation process creates the executable '`gsi.exe`' and also the batch files '`gsi-script.bat`', '`six-script.bat`', '`scheme-r5rs.bat`', '`scheme-srfi-0.bat`', etc which simply invoke '`gsi.exe`' with the same command line arguments. A Scheme script need only start with the name of the desired Scheme language variant prefixed with '`@;`'. A UNIX script can be converted to a Microsoft Windows script simply by changing the first line and storing the script in a file whose name has a '`.bat`' or '`.cmd`' extension:

```
@;gsi-script %~f0 %*
(display "files:\n")
(pretty-print (directory-files (current-directory)))
```

Note that Microsoft Windows always searches executables in the user's '`PATH`', so there is no need for an indirection such as the UNIX '`/usr/bin/env`'. However the first line must end with '`%~f0 %*`' to pass the expanded filename of the script and command line arguments to the interpreter.

3 The Gambit Scheme compiler

Synopsis:

```
gsc [-:runtimeoption,...] [-i] [-f] [-e expressions]
    [-prelude expressions] [-postlude expressions]
    [-dynamic] [-cc-options options] [-ld-options options]
    [-warnings] [-verbose] [-report] [-expansion]
    [-gvm] [-debug] [-track-scheme]
    [-o output] [-c] [-flat] [-l base] [file...]
```

3.1 Interactive mode

When no command line argument is present other than options the compiler behaves like the interpreter in interactive mode. The only difference with the interpreter is that the compilation related procedures listed in this chapter are also available (i.e. `compile-file`, `compile-file-to-c`, etc).

3.2 Customization

Just like the interpreter, the compiler will examine the initialization file unless the '`-f`' option is specified.

3.3 Batch mode

In batch mode `gsc` takes a set of file names (either with '`.scm`', '`.six`', '`.c`', or no extension) on the command line and compiles each Scheme source file into a C file. File names with no extension are taken to be Scheme source files and a '`.scm`' extension is automatically appended to the file name. For each Scheme source file '*`file`*`.scm`' and '*`file`*`.six`', the C file '*`file`*`.c`' stripped of its directory will be produced (i.e. the C file is created in the current working directory).

The C files produced by the compiler serve two purposes. They will be processed by a C compiler to generate object files, and they also contain information to be read by Gambit's linker to generate a *link file*. The link file is a C file that collects various linking information for a group of modules, such as the set of all symbols and global variables used by the modules. The linker is automatically invoked unless the '`-c`' or '`-dynamic`' options appear on the command line.

Compiler options must be specified before the first file name and after the '`-:`' runtime option (see ⟨undefined⟩ [Runtime options], page ⟨undefined⟩). If present, the '`-f`' and '`-i`' compiler options must come first. The available options are:

`-i` Force interpreter mode.

`-f` Do not examine the initialization file.

`-e` *`expressions`*
: Evaluate expressions in the interaction environment.

`-prelude` *`expressions`*
: Add expressions to the top of the source code being compiled.

`-postlude` *`expressions`*
: Add expressions to the bottom of the source code being compiled.

`-cc-options` *`options`*
: Add options to the command that invokes the C compiler.

`-ld-options` *`options`*
: Add options to the command that invokes the C linker.

`-warnings`
: Display warnings.

`-verbose`
: Display a trace of the compiler's activity.

`-report` Display a global variable usage report.

`-expansion`
: Display the source code after expansion.

`-gvm` Generate a listing of the GVM code.

`-debug` Include debugging information in the code generated.

`-track-scheme`
: Generate '`#line`' directives referring back to the Scheme code.

`-o` *`output`*
: Set name of output file.

`-c` Only compile Scheme source files to C (no link file generated).

`-dynamic`
: Only compile Scheme source files to dynamically loadable object files (no link file generated).

`-flat` Generate a flat link file instead of an incremental link file.

`-l` *`base`* Specify the link file of the base library to use for the link.

The '`-i`' option forces the compiler to process the remaining command line arguments like the interpreter.

The '`-e`' option evaluates the specified expressions in the interaction environment.

The '`-prelude`' option adds the specified expressions to the top of the source code being compiled. The main use of this option is to supply declarations on the command line. For example the following invocation of the compiler will compile the file '`bench.scm`' in unsafe mode:

```
% gsc -prelude "(declare (not safe))" bench.scm
```

The '`-postlude`' option adds the specified expressions to the bottom of the source code being compiled. The main use of this option is to supply the expression that will start the execution of the program. For example:

```
% gsc -postlude "(start-bench)" bench.scm
```

The '`-cc-options`' option is only meaningful when the '`-dynamic`' option is also used. The '`-cc-options`' option adds the specified options to the command that invokes

the C compiler. The main use of this option is to specify the include path, some symbols to define or undefine, the optimization level, or any C compiler option that is different from the default. For example:

```
% gsc -dynamic -cc-options "-U___SINGLE_HOST -O2 -I src/include" bench.scm
```

The '`-ld-options`' option is only meaningful when the '`-dynamic`' option is also used. The '`-ld-options`' option adds the specified options to the command that invokes the C linker. The main use of this option is to specify additional object files or libraries that need to be linked, or any C linker option that is different from the default (such as the library search path and flags to select between static and dynamic linking). For example:

```
% gsc -dynamic -ld-options "-L /usr/X11R6/lib -lX11 -static" bench.scm
```

The '`-warnings`' option displays on standard output all warnings that the compiler may have.

The '`-verbose`' option displays on standard output a trace of the compiler's activity.

The '`-report`' option displays on standard output a global variable usage report. Each global variable used in the program is listed with 4 flags that indicate if the global variable is defined, referenced, mutated and called.

The '`-expansion`' option displays on standard output the source code after expansion and inlining by the front end.

The '`-gvm`' option generates a listing of the intermediate code for the "Gambit Virtual Machine" (GVM) of each Scheme file on '*`file`*`.gvm`'.

The '`-debug`' option causes debugging information to be saved in the code generated. With this option run time error messages indicate the source code and its location, the backtraces are more precise, and `pp` will display the source code of compiled procedures. The debugging information is large (the size of the object file is typically 4 times bigger).

The '`-track-scheme`' options causes the generation of '`#line`' directives that refer back to the Scheme source code. This allows the use of a C debugger to debug Scheme code.

The '`-o`' option sets the name of the output file generated by the compiler. If a link file is being generated the name specified is that of the link file. Otherwise the name specified is that of the C file (this option is ignored if the compiler is generating more than one output file or is generating a dynamically loadable object file).

If the '`-c`' and '`-dynamic`' options do not appear on the command line, the Gambit linker is invoked to generate the link file from the set of C files specified on the command line or produced by the Gambit compiler. Unless the name is specified explicitly with the '`-o`' option, the link file is named '*`last`*`_.c`', where '*`last`*`.c`' is the last file in the set of C files. When the '`-c`' option is specified, the Scheme source files are compiled to C files. When the '`-dynamic`' option is specified, the Scheme source files are compiled to dynamically loadable object files ('*`.on`*' extension).

The '`-flat`' option is only meaningful if a link file is being generated (i.e. the '`-c`' and '`-dynamic`' options are absent). The '`-flat`' option directs the Gambit linker to generate a flat link file. By default, the linker generates an incremental link file (see the next section for a description of the two types of link files).

The '`-l`' option is only meaningful if an incremental link file is being generated (i.e. the '`-c`', '`-dynamic`' and '`-flat`' options are absent). The '`-l`' option specifies the link file

(without the '`.c`' extension) of the base library to use for the incremental link. By default the link file of the Gambit runtime library is used (i.e. '`~~/lib/_gambc.c`').

3.4 Link files

Gambit can be used to create applications and libraries of Scheme modules. This section explains the steps required to do so and the role played by the link files.

In general, an application is composed of a set of Scheme modules and C modules. Some of the modules are part of the Gambit runtime library and the other modules are supplied by the user. When the application is started it must setup various global tables (including the symbol table and the global variable table) and then sequentially execute the Scheme modules (more or less as if they were being loaded one after another). The information required for this is contained in one or more *link files* generated by the Gambit linker from the C files produced by the Gambit compiler.

The order of execution of the Scheme modules corresponds to the order of the modules on the command line which produced the link file. The order is usually important because most modules define variables and procedures which are used by other modules (for this reason the program's main computation is normally started by the last module).

When a single link file is used to contain the linking information of all the Scheme modules it is called a *flat link file*. Thus an application built with a flat link file contains in its link file both information on the user modules and on the runtime library. This is fine if the application is to be statically linked but is wasteful in a shared-library context because the linking information of the runtime library can't be shared and will be duplicated in all applications (this linking information typically takes hundreds of Kbytes).

Flat link files are mainly useful to bundle multiple Scheme modules to make a runtime library (such as the Gambit runtime library) or to make a single file that can be loaded with the `load` procedure.

An *incremental link file* contains only the linking information that is not already contained in a second link file (the "base" link file). Assuming that a flat link file was produced when the runtime library was linked, an application can be built by linking the user modules with the runtime library's link file, producing an incremental link file. This allows the creation of a shared-library which contains the modules of the runtime library and its flat link file. The application is dynamically linked with this shared-library and only contains the user modules and the incremental link file. For small applications this approach greatly reduces the size of the application because the incremental link file is small. A "hello world" program built this way can be as small as 5 Kbytes. Note that it is perfectly fine to use an incremental link file for statically linked programs (there is very little loss compared to a single flat link file).

Incremental link files may be built from other incremental link files. This allows the creation of shared-libraries which extend the functionality of the Gambit runtime library.

3.4.1 Building an executable program

The simplest way to create an executable program is to call up `gsc` to compile each Scheme module into a C file and create an incremental link file. The C files and the link file must then be compiled with a C compiler and linked (at the object file level) with the Gambit runtime library and possibly other libraries (such as the math library and the dynamic

loading library). Here is for example how a program with three modules (one in C and two in Scheme) can be built:

```
% uname -a
Linux bailey 1.2.13 #2 Wed Aug 28 16:29:41 GMT 1996 i586
% cat m1.c
int power_of_2 (int x) { return 1<<x; }
% cat m2.scm
(c-declare "extern int power_of_2 ();")
(define pow2 (c-lambda (int) int "power_of_2"))
(define (twice x) (cons x x))
% cat m3.scm
(write (map twice (map pow2 '(1 2 3 4)))) (newline)
% gsc -c m2.scm # create m2.c (note: .scm is optional)
% gsc -c m3.scm # create m3.c (note: .scm is optional)
% gsc m2.c m3.c # create the incremental link file m3_.c
% gcc m1.c m2.c m3.c m3_.c -lgambc
% ./a.out
((2 . 2) (4 . 4) (8 . 8) (16 . 16))
```

Alternatively, the three invocations of `gsc` can be replaced by a single invocation:

```
% gsc m2 m3
```

3.4.2 Building a loadable library

To bundle multiple modules into a single file that can be dynamically loaded with the `load` procedure, a flat link file is needed. When compiling the C files and link file generated, the flag '`-D___DYNAMIC`' must be passed to the C compiler. The three modules of the previous example can be bundled in this way:

```
% uname -a
Linux bailey 1.2.13 #2 Wed Aug 28 16:29:41 GMT 1996 i586
% gsc -flat -o foo.c m2 m3
m2:
m3:
*** WARNING -- "cons" is not defined,
***            referenced in: ("m2.c")
*** WARNING -- "map" is not defined,
***            referenced in: ("m3.c")
*** WARNING -- "newline" is not defined,
***            referenced in: ("m3.c")
*** WARNING -- "write" is not defined,
***            referenced in: ("m3.c")
% gcc -shared -fPIC -D___DYNAMIC m1.c m2.c m3.c foo.c -o foo.o1
% gsi
Gambit Version 4.0 beta 7

> (load "foo")
((2 . 2) (4 . 4) (8 . 8) (16 . 16))
"/users/feeley/foo.o1"
```

```
> ,q
```

The warnings indicate that there are no definitions (`define`s or `set!`s) of the variables `cons`, `map`, `newline` and `write` in the set of modules being linked. Before '`foo.o1`' is loaded, these variables will have to be bound; either implicitly (by the runtime library) or explicitly.

Here is a more complex example, under Solaris, which shows how to build a loadable library '`mymod.o1`' composed of the files '`m1.scm`', '`m2.scm`' and '`x.c`' that links to system shared libraries (for X-windows):

```
% uname -a
SunOS ungava 5.6 Generic_105181-05 sun4m sparc SUNW,SPARCstation-20
% gsc -flat -o mymod.c m1 m2
m1:
m2:
*** WARNING -- "*" is not defined,
***            referenced in: ("m1.c")
*** WARNING -- "+" is not defined,
***            referenced in: ("m2.c")
*** WARNING -- "display" is not defined,
***            referenced in: ("m2.c" "m1.c")
*** WARNING -- "newline" is not defined,
***            referenced in: ("m2.c" "m1.c")
*** WARNING -- "write" is not defined,
***            referenced in: ("m2.c")
% gcc -fPIC -c -I../lib -D___DYNAMIC mymod.c m1.c m2.c x.c
% /usr/ccs/bin/ld -G -o mymod.o1 mymod.o m1.o m2.o x.o -lX11 -lsocket
% gsi mymod.o1
hello from m1
hello from m2
(f1 10) = 22
% cat m1.scm
(define (f1 x) (* 2 (f2 x)))
(display "hello from m1")
(newline)

(c-declare "#include \"x.h\"")
(define x-initialize (c-lambda (char-string) bool "x_initialize"))
(define x-display-name (c-lambda () char-string "x_display_name"))
(define x-bell (c-lambda (int) void "x_bell"))
% cat m2.scm
(define (f2 x) (+ x 1))
(display "hello from m2")
(newline)

(display "(f1 10) = ")
(write (f1 10))
(newline)
```

```
(x-initialize (x-display-name))
(x-bell 50) ; sound the bell at 50%
% cat x.c
#include <X11/Xlib.h>

static Display *display;

int x_initialize (char *display_name)
{
  display = XOpenDisplay (display_name);
  return display != NULL;
}

char *x_display_name (void)
{
  return XDisplayName (NULL);
}

void x_bell (int volume)
{
  XBell (display, volume);
  XFlush (display);
}
% cat x.h
int x_initialize (char *display_name);
char *x_display_name (void);
void x_bell (int);
```

3.4.3 Building a shared-library

A shared-library can be built using an incremental link file or a flat link file. An incremental link file is normally used when the Gambit runtime library (or some other library) is to be extended with new procedures. A flat link file is mainly useful when building a "primal" runtime library, which is a library (such as the Gambit runtime library) that does not extend another library. When compiling the C files and link file generated, the flags '`-D___LIBRARY`' and '`-D___SHARED`' must be passed to the C compiler. The flag '`-D___PRIMAL`' must also be passed to the C compiler when a primal library is being built.

A shared-library '`mylib.so`' containing the two first modules of the previous example can be built this way:

```
% uname -a
Linux bailey 1.2.13 #2 Wed Aug 28 16:29:41 GMT 1996 i586
% gsc -o mylib.c m2
% gcc -shared -fPIC -D___LIBRARY -D___SHARED m1.c m2.c mylib.c -o mylib.so
```

Note that this shared-library is built using an incremental link file (it extends the Gambit runtime library with the procedures `pow2` and `twice`). This shared-library can in turn be used to build an executable program from the third module of the previous example:

```
% gsc -l mylib m3
% gcc m3.c m3_.c mylib.so -lgambc
% LD_LIBRARY_PATH=.:/usr/local/lib ./a.out
((2 . 2) (4 . 4) (8 . 8) (16 . 16))
```

3.4.4 Other compilation options and flags

The performance of the code can be increased by passing the '`-D___SINGLE_HOST`' flag to the C compiler. This will merge all the procedures of a module into a single C procedure, which reduces the cost of intra-module procedure calls. In addition the '`-O`' option can be passed to the C compiler. For large modules, it will not be practical to specify both '`-O`' and '`-D___SINGLE_HOST`' for typical C compilers because the compile time will be high and the C compiler might even fail to compile the program for lack of memory.

Normally C compilers will not automatically search '`/usr/local/Gambit-C/include`' for header files so the flag '`-I/usr/local/Gambit-C/include`' should be passed to the C compiler. Similarly, C compilers/linkers will not automatically search '`/usr/local/Gambit-C/lib`' for libraries so the flag '`-L/usr/local/Gambit-C/lib`' should be passed to the C compiler/linker. For alternatives see ⟨undefined⟩ [Top], page ⟨undefined⟩.

A variety of flags are needed by some C compilers when compiling a shared-library or a dynamically loadable library. Some of these flags are: '`-shared`', '`-call_shared`', '`-rdynamic`', '`-fpic`', '`-fPIC`', '`-Kpic`', '`-KPIC`', '`-pic`', '`+z`'. Check your compiler's documentation to see which flag you need.

3.5 Procedures and syntax

`(compile-file-to-c` *`file`* [*`options`* [*`output`*]]) [procedure]

file must be a string naming an existing file containing Scheme source code. The extension can be omitted from *file* if the Scheme file has a '`.scm`' extension. This procedure compiles the source file into a file containing C code. By default, this file is named after *file* with the extension replaced with '`.c`'. However, if *output* is supplied the file is named '*`output`*'.

Compilation options are given as a list of symbols after the file name. Any combination of the following options can be used: '`verbose`', '`report`', '`expansion`', '`gvm`', and '`debug`'.

Note that this procedure is only available in `gsc`.

`(compile-file` *`file`* [*`options`*]) [procedure]

The arguments of `compile-file` are the same as the first two arguments of `compile-file-to-c`. The `compile-file` procedure compiles the source file into an object file by first generating a C file and then compiling it with the C compiler. The object file is named after *file* with the extension replaced with '`.o`*n*', where *n* is a positive integer that acts as a version number. The next available version number is generated automatically by `compile-file`. Object files can be loaded dynamically by using the `load` procedure. The '`.o`*n*' extension can be specified (to select a particular version) or omitted (to load the highest numbered version). Versions which are no longer needed must be deleted manually and the remaining version(s) must be renamed to start with extension '`.o1`'.

Note that this procedure is only available in `gsc` and that it is only useful on operating systems that support dynamic loading.

`(link-incremental` *`module-list`* [*`output`* [*`base`*]]`)` [procedure]

The first argument must be a non empty list of strings naming Scheme modules to link (extensions must be omitted). The remaining optional arguments must be strings. An incremental link file is generated for the modules specified in *module-list*. By default the link file generated is named '*`last`*`_.c`', where *last* is the name of the last module. However, if *output* is supplied the link file is named '*`output`*'. The base link file is specified by the *base* parameter. By default the base link file is the Gambit runtime library link file '`~~/_gambc.c`'. However, if *base* is supplied the base link file is named '*`base`*`.c`'.

Note that this procedure is only available in `gsc`.

The following example shows how to build the executable program '`hello`' which contains the two Scheme modules '`m1.scm`' and '`m2.scm`'.

```
% uname -a
Linux bailey 1.2.13 #2 Wed Aug 28 16:29:41 GMT 1996 i586
% cat m1.scm
(display "hello") (newline)
% cat m2.scm
(display "world") (newline)
% gsc
Gambit Version 4.0 beta 7

> (compile-file-to-c "m1")
#t
> (compile-file-to-c "m2")
#t
> (link-incremental '("m1" "m2") "hello.c")
> ,q
% gcc m1.c m2.c hello.c -lgambc -o hello
% ./hello
hello
world
```

`(link-flat` *`module-list`* [*`output`*]`)` [procedure]

The first argument must be a non empty list of strings. The first string must be the name of a Scheme module or the name of a link file and the remaining strings must name Scheme modules (in all cases extensions must be omitted). The second argument must be a string, if it is supplied. A flat link file is generated for the modules specified in *module-list*. By default the link file generated is named '*`last`*`_.c`', where *last* is the name of the last module. However, if *output* is supplied the link file is named '*`output`*'.

Note that this procedure is only available in `gsc`.

The following example shows how to build the dynamically loadable Scheme library '`lib.o1`' which contains the two Scheme modules '`m1.scm`' and '`m2.scm`'.

```
% uname -a
Linux bailey 1.2.13 #2 Wed Aug 28 16:29:41 GMT 1996 i586
% cat m1.scm
(define (f x) (g (* x x)))
% cat m2.scm
(define (g y) (+ n y))
% gsc
Gambit Version 4.0 beta 7

> (compile-file-to-c "m1")
#t
> (compile-file-to-c "m2")
#t
> (link-flat '("m1" "m2") "lib.c")
*** WARNING -- "*" is not defined,
***            referenced in: ("m1.c")
*** WARNING -- "+" is not defined,
***            referenced in: ("m2.c")
*** WARNING -- "n" is not defined,
***            referenced in: ("m2.c")
> ,q
% gcc -shared -fPIC -D___DYNAMIC m1.c m2.c lib.c -o lib.o1
% gsc
Gambit Version 4.0 beta 7

> (load "lib")
*** WARNING -- Variable "n" used in module "m2" is undefined
"/users/feeley/lib.o1"
> (define n 10)
> (f 5)
35
> ,q
```

The warnings indicate that there are no definitions (`define`s or `set!`s) of the variables `*`, `+` and `n` in the modules contained in the library. Before the library is used, these variables will have to be bound; either implicitly (by the runtime library) or explicitly.

4 Runtime options for all programs

Both `gsi` and `gsc` as well as executable programs compiled and linked using `gsc` take a '`-:`' option which supplies parameters to the runtime system. This option must appear first on the command line. The colon is followed by a comma separated list of options with no intervening spaces.

The available options are:

`m`*`heapsize`*
: Set minimum heap size in kilobytes.

`h`*`heapsize`*
: Set maximum heap size in kilobytes.

`l`*`livepercent`*
: Set heap occupation after garbage collection.

`s`
: Select standard Scheme mode.

`S`
: Select Gambit Scheme mode.

`d`[*`OPT...`*]
: Set debugging options.

`t`[*`OPT...`*]
: Set terminal options.

`c`*`encoding`*
: Set default character encoding for I/O.

`=`*`directory`*
: Override the Gambit installation directory.

`+`*`argument`*
: Add *argument* to the command line before other arguments.

The '`m`' option specifies the minimum size of the heap. The '`m`' is immediately followed by an integer indicating the number of kilobytes of memory. The heap will not shrink lower than this size. By default, the minimum size is 0.

The '`h`' option specifies the maximum size of the heap. The '`h`' is immediately followed by an integer indicating the number of kilobytes of memory. The heap will not grow larger than this size. By default, there is no limit (i.e. the heap will grow until the virtual memory is exhausted).

The '`l`' option specifies the percentage of the heap that will be occupied with live objects after the heap is resized at the end of a garbage collection. The '`l`' is immediately followed by an integer between 1 and 100 inclusively indicating the desired percentage. The garbage collector resizes the heap to reach this percentage occupation. By default, the percentage is 50.

The '`s`' option selects standard Scheme mode. In this mode the reader is case insensitive and keywords are not recognized. The '`S`' option selects Gambit Scheme mode (the reader is case sensitive and recognizes keywords which end with a colon). By default Gambit Scheme mode is used.

The ‘d’ option sets various debugging options. The letter ‘d’ is followed by a sequence of letters indicating suboptions.

p	Uncaught exceptions will be treated as “errors” in the primordial thread only.
a	Uncaught exceptions will be treated as “errors” in all threads.
r	When an “error” occurs a new REPL will be started.
s	When an “error” occurs a new REPL will be started. Moreover the program starts in single-stepping mode.
q	When an “error” occurs the program will terminate with a nonzero exit status.
i	The REPL interaction channel will be the IDE REPL window (if the IDE is available).
c	The REPL interaction channel will be the console.
-	The REPL interaction channel will be standard input and standard output.
level	The verbosity level is set to *level* (a digit from 0 to 9). At level 0 the runtime system will not display error messages and warnings.

The default debugging options are equivalent to `-:dpqi1` (i.e. an uncaught exception in the primordial thread terminates the program after displaying an error message). If the letter ‘d’ is not followed by suboptions, it is equivalent to `-:dpri1` (i.e. a new REPL is started only when an uncaught exception occurs in the primordial thread).

The ‘t’ option sets vaious terminal options. This is not fully implemented yet.

The ‘c’ option selects the default character encoding for I/O. This is not fully implemented yet.

The ‘=’ option overrides the setting of the Gambit installation directory.

The ‘+’ option adds the text that follows to the command line before other arguments.

If the environment variable ‘GAMBCOPT’ is defined, the runtime system will take its options from that environment variable. A ‘-:’ option can be used to override some or all of the runtime system options. For example:

```
% GAMBCOPT=d0,=~/my-gambit2
% export GAMBCOPT
% gsi -e '(pretty-print (path-expand "~~")) (/ 1 0)'
"/u/feeley/my-gambit2/"
% echo $?
1
% gsi -:d1 -e '(pretty-print (path-expand "~~")) (/ 1 0)'
"/u/feeley/my-gambit2/"
*** ERROR IN string@1.25 -- Divide by zero
(/ 1 0)
```

5 Debugging

The evaluation of an expression may stop before it is completed for the following reasons:

a. An evaluation error has occured, such as attempting to divide by zero.
b. The user has interrupted the evaluation (usually by typing ⟨^C⟩).
c. A breakpoint has been reached or `(step)` was evaluated.
d. Single-stepping mode is enabled.

When an evaluation stops, a message is displayed indicating the reason and location where the evaluation was stopped. The location information includes, if known, the name of the procedure where the evaluation was stopped and the source code location in the format '*stream*`@`*line*`.`*column*', where *stream* is either a string naming a file or a symbol within parentheses, such as '`(console)`'.

A *nested REPL* is then initiated in the context of the point of execution where the evaluation was stopped. The nested REPL's continuation and evaluation environment are the same as the point where the evaluation was stopped. For example when evaluating the expression '`(let ((y (- 1 1))) (* (/ x y) 2))`', a "divide by zero" error is reported and the nested REPL's continuation is the one that takes the result and multiplies it by two. The REPL's lexical environment includes the lexical variable '`y`'. This allows the inspection of the evaluation context (i.e. the lexical and dynamic environments and continuation), which is particularly useful to determine the exact location and cause of an error.

The prompt of nested REPLs includes the nesting level; '`1>`' is the prompt at the first nesting level, '`2>`' at the second nesting level, and so on. An end of file (usually ⟨^D⟩) will cause the current REPL to be terminated and the enclosing REPL (one nesting level less) to be resumed.

At any time the user can examine the frames in the REPL's continuation, which is useful to determine which chain of procedure calls lead to an error. A backtrace that lists the chain of active continuation frames in the REPL's continuation can be obtained with the '`,b`' command. The frames are numbered from 0, that is frame 0 is the most recent frame of the continuation where execution stopped, frame 1 is the parent frame of frame 0, and so on. It is also possible to move the REPL to a specific parent continuation (i.e. a specific frame of the continuation where execution stopped) with the '`,+`', '`,-`' and '`,`*n*' commands (where *n* is the frame number). When the frame number of the frame being examined is not zero, it is shown in the prompt after the nesting level, for example '`1\5>`' is the prompt when the REPL nesting level is 1 and the frame number is 5.

Expressions entered at a nested REPL are evaluated in the environment (both lexical and dynamic) of the continuation frame currently being examined if that frame was created by interpreted Scheme code. If the frame was created by compiled Scheme code then expressions get evaluated in the global interaction environment. This feature may be used in interpreted code to fetch the value of a variable in the current frame or to change its value with `set!`. Note that some special forms (`define` in particular) can only be evaluated in the global interaction environment.

In addition to expressions, the REPL accepts the following special "comma" commands:

`,?` Give a summary of the REPL commands.

`,q` Quit the program (i.e. terminate abruptly).

`,t` Return to the outermost REPL, also known as the "top-level REPL".

`,d` Leave the current REPL and resume the enclosing REPL. This command does nothing in the top-level REPL.

`,(c` *expr*`)`
Leave the current REPL and continue the computation that initiated the REPL with a specific value. This command can only be used to continue a computation that signaled an error. The expression *expr* is evaluated in the current context and the resulting value is returned as the value of the expression which signaled the error. For example, if the evaluation of the expression '`(* (/ x y) 2)`' signaled an error because '`y`' is zero, then in the nested REPL a '`,(c (+ 4 y))`' will resume the computation of '`(* (/ x y) 2)`' as though the value of '`(/ x y)`' was 4. This command must be used carefully because the context where the error occured may rely on the result being of a particular type. For instance a '`,(c #f)`' in the previous example will cause '`*`' to signal a type error (this problem is the most troublesome when debugging Scheme code that was compiled with type checking turned off so be careful).

`,c` Leave the current REPL and continue the computation that initiated the REPL. This command can only be used to continue a computation that was stopped due to a user interrupt, breakpoint or a single-step.

`,s` Leave the current REPL and continue the computation that initiated the REPL in single-stepping mode. The computation will perform an evaluation step (as defined by `step-level-set!`) and then stop, causing a nested REPL to be entered. Just before the evaluation step is performed, a line is displayed (in the same format as `trace`) which indicates the expression that is being evaluated. If the evaluation step produces a result, the result is also displayed on another line. A nested REPL is then entered after displaying a message which describes the next step of the computation. This command can only be used to continue a computation that was stopped due to a user interrupt, breakpoint or a single-step.

`,l` This command is similar to '`,s`' except that it "leaps" over procedure calls, that is procedure calls are treated like a single step. Single-stepping mode will resume when the procedure call returns, or if and when the execution of the called procedure encounters a breakpoint.

`,`*n* Move to frame number *n* of the continuation. After changing the current frame, a one-line summary of the frame is displayed as if the '`,y`' command was entered.

`,+` Move to the next frame in the chain of continuation frames (i.e. towards older continuation frames). After changing the current frame, a one-line summary of the frame is displayed as if the '`,y`' command was entered.

`,-` Move to the previous frame in the chain of continuation frames (i.e. towards more recently created continuation frames). After changing the current frame, a one-line summary of the frame is displayed as if the '`,y`' command was entered.

`,y` Display a one-line summary of the current frame. The information is displayed in four fields. The first field is the frame number. The second field is the procedure that created the frame or '`(interaction)`' if the frame was created by an expression entered at the REPL. The remaining fields describe the subproblem associated with the frame, that is the expression whose value is being computed. The third field is the location of the subproblem's source code and the fourth field is a reproduction of the source code, possibly truncated to fit on the line. The last two fields may be missing if that information is not available. In particular, the third field is missing when the frame was created by a user call to the '`eval`' procedure, and the last two fields are missing when the frame was created by a compiled procedure not compiled with the '`-debug`' option.

`,b` Display a backtrace summarizing each frame in the chain of continuation frames starting with the current frame. For each frame, the same information as for the '`,y`' command is displayed (except that location information is displayed in the format '*`stream@line:column`*'). If there are more that 15 frames in the chain of continuation frames, some of the middle frames will be omitted.

`,i` Pretty print the procedure that created the current frame or '`(interaction)`' if the frame was created by an expression entered at the REPL. Compiled procedures will only be pretty printed if compiled with the '`-debug`' option.

`,e` Display the environment which is accessible from the current frame. Both the lexical and dynamic environments are displayed. However, only non-global lexical variables are displayed and only if the frame was created by interpreted code or code compiled with the '`-debug`' option. Due to space safety considerations and compiler optimizations, some of the lexical variable bindings may be missing. Lexical variable bindings are displayed using the format '*`variable`* `=` *`expression`*' and dynamically-bound parameter bindings are displayed using the format '`(`*`parameter`*`) =` *`expression`*'. Note that *expression* can be a self-evaluating expression (number, string, boolean, character, ...), a quoted expression, a lambda expression or a global variable (the last two cases, which are only used when the value of the variable or parameter is a procedure, simplifies the debugging of higher-order procedures). A *parameter* can be a quoted expression or a global variable. Lexical bindings are displayed in inverse binding order (most deeply nested first) and shadowed variables are included in the list.

Here is a sample interaction with `gsi`:

```
% gsi
Gambit Version 4.0 beta 7

> (define (invsqr x) (/ 1 (expt x 2)))
> (define (mymap fn lst)
    (define (mm in)
      (if (null? in)
          '()
          (cons (fn (car in)) (mm (cdr in)))))
    (mm lst))
> (mymap invsqr '(5 2 hello 9 1))
```

```
*** ERROR IN invsqr, (console)@1.25 -- (Argument 1) NUMBER expected
(expt 'hello 2)
1> ,i
#<procedure 3 invsqr> =
(lambda (x) (/ 1 (expt x 2)))
1> ,e
x = 'hello
(current-exception-handler) = primordial-exception-handler
(current-input-port) = '#<input-port 4 (stdin)>
(current-output-port) = '#<output-port 5 (stdout)>
(current-directory) = "/Users/feeley/gambc40b3/"
1> ,b
0  invsqr                    (console)@1:25          (expt x 2)
1  #<procedure 2>            (console)@6:17          (fn (car in))
2  #<procedure 2>            (console)@6:31          (mm (cdr in))
3  #<procedure 2>            (console)@6:31          (mm (cdr in))
4  (interaction)             (console)@8:1           (mymap invsqr '(5 2 he
1> ,+
1  #<procedure 2>            (console)@6.17          (fn (car in))
1\1> (pp #2)
(lambda (in) (if (null? in) '() (cons (fn (car in)) (mm (cdr in)))))
1\1> ,e
in = '(hello 9 1)
mm = (lambda (in) (if (null? in) '() (cons (fn (car in)) (mm (cdr in)))))
fn = invsqr
lst = '(5 2 hello 9 1)
(current-exception-handler) = primordial-exception-handler
(current-input-port) = '#<input-port 4 (stdin)>
(current-output-port) = '#<output-port 5 (stdout)>
(current-directory) = "/Users/feeley/gambc40b3/"
1\1> fn
#<procedure 3 invsqr>
1\1> (pp fn)
(lambda (x) (/ 1 (expt x 2)))
1\1> ,+
2  #<procedure 2>            (console)@6.31          (mm (cdr in))
1\2> ,e
in = '(2 hello 9 1)
mm = (lambda (in) (if (null? in) '() (cons (fn (car in)) (mm (cdr in)))))
fn = invsqr
lst = '(5 2 hello 9 1)
(current-exception-handler) = primordial-exception-handler
(current-input-port) = '#<input-port 4 (stdin)>
(current-output-port) = '#<output-port 5 (stdout)>
(current-directory) = "/Users/feeley/gambc40b3/"
1\2> ,(c (list 3 4 5))
(1/25 1/4 3 4 5)
```

```
> ,q
```

5.1 Procedures and syntax

`(trace` *proc...*`)` [procedure]
`(untrace` *proc...*`)` [procedure]

`trace` starts tracing calls to the specified procedures. When a traced procedure is called, a line containing the procedure and its arguments is displayed (using the procedure call expression syntax). The line is indented with a sequence of vertical bars which indicate the nesting depth of the procedure's continuation. After the vertical bars is a greater-than sign which indicates that the evaluation of the call is starting.

When a traced procedure returns a result, it is displayed with the same indentation as the call but without the greater-than sign. This makes it easy to match calls and results (the result of a given call is the value at the same indentation as the greater-than sign). If a traced procedure P1 performs a tail call to a traced procedure P2, then P2 will use the same indentation as P1. This makes it easy to spot tail calls. The special handling for tail calls is needed to preserve the space complexity of the program (i.e. tail calls are implemented as required by Scheme even when they involve traced procedures).

`untrace` stops tracing calls to the specified procedures. With no argument, `trace` returns the list of procedures currently being traced. The void object is returned by `trace` if it is passed one or more arguments. With no argument `untrace` stops all tracing and returns the void object. A compiled procedure may be traced but only if it is bound to a global variable.

For example:

```
> (define (fact n) (if (< n 2) 1 (* n (fact (- n 1)))))
> (trace fact)
> (fact 5)
| > (fact 5)
| | > (fact 4)
| | | > (fact 3)
| | | | > (fact 2)
| | | | | > (fact 1)
| | | | | 1
| | | | 2
| | | 6
| | 24
| 120
120
> (trace -)
*** WARNING -- Rebinding global variable "-" to an interpreted procedur
> (define (fact-iter n r) (if (< n 2) r (fact-iter (- n 1) (* n r))))
> (trace fact-iter)
> (fact-iter 5 1)
| > (fact-iter 5 1)
| | > (- 5 1)
```

```
| | 4
| > (fact-iter 4 5)
| | > (- 4 1)
| | 3
| > (fact-iter 3 20)
| | > (- 3 1)
| | 2
| > (fact-iter 2 60)
| | > (- 2 1)
| | 1
| > (fact-iter 1 120)
| 120
120
> (trace)
(#<procedure fact-iter> #<procedure -> #<procedure fact>)
> (untrace)
> (fact 5)
120
```

`(step)` [procedure]
`(step-level-set!` *`level`*`)` [procedure]

The procedure `step` enables single-stepping mode. After the call to `step` the computation will stop just before the interpreter executes the next evaluation step (as defined by `step-level-set!`). A nested REPL is then started. Note that because single-stepping is stopped by the REPL whenever the prompt is displayed it is pointless to enter `(step)` by itself. On the other hand entering `(begin (step)` *`expr`*`)` will evaluate *expr* in single-stepping mode.

The procedure `step-level-set!` sets the stepping level which determines the granularity of the evaluation steps when single-stepping is enabled. The stepping level *level* must be an exact integer in the range 0 to 7. At a level of 0, the interpreter ignores single-stepping mode. At higher levels the interpreter stops the computation just before it performs the following operations, depending on the stepping level:

1. procedure call
2. `delay` special form and operations at lower levels
3. `lambda` special form and operations at lower levels
4. `define` special form and operations at lower levels
5. `set!` special form and operations at lower levels
6. variable reference and operations at lower levels
7. constant reference and operations at lower levels

The default stepping level is 7.

For example:

```
> (define (fact n) (if (< n 2) 1 (* n (fact (- n 1)))))
> (step-level-set! 1)
> (begin (step) (fact 5))
*** STOPPED IN (stdin)@3.15
```

```
1> ,s
| > (fact 5)
*** STOPPED IN fact, (stdin)@1.22
1> ,s
| | > (< n 2)
| | #f
*** STOPPED IN fact, (stdin)@1.43
1> ,s
| | > (- n 1)
| | 4
*** STOPPED IN fact, (stdin)@1.37
1> ,s
| | > (fact (- n 1))
*** STOPPED IN fact, (stdin)@1.22
1> ,s
| | | > (< n 2)
| | | #f
*** STOPPED IN fact, (stdin)@1.43
1> ,s
| | | > (- n 1)
| | | 3
*** STOPPED IN fact, (stdin)@1.37
1> ,l
| | | > (fact (- n 1))
| | | 6
*** STOPPED IN fact, (stdin)@1.32
1> ,l
| | > (* n (fact (- n 1)))
| | 24
*** STOPPED IN fact, (stdin)@1.32
1> ,l
| > (* n (fact (- n 1)))
| 120
120
```

`(break` *proc...*`)` [procedure]
`(unbreak` *proc...*`)` [procedure]

`break` places a breakpoint on each of the specified procedures. When a procedure is called that has a breakpoint, the interpreter will enable single-stepping mode (as if `step` had been called). This typically causes the computation to stop soon inside the procedure if the stepping level is high enough.

`unbreak` removes the breakpoints on the specified procedures. With no argument, `break` returns the list of procedures currently containing breakpoints. The void object is returned by `break` if it is passed one or more arguments. With no argument `unbreak` removes all the breakpoints and returns the void object. A breakpoint can be placed on a compiled procedure but only if it is bound to a global variable.

For example:

```
> (define (double x) (+ x x))
> (define (triple y) (- (double (double y)) y))
> (define (f z) (* (triple z) 10))
> (break double)
> (break -)
*** WARNING -- Rebinding global variable "-" to an interpreted procedur
> (f 5)
*** STOPPED IN double, (stdin)@1.21
1> ,b
0  double                  (stdin)@1:21            +
1  triple                  (stdin)@2:31            (double y)
2  f                       (stdin)@3:18            (triple z)
3  (interaction)           (stdin)@6:1             (f 5)
4  ##initial-continuation
1> ,e
x = 5
1> ,c
*** STOPPED IN double, (stdin)@1.21
1> ,c
*** STOPPED IN f, (stdin)@3.29
1> ,c
150
> (break)
(#<procedure -> #<procedure double>)
> (unbreak)
> (f 5)
150
```

(proper-tail-calls-set! *proper?*) [procedure]

`proper-tail-calls-set!` sets a flag that controls how the interpreter handles tail calls. When *proper?* is `#f` the interpreter will treat tail calls like nontail calls, that is a new continuation will be created for the call. This setting is useful for debugging, because when a primitive signals an error the location information will point to the call site of the primitive even if this primitive was called with a tail call. The default setting of this flag is `#t`, which means that a tail call will reuse the continuation of the calling function.

The setting of this flag only affects code that is subsequently processed by `load` or `eval`, or entered at the REPL.

(display-environment-set! *display?*) [procedure]

`display-environment-set!` sets a flag that controls the automatic display of the environment by the REPL. If *display?* is true, the environment is displayed by the REPL before the prompt. The default setting is not to display the environment.

(object->serial-number *integer*) [procedure]

(serial-number->object *object*) [procedure]

5.2 Console line editing

to do!

5.3 Emacs interface

Gambit comes with the Emacs package '`gambit.el`' which provides a nice environment for running Gambit from within the Emacs editor. This package filters the standard output of the Gambit process and when it intercepts a location information (in the format '*`stream@line.column`*' where *stream* is either '`(stdin)`' if the expression was obtained from standard input or a string naming a file) it opens a window to highlight the corresponding expression.

To use this package, make sure the file '`gambit.el`' is accessible from your load-path and that the following lines are in your '`.emacs`' file:

```
(autoload 'gambit-inferior-mode "gambit" "Hook Gambit mode into cmuscheme.")
(autoload 'gambit-mode "gambit" "Hook Gambit mode into scheme.")
(add-hook 'inferior-scheme-mode-hook (function gambit-inferior-mode))
(add-hook 'scheme-mode-hook (function gambit-mode))
(setq scheme-program-name "gsi -:t")
```

Alternatively, if you don't mind always loading this package, you can simply add this line to your '`.emacs`' file:

```
(require 'gambit)
```

You can then start an inferior Gambit process by typing '`M-x run-scheme`'. The commands provided in '`cmuscheme`' mode will be available in the Gambit interaction buffer (i.e. '`*scheme*`') and in buffers attached to Scheme source files. Here is a list of the most useful commands (for a complete list type '`C-h m`' in the Gambit interaction buffer):

`C-x C-e` Evaluate the expression which is before the cursor (the expression will be copied to the Gambit interaction buffer).

`C-c C-z` Switch to Gambit interaction buffer.

`C-c C-l` Load a file (file attached to current buffer is default) using `(load` *`file`*`)`.

`C-c C-k` Compile a file (file attached to current buffer is default) using `(compile-file` *`file`*`)`.

The file '`gambit.el`' provides these additional commands:

`C-c c` Continue the computation (same as typing '`,c`' to the REPL).

`C-c s` Step the computation (same as typing '`,s`' to the REPL).

`C-c l` Leap the computation (same as typing '`,l`' to the REPL).

`C-c [` Move to older frame (same as typing '`,+`' to the REPL).

`C-c ]` Move to newer frame (same as typing '`,-`' to the REPL).

`C-c _` Removes the last window that was opened to highlight an expression.

These commands can be shortened to '`M-c`', '`M-s`', '`M-l`', '`M-[`', '`M-]`', and '`M-_`' respectively by adding this line to your '`.emacs`' file:

```
(setq gambit-repl-command-prefix "\e")
```

This is more convenient to type than the two keystroke '`C-c`' based sequences but the purist may not like this because it does not follow normal Emacs conventions.

Here is what a typical '`.emacs`' file will look like:

```
(setq load-path
  (cons "/usr/local/Gambit-C/share/emacs/site-lisp" ; location of gambit.el
        load-path))
(setq scheme-program-name "/tmp/gsi -:t") ; if gsi not in executable path
(setq gambit-highlight-color "gray") ; if you don't like the default
(setq gambit-repl-command-prefix "\e") ; if you want M-c, M-s, etc
(require 'gambit)
```

6 Host environment access

The host environment is the set of resources, such as the file system, network and processes, that are managed by the operating system within which the Scheme program is executing. This chapter specifies how the host environment can be accessed from within the Scheme program.

In this chapter we say that the Scheme program being executed is a process, even though the concept of process does not exist in some operating systems supported by Gambit (e.g. MSDOS and Classic Mac OS).

6.1 Handling of file names

Gambit uses a naming convention for files that is compatible with the one used by the host environment but extended to allow referring to the *home directory* of the current user or some specific user and the *Gambit installation directory*.

A *path* is a string that denotes a file, for example `"src/readme.txt"`. Each component of a path is separated by a '`/`' under UNIX and Mac OS X, by a '`/`' or '\' under MSDOS and Microsoft Windows, and by a '`:`' under Classic Mac OS. A leading separator indicates an absolute path under UNIX, Mac OS X, MSDOS and Microsoft Windows but indicates a relative path under Classic Mac OS. A path which does not contain a path separator is relative to the *current working directory* on all operating systems, including Classic Mac OS. A volume specifier such as '`C:`' may prefix a file name under MSDOS and Microsoft Windows.

Under Classic Mac OS the folder '`Gambit-C`' must exist in the '`Preferences`' folder in the '`System`' folder and must not be an alias.

The rest of this section uses '`/`' to represent the path separator.

A path which starts with the characters '`~~/`' denotes a file in the Gambit installation directory. This directory is normally '`/usr/local/Gambit-C/`' under UNIX and Mac OS X, '`C:\Gambit-C\`' under MSDOS and Microsoft Windows, and under Classic Mac OS the '`Gambit-C`' folder. To override this binding under UNIX, Mac OS X, MSDOS and Microsoft Windows, use the '`-:=<dir>`' runtime option or define the '`GAMBCOPT`' environment variable.

A path which starts with the characters '`~/`' denotes a file in the user's home directory. The user's home directory is contained in the '`HOME`' environment variable under UNIX, Mac OS X, MSDOS and Microsoft Windows. Under MSDOS and Microsoft Windows, if the '`HOME`' environment variable is not defined, the environment variables '`HOMEDRIVE`' and '`HOMEPATH`' are concatenated if they are defined. If this fails to yield a home directory, the Gambit installation directory is used instead. Under Classic Mac OS the user's home directory is the folder which contains the application.

A path which starts with the characters '`~`*`username`*`/`' denotes a file in the home directory of the given user. Under UNIX and Mac OS X this is found using the password file. There is no equivalent under MSDOS, Microsoft Windows, and Classic Mac OS.

`(current-directory [`*`new-current-directory`*`])` [procedure]

The parameter object `current-directory` is bound to the current working directory. Calling this procedure with no argument returns the absolute *normalized path*

of the directory and calling this procedure with one argument changes the directory. The initial binding of this parameter object is the current working directory of the current process. Modifications of the parameter object do not change the current working directory of the current process. It is an error to mutate the string returned by `current-directory`.

For example under UNIX:

```
> (current-directory)
"/u/feeley/work/"
> (current-directory "..")
> (current-directory)
"/u/feeley/"
> (parameterize ((current-directory "~~")) (path-expand "foo"))
"/usr/local/Gambit-C/foo"
```

`(path-expand` *`path`* [*`origin-directory`*]`)` [procedure]

The procedure `path-expand` takes the path of a file or directory and returns an expanded path, which is an absolute path when *`path`* or *`origin-directory`* are absolute paths. The optional *`origin-directory`* parameter, which defaults to the current working directory, is the directory used to resolve relative paths. Components of the paths *`path`* and *`origin-directory`* need not exist.

For example under UNIX:

```
> (path-expand "foo")
"/u/feeley/work/foo"
> (path-expand "~/foo")
"/u/feeley/foo"
> (path-expand "~~/foo")
"/usr/local/Gambit-C/foo"
> (path-expand "../foo")
"/u/feeley/work/../foo"
> (path-expand "foo" "")
"foo"
> (path-expand "foo" "/tmp")
"/tmp/foo"
> (path-expand "this/file/does/not/exist")
"/u/feeley/work/this/file/does/not/exist"
> (path-expand "")
"/u/feeley/work/"
```

`(path-normalize` *`path`* [*`allow-relative?`* [*`origin-directory`*]]`)` [procedure]

The procedure `path-normalize` takes a path of a file or directory and returns its normalized path. The optional *`origin-directory`* parameter, which defaults to the current working directory, is the directory used to resolve relative paths. All components of the paths *`path`* and *`origin-directory`* must exist, except possibly the last component of *`path`*. A normalized path is a path containing no redundant parts and which is consistent with the current structure of the filesystem. A normalized

path of a directory will always end with a path separator (i.e. '`/`', '`\`', or '`:`' depending on the operating system). The optional *`allow-relative?`* parameter, which defaults to `#f`, indicates if the path returned can be expressed relatively to *`origin-directory`*: a `#f` requests an absolute path, the symbol `shortest` requests the shortest of the absolute and relative paths, and any other value requests the relative path. The shortest path is useful for interaction with the user because short relative paths are typically easier to read than long absolute paths.

`(path-extension` *`path`*`)` [procedure]
`(path-strip-extension` *`path`*`)` [procedure]
`(path-directory` *`path`*`)` [procedure]
`(path-strip-directory` *`path`*`)` [procedure]
`(path-volume` *`path`*`)` [procedure]
`(path-strip-volume` *`path`*`)` [procedure]

These procedures extract various parts of a path, which need not be a normalized path. The procedure `path-extension` returns the file extension (including the period) or the empty string if there is no extension. The procedure `path-strip-extension` returns the path with the extension stripped off. The procedure `path-directory` returns the file's directory (including the last path separator) or the empty string if no directory is specified in the path. The procedure `path-strip-directory` returns the path with the directory stripped off. The procedure `path-volume` returns the file's volume (including the last path separator) or the empty string if no volume is specified in the path. The procedure `path-strip-volume` returns the path with the volume stripped off.

For example under UNIX:

```
> (path-extension "/tmp/foo")
""
> (path-extension "/tmp/foo.txt")
".txt"
> (path-strip-extension "/tmp/foo.txt")
"/tmp/foo"
> (path-directory "/tmp/foo.txt")
"/tmp/"
> (path-strip-directory "/tmp/foo.txt")
"foo.txt"
> (path-volume "/tmp/foo.txt")
""
> (path-volume "C:/tmp/foo.txt")
""
> (path-expand "../foo")
"/u/feeley/work/../foo"
> (path-expand "foo" "")
"foo"
> (path-expand "foo" "/tmp")
"/tmp/foo"
> (path-expand "this/file/does/not/exist")
"/u/feeley/work/this/file/does/not/exist"
```

```
> (path-expand "")
"/u/feeley/work/"
```

6.2 Shell command execution

(shell-command *command*) [procedure]

The procedure `shell-command` calls up the shell to execute *command* which must be a string. This procedure returns the exit status of the shell in the form that the C library's `system` routine returns.

For example under UNIX:

```
> (shell-command "ls -sk f*.scm")
4 fact.scm   4 fib.scm
0
```

6.3 Process termination

(exit [*status*]) [procedure]

The procedure `exit` causes the process to terminate with the status *status* which must be an exact integer in the range 0 to 255. If it is not specified, the status defaults to 0.

For example under UNIX:

```
% gsi
Gambit Version 4.0 beta 7

> (exit 42)
% echo $?
42
```

6.4 Command line arguments

(command-line) [procedure]

The procedure `command-line` returns a list of strings corresponding to the command line arguments, including the program file name as the first element of the list. When the interpreter executes a Scheme script, the list returned by `command-line` contains the script's absolute path followed by the remaining command line arguments.

For example under UNIX:

```
% gsi -:d -e "(pp (command-line))"
("gsi" "-e" "(pp (command-line))")
% cat foo
#!/usr/local/Gambit-C/bin/gsi-script
(pp (command-line))
% ./foo 1 2 "3 4"
("/u/feeley/./foo" "1" "2" "3 4")
```

6.5 Environment variables

(getenv *name* [*default*]) [procedure]
(setenv *name value*) [procedure]

The procedure `getenv` returns the value of the environment variable *name* of the current process. Variable names are denoted with strings. A string is returned if the environment variable is bound, otherwise *default* is returned if it is specified, otherwise an exception is raised.

The procedure `setenv` changes the binding of the environment variable *name*. If *value* is `#f` the binding is removed.

For example under UNIX:

```
> (getenv "HOME")
"/u/feeley"
> (getenv "DOES_NOT_EXIST" #f)
#f
> (setenv "DOES_NOT_EXIST" "it does now")
> (getenv "DOES_NOT_EXIST" #f)
"it does now"
> (setenv "DOES_NOT_EXIST" #f)
> (getenv "DOES_NOT_EXIST" #f)
#f
```

6.6 Measuring time

Procedures are available for measuring real time (aka "wall" time) and cpu time (the amount of time the cpu has been executing the process). The resolution of the real time and cpu time clock is platform dependent. Typically the resolution of the cpu time clock is rather coarse (measured in "ticks" of 1/60th or 1/100th of a second). Real time is internally computed relative to some arbitrary point in time using floating point numbers, which means that there is a gradual loss of resolution as time elapses. Moreover, some operating systems report time in number of ticks using a 32 bit integer so the value returned by the time related procedures may wraparound much before any significant loss of resolution occurs (for example 2.7 years if ticks are 1/50th of a second).

(current-time) [procedure]
(time->seconds *time*) [procedure]
(seconds->time *x*) [procedure]

The procedure `current-time` returns a "time" object representing the current point in real time. The procedure `time->seconds` converts the time object *time* into an inexact real number representing the number of seconds elapsed since the "epoch" (which is 00:00:00 Coordinated Universal Time 01-01-1970). The procedure `time->seconds` converts the real number *x* representing the number of seconds elapsed since the "epoch" into a time object.

For example:

```
> (time->seconds (current-time))
1083118758.63973
> (time->seconds (current-time))
```

```
1083118759.909163
```

(process-times) [procedure]
(cpu-time) [procedure]
(real-time) [procedure]

The procedure `process-times` returns a three element f64vector containing the cpu time that has been used by the program and the real time that has elapsed since it was started. The first element corresponds to "user" time in seconds, the second element corresponds to "system" time in seconds and the third element is the elapsed real time in seconds. On operating systems that can't differentiate user and system time, the system time is zero. On operating systems that can't measure cpu time, the user time is equal to the elapsed real time and the system time is zero.

The procedure `cpu-time` returns the cpu time in seconds that has been used by the program (user time plus system time).

The procedure `real-time` returns the real time that has elapsed since the program was started.

For example:

```
> (process-times)
#f64(.07 0. 486.77118492126465)
> (cpu-time)
.08
> (real-time)
615.2873070240021
```

6.7 File information

(file-info *path*) [procedure]
(file-info? *object*) [procedure]
(file-info-attributes *file-info*) [procedure]
(file-info-creation-time *file-info*) [procedure]
(file-info-device *file-info*) [procedure]
(file-info-group *file-info*) [procedure]
(file-info-inode *file-info*) [procedure]
(file-info-last-access-time *file-info*) [procedure]
(file-info-last-change-time *file-info*) [procedure]
(file-info-last-modification-time *file-info*) [procedure]
(file-info-mode *file-info*) [procedure]
(file-info-number-of-links *file-info*) [procedure]
(file-info-owner *file-info*) [procedure]
(file-info-size *file-info*) [procedure]
(file-info-type *file-info*) [procedure]

(file-attributes *path*) [procedure]
(file-creation-time *path*) [procedure]
(file-device *path*) [procedure]
(file-group *path*) [procedure]
(file-inode *path*) [procedure]

(file-last-access-time *path*) [procedure]
(file-last-change-time *path*) [procedure]
(file-last-modification-time *path*) [procedure]
(file-mode *path*) [procedure]
(file-number-of-links *path*) [procedure]
(file-owner *path*) [procedure]
(file-size *path*) [procedure]
(file-type *path*) [procedure]

6.8 Group information

(group-info *group-name-or-id*) [procedure]
(group-info? *object*) [procedure]
(group-info-gid *group-info*) [procedure]
(group-info-members *group-info*) [procedure]
(group-info-name *group-info*) [procedure]

6.9 User information

(user-info *user-name-or-id*) [procedure]
(user-info? *object*) [procedure]
(user-info-gid *user-info*) [procedure]
(user-info-home *user-info*) [procedure]
(user-info-name *user-info*) [procedure]
(user-info-shell *user-info*) [procedure]
(user-info-uid *user-info*) [procedure]

6.10 Host information

(host-info *host-name-or-address*) [procedure]
(host-info? *object*) [procedure]
(host-info-addresses *host-info*) [procedure]
(host-info-aliases *host-info*) [procedure]
(host-info-name *host-info*) [procedure]

7 I/O and ports

7.1 Unidirectional and bidirectional ports

Unidirectional ports allow communication between a producer of information and a consumer. An input-port's producer is typically a resource managed by the operating system (such as a file, a process or a network connection) and the consumer is the Scheme program. The roles are reversed for an output-port.

Associated with each port are settings that affect I/O operations on that port (encoding of characters to bytes, end-of-line encoding, type of buffering, etc). Port settings are specified when the port is created. Some port settings can be changed after a port is created.

Bidirectional ports, also called input-output-ports, allow communication in both directions. They are best viewed as an object that groups two separate unidirectional ports (one in each direction). Each direction has its own port settings and can be closed independently from the other direction.

7.2 Port classes

The four classes of ports form an inheritance hierarchy. Operations possible for a certain class of port are also possible for the subclasses. Only device-ports are connected to a device managed by the operating system. For instance it is possible to create ports that behave as a FIFO where the Scheme program is both the producer and consumer of information.

1. An *object-port* (or simply a port) provides operations to read and write Scheme data (i.e. any Scheme object) to/from the port. It also provides operations to force output to occur, to change the way threads block on the port, and to close the port.
2. A *character-port* provides all the operations of an object-port, and also operations to read and write individual characters to/from the port. When a Scheme object is written to a character-port, it is converted into the sequence of characters that corresponds to its external-representation. When reading a Scheme object, an inverse conversion occurs.
3. A *byte-port* provides all the operations of a character-port, and also operations to read and write individual bytes to/from the port. When a character is written to a byte-port, some encoding of that character into a sequence of bytes will occur (for example, `#\newline` might be encoded as the 2 bytes CR-LF when using LATIN-1 encoding, or a non-ASCII character will generate more than 1 byte when using UTF8 encoding). When reading a character, a similar decoding occurs.
4. A *device-port* provides all the operations of a byte-port, and also operations to control the operating system managed device (file, network connection, terminal, etc) that is connected to the port.

7.3 Port settings

Some port settings are only valid for specific port classes whereas some others are valid for all ports. Port settings are specified when a port is created. The settings that are not specified will default to reasonable values. Keyword objects are used to name the settings to be set. As a simple example, a device-port connected to the file `"foo"` can be created using the call

```
(open-input-file "foo")
```

This will use default settings for the character encoding, buffering, etc. If the UTF8 character encoding is desired, then the port could be opened using the call

```
(open-input-file (list path: "foo" char-encoding: 'utf8))
```

Here the argument of the procedure `open-input-file` has been replaced by a *port settings list* which specifies the value of each port setting that should not be set to the default value. Note that some port settings are required, such as the `path:` in the case of the file opening procedures. All port creation procedures (i.e. named `open-...`) take a single argument that can either be a port settings list or a value of a type that depends on the kind of port being created.

What follows is a list of port settings that are valid for more than one type of port. The port settings that are specific to a type of port are described with the corresponding port creation procedure.

- `direction:` (`input` | `output` | `input-output`)

  This setting controls the direction of the port. The symbol `input` indicates a unidirectional input-port, the symbol `output` indicates a unidirectional output-port, and the symbol `input-output` indicates a bidirectional port. The default depends on the port creation procedure.

- `readtable:` *readtable*

  This setting determines the readtable attached to the character-port. Readtables control the external representation of Scheme objects, that is the encoding of Scheme objects using characters. The behavior of the reader (i.e. the procedure `read`) and of the printer (i.e. the procedures `write`, `pretty-print`, etc) are affected by the port's readtable. The default is the value bound to the parameter object `current-readtable`.

- `char-encoding:` *encoding*

  This setting controls the character encoding of the byte-port. For bidirectional byte-ports, the character encoding for input and output is set. To set each direction separately the keywords `input-char-encoding:` and `output-char-encoding:` must be used instead of `char-encoding:`. The default is operating system dependent, but this can be overriden through the runtime options (see ⟨undefined⟩ [Runtime options], page ⟨undefined⟩). The following encodings are supported:

  `latin1` LATIN1 character encoding. Each character is encoded by a single byte. Only Unicode characters with a code in the range 0 to 255 are allowed.

  `ascii` ASCII character encoding. Each character is encoded by a single byte. In principle only Unicode characters with a code in the range 0 to 127 are allowed but most types of ports treat this exactly like `latin1`.

  `ucs2` UCS2 character encoding. Each character is encoded by 16 bits, i.e. two bytes. The 16 bits may be encoded using little-endian encoding or big-endian encoding. If the port is an input-port and the first two bytes read are a BOM ("Byte Order Mark" character with hexadecimal code FEFF) then the BOM will be discarded and the endianness will be set accordingly, otherwise the endianness depends on the operating system and how the

Gambit runtime was compiled. If the port is an output-port then a BOM will be output at the beginning of the stream and the endianness depends on the operating system and how the Gambit runtime was compiled.

`ucs2le` UCS2 character encoding with little-endian endianness. It is like `ucs2` except the endianness is set to little-endian and there is no BOM processing. If a BOM is needed at the beginning of the stream then it must be explicitly written.

`ucs2be` UCS2 character encoding with big-endian endianness. It is like `ucs2le` except the endianness is set to big-endian.

`ucs4` UCS4 character encoding. Each character is encoded by 32 bits, i.e. four bytes. The 32 bits may be encoded using little-endian encoding or big-endian encoding. If the port is an input-port and the first four bytes read are a BOM ("Byte Order Mark" character with hexadecimal code 0000FEFF) then the BOM will be discarded and the endianness will be set accordingly, otherwise the endianness depends on the operating system and how the Gambit runtime was compiled. If the port is an output-port then a BOM will be output at the beginning of the stream and the endianness depends on the operating system and how the Gambit runtime was compiled.

`ucs4le` UCS4 character encoding with little-endian endianness. It is like `ucs4` except the endianness is set to little-endian and there is no BOM processing. If a BOM is needed at the beginning of the stream then it must be explicitly written.

`ucs4be` UCS4 character encoding with big-endian endianness. It is like `ucs4le` except the endianness is set to big-endian.

`native` Native character encoding using one byte per character. Currently this is treated the same as `latin1`.

- `eol-encoding:` *encoding*

  This setting controls the end-of-line encoding of the byte-port. To set each direction separately the keywords `input-eol-encoding:` and `output-eol-encoding:` must be used instead of `eol-encoding:`. The default is operating system dependent, but this can be overriden through the runtime options (see ⟨undefined⟩ [Runtime options], page ⟨undefined⟩). Note that for output-ports the end-of-line encoding is applied before the character encoding, and for input-ports it is applied after. The following encodings are supported:

  `lf` For an output-port, writing a `#\newline` character outputs a `#\linefeed` character to the stream (Unicode character code 10). For an input-port, a `#\newline` character is read when a `#\linefeed` character is encountered on the stream. Note that `#\linefeed` and `#\newline` are two names for the same character, so this end-of-line encoding is actually the identity function. Text files created by UNIX applications typically use this end-of-line encoding.

  `cr` For an output-port, writing a `#\newline` character outputs a `#\return` character to the stream (Unicode character code 10). For an input-port,

a `#\newline` character is read when a `#\linefeed` character or a `#\return` character is encountered on the stream. Text files created by Classic Mac OS applications typically use this end-of-line encoding.

`cr-lf` For an output-port, writing a `#\newline` character outputs to the stream a `#\return` character followed by a `#\linefeed` character. For an input-port, a `#\newline` character is read when a `#\linefeed` character or a `#\return` character is encountered on the stream. Moreover, if this character is immediately followed by the opposite character (`#\linefeed` followed by `#\return` or `#\return` followed by `#\linefeed`) then the second character is ignored. In other words, all four possible end-of-line encodings are read as a single `#\newline` character. Text files created by DOS and Microsoft Windows applications typically use this end-of-line encoding.

`lf-cr` For an output-port, writing a `#\newline` character outputs to the stream a `#\linefeed` character followed by a `#\return` character. For an input-port, this is exactly like `cr-lf`.

- `buffering:` (`#f` | `#t` | `line`)

  This setting controls the buffering of the port. To set each direction separately the keywords `input-buffering:` and `output-buffering:` must be used instead of `buffering:`. The value `#f` selects unbuffered I/O, the value `#t` selects fully buffered I/O, and the symbol `line` selects line buffered I/O (the output buffer is drained when a `#\newline` character is written). The default is operating system dependent except consoles which are unbuffered.

- `output-width:` *positive-integer*

  This setting indicates the width of the character-port in number of characters. This information is used by the pretty-printer. The default is 80.

7.4 Object-ports

`(input-port?` *obj*`)` [procedure]
`(output-port?` *obj*`)` [procedure]
`(port?` *obj*`)` [procedure]

The procedure `input-port?` returns `#t` if *obj* is a unidirectional input-port or a bidirectional port and `#f` otherwise.

The procedure `output-port?` returns `#t` if *obj* is a unidirectional output-port or a bidirectional port and `#f` otherwise.

The procedure `port?` returns `#t` if *obj* is a port (either unidirectional or bidirectional) and `#f` otherwise.

`(read` [*port*]`)` [procedure]

This procedure reads and returns the next Scheme datum from the input-port *port*. The end-of-file object is returned when the end of the stream is reached. If it is not specified, *port* defaults to the current input-port.

`(read-all [port [reader]])` [procedure]

This procedure repeatedly calls the procedure *reader* with *port* as the sole parameter and accumulates a list of each value returned up to the end-of-file object. The procedure `read-all` return the accumulated list. If it is not specified, *port* defaults to the current input-port. If it is not specified, *reader* defaults to the procedure `read`.

`(write obj [port])` [procedure]

This procedure writes the Scheme datum *obj* to the output-port *port* and the value returned is unspecified. If it is not specified, *port* defaults to the current output-port.

`(newline [port])` [procedure]

This procedure writes an "object separator" to the output-port *port* and the value returned is unspecified. The separator ensures that the next Scheme datum written with the `write` procedure will not be confused with the latest datum that was written. On character-ports this is done by writing the character `#\newline`. On other ports this procedure does nothing.

Regardless of the class of a port *p* and assuming that the external textual representation of the object *x* is readable, then the expression `(begin (write x p) (newline p))` will write to *p* a representation of *x* that can be read back with the procedure `read`. If it is not specified, *port* defaults to the current output-port.

`(force-output [port])` [procedure]

The procedure `force-output` causes the output buffers of the output-port *port* to be drained (i.e. the data is sent to its destination). If *port* is not specified, the current output port is used.

`(close-input-port port)` [procedure]
`(close-output-port port)` [procedure]
`(close-port port)` [procedure]

The *port* argument of these procedures must be a unidirectional or a bidirectional port. For all three procedures the value returned is unspecified.

The procedure `close-input-port` closes the input-port side of *port*, which must not be a unidirectional output-port.

The procedure `close-output-port` closes the output-port side of *port*, which must not be a unidirectional input-port. The ouput buffers are drained before *port* is closed.

The procedure `close-port` closes all sides of the *port*. Unless *port* is a unidirectional input-port, the ouput buffers are drained before *port* is closed.

`(input-port-timeout-set! port timeout [thunk])` [procedure]
`(output-port-timeout-set! port timeout [thunk])` [procedure]

When a thread tries to perform an I/O operation on a port, the requested operation may not be immediately possible and the thread must wait. For example, the thread may be trying to read a line of text from the console and the user has not typed anything yet, or the thread may be trying to write to a network connection faster than the network can handle. In such situations the thread normally blocks until the operation becomes possible.

It is sometimes necessary to guarantee that the thread will not block too long. For this purpose, to each input-port and output-port is attached a *timeout* and *timeout-thunk*. The timeout indicates the point in time beyond which the thread should stop waiting on an input and output operation respectively. When the timeout is reached, the thread calls the port's timeout-thunk. If the timeout-thunk returns `#f` the thread abandons trying to perform the operation (in the case of an input operation an end-of-file is read and in the case of an output operation an exception is raised). Otherwise, the thread will block again waiting for the operation to become possible (note that if the port's timeout has not changed the thread will immediately call the timeout-thunk again).

The procedure `input-port-timeout-set!` sets the timeout of the input-port *port* to *timeout* and the timeout-thunk to *thunk*. The procedure `output-port-timeout-set!` sets the timeout of the output-port *port* to *timeout* and the timeout-thunk to *thunk*. If it is not specified, the *thunk* defaults to a thunk that returns `#f`. The *timeout* is either a time object indicating an absolute point in time, or it is a real number which indicates the number of seconds relative to the moment the procedure is called. For both procedures the value returned is unspecified.

When a port is created the timeout is set to infinity (`+inf.`). This causes the thread to wait as long as needed for the operation to become possible. Setting the timeout to a point in the past (`-inf.`) will cause the thread to attempt the I/O operation and never block (i.e. the timeout-thunk is called if the operation is not immediately possible).

7.5 Character-ports

`(input-port-line` *port*`)` [procedure]
`(input-port-column` *port*`)` [procedure]
`(output-port-line` *port*`)` [procedure]
`(output-port-column` *port*`)` [procedure]
`(output-port-width` *port*`)` [procedure]

`(read-char` [*port*]`)` [procedure]
This procedure reads and returns the next character from the input-port *port*. The end-of-file object is returned when the end of the stream is reached. If it is not specified, *port* defaults to the current input-port.

`(peek-char` [*port*]`)` [procedure]
...

`(write-char` *char* [*port*]`)` [procedure]
This procedure writes the character *char* to the output-port *port* and the value returned is unspecified. If it is not specified, *port* defaults to the current output-port.

`(read-line` [*port* [*separator* [*include-separator?*]]]`)` [procedure]

`(read-substring` *string start end* [*port*]`)` [procedure]
`(write-substring` *string start end* [*port*]`)` [procedure]

7.6 Byte-ports

(read-byte [*port*]) [procedure]

(write-byte *n* [*port*]) [procedure]

(read-subu8vector *u8vector start end* [*port*]) [procedure]
(write-subu8vector *u8vector start end* [*port*]) [procedure]

7.7 Device-ports

(open-file *path-or-settings*) [procedure]
(open-input-file *path-or-settings*) [procedure]
(open-output-file *path-or-settings*) [procedure]
(call-with-input-file *path-or-settings proc*) [procedure]
(call-with-output-file *path-or-settings proc*) [procedure]
(with-input-from-file *path-or-settings thunk*) [procedure]
(with-output-to-file *path-or-settings thunk*) [procedure]

These procedures take an optional argument which specifies special settings (character encoding, end-of-line encoding, buffering, etc).

*** This documentation is incomplete!

(tty? *port*) [procedure]
(tty-type-set! *port term-type emacs-bindings*) [procedure]
(tty-text-attributes-set! *port input output*) [procedure]
(tty-history *port*) [procedure]
(tty-history-set! *port history*) [procedure]
(tty-max-history-length-set! *port max-length*) [procedure]
(tty-paren-balance-duration-set! *port duration*) [procedure]
(tty-mode-set! *port mode*) [procedure]

(open-vector [*vector* [*settings*]]) [procedure]
(open-input-vector *vector* [*settings*]) [procedure]
(open-output-vector [*settings*]) [procedure]
(get-output-vector *port*) [procedure]
(call-with-input-vector *vector proc* [*settings*]) [procedure]
(call-with-output-vector *proc* [*settings*]) [procedure]
(with-input-from-vector *vector thunk* [*settings*]) [procedure]
(with-output-to-vector *thunk* [*settings*]) [procedure]
(open-string [*string* [*settings*]]) [procedure]
(open-input-string *string* [*settings*]) [procedure]
(open-output-string [*settings*]) [procedure]
(get-output-string *port*) [procedure]
(call-with-input-string *string proc* [*settings*]) [procedure]
(call-with-output-string *proc* [*settings*]) [procedure]
(with-input-from-string *string thunk* [*settings*]) [procedure]
(with-output-to-string *thunk* [*settings*]) [procedure]
(open-u8vector [*u8vector* [*settings*]]) [procedure]
(open-input-u8vector *u8vector* [*settings*]) [procedure]

(open-output-u8vector [*settings*]) [procedure]
(get-output-u8vector *port*) [procedure]
(call-with-input-u8vector *u8vector proc* [*settings*]) [procedure]
(call-with-output-u8vector *proc* [*settings*]) [procedure]
(with-input-from-u8vector *u8vector thunk* [procedure]
[*settings*])
(with-output-to-u8vector *thunk* [*settings*]) [procedure]

(directory-files *path*) [procedure]

(create-directory *path* [*settings*]) [procedure]
(create-fifo *path* [*settings*]) [procedure]
(create-link *source-path destination-path*) [procedure]
(create-symbolic-link *source-path* [procedure]
destination-path)
(rename-file *source-path destination-path*) [procedure]
(copy-file *source-path destination-path*) [procedure]
(delete-file *path*) [procedure]
(delete-directory *path*) [procedure]

(current-console-port [*new-value*]) [procedure]
(current-directory [*new-value*]) [procedure]
(current-error-port [*new-value*]) [procedure]
(current-exception-handler [*new-value*]) [procedure]
(current-input-port [*new-value*]) [procedure]
(current-output-port [*new-value*]) [procedure]
(current-readtable [*new-value*]) [procedure]
(current-user-interrupt-handler [*new-value*]) [procedure]

(open-file *path* [*settings*]) [procedure]
(open-input-file *path* [*settings*]) [procedure]
(open-output-file *path* [*settings*]) [procedure]
(open-string *string* [*settings*]) [procedure]
(open-input-string *string* [*settings*]) [procedure]
(open-output-string *string* [*settings*]) [procedure]
(open-vector *vector* [*settings*]) [procedure]
(open-input-vector *vector* [*settings*]) [procedure]
(open-output-vector *vector* [*settings*]) [procedure]
(open-u8vector *u8vector* [*settings*]) [procedure]
(open-input-u8vector *u8vector* [*settings*]) [procedure]
(open-output-u8vector *u8vector* [*settings*]) [procedure]

(open-directory *path* [*settings*]) [procedure]

(open-process *path-and-arguments* [*environment* [procedure]
[*settings*]])

(open-tcp-client *host-name-or-address port-number* [procedure]
[*settings*])
(open-tcp-server *port-number* [*settings*]) [procedure]

(readtable? *object*) [procedure]

`(readtable-copy` *`readtable`*`)` [procedure]
`(readtable-case-conversion?-set!` *`readtable new-value`*`)` [procedure]
`(readtable-eval-allowed?-set!` *`readtable new-value`*`)` [procedure]
`(readtable-keywords-allowed?-set!` *`readtable new-value`*`)` [procedure]
`(readtable-max-write-length-set!` *`readtable new-value`*`)` [procedure]
`(readtable-max-write-level-set!` *`readtable new-value`*`)` [procedure]
`(readtable-sharing-allowed?-set!` *`readtable new-value`*`)` [procedure]
`(readtable-sharp-quote-keyword-set!` *`readtable new-value`*`)` [procedure]
`(readtable-start-syntax-set!` *`readtable new-value`*`)` [procedure]

8 Section 4

9 Numbers

`(bitwise-ior` *n...*`)` [procedure]
`(bitwise-xor` *n...*`)` [procedure]
`(bitwise-and` *n...*`)` [procedure]
`(bitwise-not` *n*`)` [procedure]
`(arithmetic-shift` *n1 n2*`)` [procedure]
`(bit-count` *n*`)` [procedure]
`(integer-length` *n*`)` [procedure]
`(bitwise-merge` *n1 n2 n3*`)` [procedure]
`(bit-set?` *n1 n2*`)` [procedure]
`(any-bits-set?` *n1 n2*`)` [procedure]
`(all-bits-set?` *n1 n2*`)` [procedure]
`(first-set-bit` *n*`)` [procedure]
`(extract-bit-field` *n1 n2 n3*`)` [procedure]
`(test-bit-field?` *n1 n2 n3*`)` [procedure]
`(clear-bit-field` *n1 n2 n3*`)` [procedure]
`(insert-bit-field` *n1 n2 n3 n4*`)` [procedure]
`(copy-bit-field` *n1 n2 n3 n4*`)` [procedure]

`(integer-sqrt` *n*`)` [procedure]
`(integer-nth-root` *n1 n2*`)` [procedure]

The procedure `integer-sqrt` returns the integer part of the square root of the non-negative exact integer *n*.

The procedure `integer-nth-root` returns the integer part of *n1* raised to the power $1/n2$, where *n1* is a non-negative exact integer and *n2* is a positive exact integer.

For example:

```
> (integer-sqrt 123)
11
> (integer-nth-root 100 3)
4
```

10 Scheme infix syntax extension

The reader supports an infix syntax extension which is called SIX (Scheme Infix eXtension). This extension is both supported by the '`read`' procedure and in program source code.

The backslash character is a delimiter that marks the beginning of a single datum expressed in the infix syntax (the details are given below). One way to think about it is that the backslash character escapes the prefix syntax temporarily to use the infix syntax. For example a three element list could be written as '`(X \Y Z)`', the elements X and Z are expressed using the normal prefix syntax and Y is expressed using the infix syntax.

When the reader encounters an infix datum, it constructs a syntax tree for that particular datum. Each node of this tree is represented with a list whose first element is a symbol indicating the type of node. For example, '`(six.identifier abc)`' is the representation of the infix identifier '`abc`' and '`(six.index (six.identifier abc) (six.identifier i))`' is the representation of the infix datum '`abc[i];`'.

The infix grammar is shown below with the corresponding syntax tree representation on the right hand side. *** The grammar is out of date!

```
<infix datum> ::=
   <stat>                                        $1

<stat> ::=
   <if stat>                                     $1
 | <while stat>                                  $1
 | <for stat>                                    $1
 | <expression stat>                             $1
 | <block>                                       $1
 | ;                                             (six.compound)

<if stat> ::=
   if ( <expr> ) <stat>                          (six.if $3 $5)
 | if ( <expr> ) <stat> else <stat>              (six.if $3 $5 $7)

<while stat> ::=
   while ( <expr> ) <stat>                       (six.while $3 $5)

<for stat> ::=
    for ( <oexpr> ; <oexpr> ; <oexpr> )  (six.for $3 $5 $7 $9)
<stat>
<oexpr> ::=
   <expr>                                        $1
 |                                               #f

<expression stat> ::=
   <expr> ;                                      (six.expression $1)
 | <expr> .                                      (six.clause $1)
```

```
<expr> ::=
   <expr18>                                  $1

<expr18> ::=
   <expr17> :- <expr18>                      (six.x:-y $1 $3)
 | <expr17>                                  $1

<expr17> ::=
   <expr17> , <expr16>                       (|six.x,y| $1 $3)
 | <expr16>                                  $1

<expr16> ::=
   <expr15> := <expr16>                      (six.x:=y $1 $3)
 | <expr15>                                  $1

<expr15> ::=
   <expr14> %= <expr15>                      (six.x%=y $1 $3)
 | <expr14> &= <expr15>                      (six.x&=y $1 $3)
 | <expr14> *= <expr15>                      (six.x*=y $1 $3)
 | <expr14> += <expr15>                      (six.x+=y $1 $3)
 | <expr14> -= <expr15>                      (six.x-=y $1 $3)
 | <expr14> /= <expr15>                      (six.x/=y $1 $3)
 | <expr14> <<= <expr15>                     (six.x<<=y $1 $3)
 | <expr14> = <expr15>                       (six.x=y $1 $3)
 | <expr14> >>= <expr15>                     (six.x>>=y $1 $3)
 | <expr14> ^= <expr15>                      (six.x^=y $1 $3)
 | <expr14> |= <expr15>                      (|six.x\|=y| $1 $3)
 | <expr14>                                  $1

<expr14> ::=
   <expr13> : <expr14>                       (six.x:y $1 $3)
 | <expr13>                                  $1

<expr13> ::=
   <expr12> ? <expr> : <expr13>              (six.x?y:z $1 $3 $5)
 | <expr12>                                  $1

<expr12> ::=
   <expr12> || <expr11>                      (|six.x\|\|y| $1 $3)
 | <expr11>                                  $1

<expr11> ::=
   <expr11> && <expr10>                      (six.x&&y $1 $3)
 | <expr10>                                  $1

<expr10> ::=
   <expr10> | <expr9>                        (|six.x\|y| $1 $3)
```

```
  | <expr9>                              $1

<expr9> ::=
    <expr9> ^ <expr8>                   (six.x^y $1 $3)
  | <expr8>                             $1

<expr8> ::=
    <expr8> & <expr7>                   (six.x&y $1 $3)
  | <expr7>                             $1

<expr7> ::=
    <expr7> != <expr6>                  (six.x!=y $1 $3)
  | <expr7> == <expr6>                  (six.x==y $1 $3)
  | <expr6>                             $1

<expr6> ::=
    <expr6> < <expr5>                   (six.x<y $1 $3)
  | <expr6> <= <expr5>                  (six.x<=y $1 $3)
  | <expr6> > <expr5>                   (six.x>y $1 $3)
  | <expr6> >= <expr5>                  (six.x>=y $1 $3)
  | <expr5>                             $1

<expr5> ::=
    <expr5> << <expr4>                  (six.x<<y $1 $3)
  | <expr5> >> <expr4>                  (six.x>>y $1 $3)
  | <expr4>                             $1

<expr4> ::=
    <expr4> + <expr3>                   (six.x+y $1 $3)
  | <expr4> - <expr3>                   (six.x-y $1 $3)
  | <expr3>                             $1

<expr3> ::=
    <expr3> % <expr2>                   (six.x%y $1 $3)
  | <expr3> * <expr2>                   (six.x*y $1 $3)
  | <expr3> / <expr2>                   (six.x/y $1 $3)
  | <expr2>                             $1

<expr2> ::=
    & <expr2>                           (six.&x $2)
  | + <expr2>                           (six.+x $2)
  | - <expr2>                           (six.-x $2)
  | * <expr2>                           (six.*x $2)
  | ! <expr2>                           (six.!x $2)
  | !                                   (six.cut)
  | ++ <expr2>                          (six.++x $2)
  | -- <expr2>                          (six.--x $2)
```

```
  | ~ <expr2>                                  (six.~x $2)
  | <expr1>                                    $1

<expr1> ::=
    <expr1> ++                                 (six.x++ $1)
  | <expr1> --                                 (six.x-- $1)
  | <expr1> ( ... )                            (six.call $1)
  | <expr1> [ ... ]                            (six.index $1)
  | <expr1> -> <identifier>                    (six.arrow $1 $3)
  | <expr1> . <identifier>                     (six.dot $1 $3)
  | <expr0>                                    $1

<expr0> ::=
    <identifier>                               (six.identifier $1)
  | <string>                                   (six.string $1)
  | <char>                                     (six.char $1)
  | <number>                                   (six.number $1)
  | ( <expr> )                                 $2
  | ( <block> )                                $2
  | [ ... ]                                    (six.list $1)
  | \ <datum>                                  (six.prefix $2)
  | <type> ( <parameter list> ) <block>        (six.function $1 $3 $5)

<block> ::=
    { <stat list> }                            (six.compound . $2)

<stat list> ::=
    <stat> <stat list>                         ($1 . $2)
  |                                            ()

<declaration> ::=
    <type> <identifier> = <expr> ;             (six.declaration $1 $2
                                               $4)
  | <type> <identifier> ( <parameter list> )   (six.function-
<block>                                        declaration $1 $2 $4
                                               $6)

<parameter list> ::=
    <nonempty parameter list>                  $1
  |                                            ()

<nonempty parameter list> ::=
    <parameter> , <nonempty parameter list>    ($1 . $2)
  | <parameter>                                ($1)

<parameter> ::=
```

<type> <identifier>	`(`*`$1 $2`*`)`

<type> ::=

`obj`	`obj`
`| int`	`int`
`| void`	`void`

To make SIX useful for writing programs, most of the symbols representing the type of node are predefined macros which approximate the semantics of C. The semantics of SIX can be changed or extended by redefining these macros.

11 Handling of file names

...

12 Emacs interface

...

13 Extensions to Scheme

The Gambit Scheme system conforms to the R4RS, R5RS and IEEE Scheme standards. Gambit supports a number of extensions to these standards by extending the behavior of standard special forms and procedures, and by adding special forms and procedures.

13.1 Standard special forms and procedures

The extensions given in this section are all compatible with the Scheme standards. This means that the special forms and procedures behave as defined in the standards when they are used according to the standards.

`(transcript-on` *`file`*`)` [procedure]
`(transcript-off)` [procedure]

These procedures do nothing.

`(read` [*`port`* [*`readtable`*]]`)` [procedure]
`(write` *`obj`* [*`port`* [*`readtable`*]]`)` [procedure]
`(display` *`obj`* [*`port`* [*`readtable`*]]`)` [procedure]

The `read`, `write` and `display` procedures take an optional *readtable* argument which specifies the readtable to use. If it is not specified, the readtable defaults to the current readtable.

These procedures support the following features.

- Keyword objects (see ⟨undefined⟩ [procedure `keyword?`], page ⟨undefined⟩).
- Extended character names:

  `#\newline`
  : newline character

  `#\space` space character

  `#\nul` Unicode character 0

  `#\bel` Unicode character 7

  `#\backspace`
  : Unicode character 8

  `#\tab` Unicode character 9

  `#\linefeed`
  : Unicode character 10

  `#\vt` Unicode character 11

  `#\page` Unicode character 12

  `#\return`
  : Unicode character 13

  `#\rubout`
  : Unicode character 127

`#\`*n* Unicode character *n* (*n* must start with a "#" character and it must represent an exact integer, for example `#\#x20` is the space character, `#\#d9` is the tab character, and `#\#e1.2e2` is the lower case character "x")

- Escape sequences inside character strings:

  `\n` newline character

  `\a` Unicode character 7

  `\b` Unicode character 8

  `\t` Unicode character 9

  `\v` Unicode character 11

  `\f` Unicode character 12

  `\r` Unicode character 13

  `\"` "

  `\\` \

  `\`*ooo* character encoded in octal (1 to 3 octal digits)

  `\x`*hh* character encoded in hexadecimal (>= 1 hexadecimal digit)

- Symbols can be represented with a leading and trailing vertical bar (i.e. '`|`'). The symbol's name corresponds verbatim to the characters between the vertical bars except for escaped characters. The same escape sequences as for strings are permitted except that '`"`' does not need to be escaped and '`|`' needs to be escaped (in other words the function of the '`"`' and '`|`' characters is interchanged with respect to the string syntax).
- Multiline comments are delimited by the tokens '`#|`' and '`|#`'. These comments can be nested.
- Special "`#!`" objects:

  `#!eof` end-of-file object

  `#!void` void object

  `#!optional` optional object

  `#!rest` rest object

  `#!key` key object

- Special inexact real numbers:

  `+inf.` positive infinity

  `-inf.` negative infinity

  `+nan.` "not a number"

  `-0.` negative zero ('`0.`' is the positive zero)

- Bytevectors are uniform vectors containing raw numbers (signed or unsigned exact integers or inexact reals). There are 10 types of bytevectors: '`s8vector`' (vector of 8 bit signed integers), '`u8vector`' (vector of 8 bit unsigned integers), '`s16vector`' (vector of 16 bit signed integers), '`u16vector`' (vector of 16 bit unsigned integers), '`s32vector`' (vector of 32 bit signed integers), '`u32vector`' (vector of 32 bit unsigned integers), '`s64vector`' (vector of 64 bit signed integers), '`u64vector`' (vector of 64 bit unsigned integers), '`f32vector`' (vector of 32 bit floating point numbers), and '`f64vector`' (vector of 64 bit floating point numbers). The external representation of bytevectors is similar to normal vectors but with the '`#(`' prefix replaced respectively with '`#s8(`', '`#u8(`', '`#s16(`', '`#u16(`', '`#s32(`', '`#u32(`', '`#s64(`', '`#u64(`', '`#f32(`', and '`#f64(`'. The elements of the integer bytevectors must be exact integers fitting in the given precision. The elements of the floating point bytevectors must be inexact reals.

`(= z1...)` [procedure]
`(< x1...)` [procedure]
`(> x1...)` [procedure]
`(<= x1...)` [procedure]
`(>= x1...)` [procedure]
`(char=? char1...)` [procedure]
`(char<? char1...)` [procedure]
`(char>? char1...)` [procedure]
`(char<=? char1...)` [procedure]
`(char>=? char1...)` [procedure]
`(char-ci=? char1...)` [procedure]
`(char-ci<? char1...)` [procedure]
`(char-ci>? char1...)` [procedure]
`(char-ci<=? char1...)` [procedure]
`(char-ci>=? char1...)` [procedure]
`(string=? string1...)` [procedure]
`(string<? string1...)` [procedure]
`(string>? string1...)` [procedure]
`(string<=? string1...)` [procedure]
`(string>=? string1...)` [procedure]
`(string-ci=? string1...)` [procedure]
`(string-ci<? string1...)` [procedure]
`(string-ci>? string1...)` [procedure]
`(string-ci<=? string1...)` [procedure]
`(string-ci>=? string1...)` [procedure]

These procedures take any number of arguments including no argument. This is useful to test if the elements of a list are sorted in a particular order. For example, testing that the list of numbers `lst` is sorted in nondecreasing order can be done with the call `(apply < lst)`.

13.2 Additional special forms and procedures

`(include` *`file`*`)` [special form]

file must be a string naming an existing file containing Scheme source code. The `include` special form splices the content of the specified source file. This form can only appear where a `define` form is acceptable.

For example:

```
(include "macros.scm")

(define (f lst)
  (include "sort.scm")
  (map sqrt (sort lst)))
```

`(define-macro (`*`name arg...`*`) `*`body`*`)` [special form]

Define *name* as a macro special form which expands into *body*. This form can only appear where a `define` form is acceptable. Macros are lexically scoped. The scope of a local macro definition extends from the definition to the end of the body of the surrounding binding construct. Macros defined at the top level of a Scheme module are only visible in that module. To have access to the macro definitions contained in a file, that file must be included using the `include` special form. Macros which are visible from the REPL are also visible during the compilation of Scheme source files.

For example:

```
(define-macro (push val var)
  `(set! ,var (cons ,val ,var)))

(define-macro (unless test . body)
  `(if ,test #f (begin ,@body)))
```

To examine the code into which a macro expands you can use the compiler's '`-expansion`' option or the `pp` procedure. For example:

```
> (define-macro (push val var) `(set! ,var (cons ,val ,var)))
> (pp (lambda () (push 1 stack) (push 2 stack) (push 3 stack)))
(lambda ()
  (set! stack (cons 1 stack))
  (set! stack (cons 2 stack))
  (set! stack (cons 3 stack)))
```

`(declare` *`declaration...`*`)` [special form]

This form introduces declarations to be used by the compiler (currently the interpreter ignores the declarations). This form can only appear where a `define` form is acceptable. Declarations are lexically scoped in the same way as macros. The following declarations are accepted by the compiler:

`(`*`dialect`*`)`

Use the given dialect's semantics. *dialect* can be: '`ieee-scheme`' or '`r4rs-scheme`'.

`(`*strategy*`)`
: Select block compilation or separate compilation. In block compilation, the compiler assumes that global variables defined in the current file that are not mutated in the file will never be mutated. *strategy* can be: '`block`' or '`separate`'.

`([not] inline)`
: Allow (or disallow) inlining of user procedures.

`(inlining-limit` *n*`)`
: Select the degree to which the compiler inlines user procedures. *n* is the upper-bound, in percent, on code expansion that will result from inlining. Thus, a value of 300 indicates that the size of the program will not grow by more than 300 percent (i.e. it will be at most 4 times the size of the original). A value of 0 disables inlining. The size of a program is the total number of subexpressions it contains (i.e. the size of an expression is one plus the size of its immediate subexpressions). The following conditions must hold for a procedure to be inlined: inlining the procedure must not cause the size of the call site to grow more than specified by the inlining limit, the site of definition (the `define` or `lambda`) and the call site must be declared as `(inline)`, and the compiler must be able to find the definition of the procedure referred to at the call site (if the procedure is bound to a global variable, the definition site must have a `(block)` declaration). Note that inlining usually causes much less code expansion than specified by the inlining limit (an expansion around 10% is common for n=300).

`([not] lambda-lift)`
: Lambda-lift (or don't lambda-lift) locally defined procedures.

`([not] standard-bindings` *var*`...)`
: The given global variables are known (or not known) to be equal to the value defined for them in the dialect (all variables defined in the standard if none specified).

`([not] extended-bindings` *var*`...)`
: The given global variables are known (or not known) to be equal to the value defined for them in the runtime system (all variables defined in the runtime if none specified).

`([not] safe)`
: Generate (or don't generate) code that will prevent fatal errors at run time. Note that in '`safe`' mode certain semantic errors will not be checked as long as they can't crash the system. For example the primitive `char=?` may disregard the type of its arguments in '`safe`' as well as '`not safe`' mode.

`([not] interrupts-enabled)`
: Generate (or don't generate) interrupt checks. Interrupt checks are used to detect user interrupts and also to check for stack overflows. Interrupt checking should not be turned off casually.

`(`*`number-type primitive`*`...)`

Numeric arguments and result of the specified primitives are known to be of the given type (all primitives if none specified). *number-type* can be: '`generic`', '`fixnum`', or '`flonum`'.

The default declarations used by the compiler are equivalent to:

```
(declare
  (ieee-scheme)
  (separate)
  (inline)
  (inlining-limit 300)
  (lambda-lift)
  (not standard-bindings)
  (not extended-bindings)
  (safe)
  (interrupts-enabled)
  (generic)
)
```

These declarations are compatible with the semantics of Scheme. Typically used declarations that enhance performance, at the cost of violating the Scheme semantics, are: `(standard-bindings)`, `(block)`, `(not safe)` and `(fixnum)`.

`(lambda` *`lambda-formals body`*`)` [special form]

`(define (`*`variable define-formals`*`)` *`body`*`)` [special form]

lambda-formals = `(` *formal-argument-list* `)` | *r4rs-lambda-formals*

define-formals = *formal-argument-list* | *r4rs-define-formals*

formal-argument-list = *reqs opts rest keys*

reqs = *required-formal-argument**

required-formal-argument = *variable*

opts = `#!optional` *optional-formal-argument** | *empty*

optional-formal-argument = *variable* | `(` *variable initializer* `)`

rest = `#!rest` *rest-formal-argument* | *empty*

rest-formal-argument = *variable*

keys = `#!key` *keyword-formal-argument** | *empty*

keyword-formal-argument = *variable* | `(` *variable initializer* `)`

initializer = *expression*

r4rs-lambda-formals = `(` *variable** `)` | `(` *variable*`+ .` *variable* `)` | *variable*

r4rs-define-formals = *variable** | *variable** `.` *variable*

These forms are extended versions of the `lambda` and `define` special forms of standard Scheme. They allow the use of optional and keyword formal arguments with the syntax and semantics of the DSSSL standard.

When the procedure introduced by a `lambda` (or `define`) is applied to a list of actual arguments, the formal and actual arguments are processed as specified in the

R4RS if the *lambda-formals* (or *define-formals*) is a *r4rs-lambda-formals* (or *r4rs-define-formals*), otherwise they are processed as specified in the DSSSL language standard:

a. *Variable*s in *required-formal-argument*s 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-argument*s.

b. Next *variable*s in *optional-formal-argument*s are bound to remaining actual arguments. If there are fewer remaining actual arguments than *optional-formal-argument*s, then the variables are bound to the result of evaluating *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.

c. If there is a *rest-formal-argument*, then it is bound to a list of all remaining actual arguments. These remaining actual arguments are also eligible to be bound to *keyword-formal-argument*s. If there is no *rest-formal-argument* and there are no *keyword-formal-argument*s, then it shall be an error if there are any remaining actual arguments.

d. 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 to `#f`. The *initializer* is evaluated in an environment in which all previous formal arguments have been bound.

It shall be an error for a *variable* to appear more than once in a *formal-argument-list*.

It is unspecified whether variables receive their value by binding or by assignment. Currently the compiler and interpreter use different methods, which can lead to different semantics if `call-with-current-continuation` is used in an *initializer*. Note that this is irrelevant for DSSSL programs because `call-with-current-continuation` does not exist in DSSSL.

For example:

```
> ((lambda (#!rest x) x) 1 2 3)
(1 2 3)
> (define (f a #!optional b) (list a b))
> (define (g a #!optional (b a) #!key (c (* a b))) (list a b c))
> (define (h a #!rest b #!key c) (list a b c))
> (f 1)
(1 #f)
> (f 1 2)
(1 2)
> (g 3)
```

```
(3 3 9)
> (g 3 4)
(3 4 12)
> (g 3 4 c: 5)
(3 4 5)
> (g 3 4 c: 5 c: 6)
(3 4 5)
> (h 7)
(7 () #f)
> (h 7 c: 8)
(7 (c: 8) 8)
> (h 7 c: 8 z: 9)
(7 (c: 8 z: 9) 8)
```

`(c-declare` *`c-declaration`*`)` [special form]
`(c-initialize` *`c-code`*`)` [special form]
`(c-lambda (`*`type1...`*`)` *`result-type c-name-or-code`*`)` [special form]
`(c-define (`*`variable define-formals`*`) (`*`type1...`*`)` *`result-type c-name scope body`*`)` [special form]
`(c-define-type` *`name type`* `[`*`c-to-scheme scheme-to-c`* `[`*`cleanup`*`]])` [special form]

These special forms are part of the "C-interface" which allows Scheme code to interact with C code. For a complete description of the C-interface see ⟨undefined⟩ [C-interface], page ⟨undefined⟩.

`(define-structure` *`name field...`*`)` [special form]

Record data types similar to Pascal records and C `struct` types can be defined using the `define-structure` special form. The identifier *name* specifies the name of the new data type. The structure name is followed by k identifiers naming each field of the record. The `define-structure` expands into a set of definitions of the following procedures:

- '`make-`*name*' – A k argument procedure which constructs a new record from the value of its k fields.
- '*name*`?`' – A procedure which tests if its single argument is of the given record type.
- '*name*`-`*field*' – For each field, a procedure taking as its single argument a value of the given record type and returning the content of the corresponding field of the record.
- '*name*`-`*field*`-set!`' – For each field, a two argument procedure taking as its first argument a value of the given record type. The second argument gets assigned to the corresponding field of the record and the void object is returned.

Record data types have a The printed representation os a record data type includes the name of the type and the name and value of each field. Record data types can not be read by the `read` procedure.

For example:

```
> (define-structure point x y color)
> (define p (make-point 3 5 'red))
> p
#<point 3 x: 3 y: 5 color: red>
> (point-x p)
3
> (point-color p)
red
> (point-color-set! p 'black)
> p
#<point 3 x: 3 y: 5 color: black>
```

(file-exists? *path*) [procedure]

path must be a string. `file-exists?` returns `#t` if a file by that name exists, and returns `#f` otherwise.

(pretty-print *obj* [*port*]) [procedure]
(pp *obj* [*port*]) [procedure]

`pretty-print` and `pp` are similar to `write` except that the result is nicely formatted. If *obj* is a procedure created by the interpreter or a procedure created by code compiled with the '`-debug`' option, `pp` will display its source code. The argument *readtable* specifies the readtable to use. If it is not specified, the readtable defaults to the current readtable.

(open-input-string *string* [*settings*]) [procedure]
(open-output-string [*settings*]) [procedure]
(get-output-string *port*) [procedure]

These procedures implement string ports and they are compatible with SRFI 6. String ports can be used like normal ports. `open-input-string` returns an input string port which obtains characters from the given string instead of a file. When the port is closed with a call to `close-input-port`, a string containing the characters that were not read is returned. `open-output-string` returns an output string port which accumulates the characters written to it. The procedure `get-output-string` retrieves the characters sent to the output port since it was opened or since the last call to `get-output-string`.

For example:

```
> (let ((i (open-input-string "alice #(1 2)")))
    (let* ((a (read i)) (b (read i)) (c (read i)))
      (list a b c)))
(alice #(1 2) #!eof)
> (let ((o (open-output-string)))
    (write "cloud" o)
    (write (* 3 3) o)
    (pp (get-output-string o))
    (pp (get-output-string o))
    (write o o)
    (pp (get-output-string o))
```

```
    (close-output-port o))
"\"cloud\"9"
""
"#<output-port 2 (string)>"
```

(call-with-input-string *string proc* [*settings*]) [procedure]
(call-with-output-string *proc* [*settings*]) [procedure]

The procedure `call-with-input-string` is similar to `call-with-input-file` except that the characters are obtained from the string *string*. The procedure `call-with-output-string` calls the procedure *proc* with a freshly created string port and returns a string containing all characters output to that port.

For example:

```
> (call-with-input-string
    "(1 2)"
    (lambda (p) (read-char p) (read p)))
1
> (call-with-output-string
    (lambda (p) (write p p)))
"#<output-port 2 (string)>"
```

(with-input-from-string *string thunk* [*settings*]) [procedure]
(with-output-to-string *thunk* [*settings*]) [procedure]

The procedure `with-input-from-string` is similar to `with-input-from-file` except that the characters are obtained from the string *string*. The procedure `with-output-to-string` calls the thunk and returns a string containing all characters output to the current output port.

For example:

```
> (with-input-from-string
    "(1 2) hello"
    (lambda () (read) (read)))
hello
> (with-output-to-string
    (lambda () (write car)))
"#<procedure 2 car>"
```

(with-input-from-port *port thunk*) [procedure]
(with-output-to-port *port thunk*) [procedure]

These procedures are respectively similar to `with-input-from-file` and `with-output-to-file`. The difference is that the first argument is a port instead of a file name.

(readtable? *obj*) [procedure]

*** This documentation is incomplete!

(current-readtable) [procedure]

Returns the current readtable.

Readtables control the behavior of the reader (i.e. the `read` procedure and the parser used by the `load` procedure and the interpreter and compiler) and the printer (i.e.

the procedures `write`, `display`, `pretty-print`, and `pp`, and the procedure used by the REPL to print results). Both the reader and printer need to know the readtable so that they can preserve write/read invariance. For example a symbol which contains upper case letters will be printed with special escapes if the readtable indicates that the reader is case insensitive.

`(case-conversion?-set!` *`readtable conversion?`*`)` [procedure]
`(keywords-allowed?-set!` *`readtable allowed?`*`)` [procedure]

These procedures configure readtables. The argument *readtable* specifies the readtable to configure.

For the procedure `case-conversion?-set!`, if *conversion?* is `#f`, the reader will preserve the case of the symbols that are read; if *conversion?* is the symbol `upcase`, the reader will convert letters to upper case; otherwise the reader will convert to lower case. The default is to preserve the case.

For the procedure `keywords-allowed?-set!`, if *allowed?* is `#f`, the reader will not recognize keyword objects; if *allowed?* is the symbol `prefix`, the reader will recognize keyword objects that start with a colon (as in Common Lisp); otherwise the reader will recognize keyword objects that end with a colon (as in DSSSL). The default is to recognize keyword objects that end in a colon.

For example:

```
> (case-conversion?-set! #f)
> 'TeX
TeX
> (case-conversion?-set! #t)
> 'TeX
tex
> (keywords-allowed?-set! #f)
> (symbol? 'foo:)
#t
> (keywords-allowed?-set! #t)
> (keyword? 'foo:) ; quote not really neededa
#t
> (keywords-allowed?-set! 'prefix)
> (keyword? ':foo) ; quote not really needed
#t
```

`(keyword?` *`obj`*`)` [procedure]
`(keyword->string` *`keyword`*`)` [procedure]
`(string->keyword` *`string`*`)` [procedure]

These procedures implement the *keyword* data type. Keywords are similar to symbols but are self evaluating and distinct from the symbol data type. A keyword is an identifier immediately followed by a colon (or preceded by a colon if `(set-keywords-allowed! 'prefix)` was called). The procedure `keyword?` returns `#t` if *obj* is a keyword, and otherwise returns `#f`. The procedure `keyword->string` returns the name of *keyword* as a string, excluding the colon. The procedure `string->keyword` returns the keyword whose name is *string* (the name does not include the colon).

For example:

```
> (keyword? 'color)
#f
> (keyword? color:)
#t
> (keyword->string color:)
"color"
> (string->keyword "color")
color:
```

(gc-report-set! *report?*) [procedure]

gc-report-set! controls the generation of reports during garbage collections. If the argument is true, a brief report of memory usage is generated after every garbage collection. It contains: the time taken for this garbage collection, the amount of memory allocated in kilobytes since the program was started, the size of the heap in kilobytes, the heap memory in kilobytes occupied by live data, the proportion of the heap occupied by live data, and the number of bytes occupied by movable and nonmovable objects.

(with-exception-catcher *handler thunk*) [procedure]
(with-exception-handler *handler thunk*) [procedure]

(box? *obj*) [procedure]
(box *obj*) [procedure]
(unbox *box*) [procedure]
(set-box! *box val*) [procedure]

*** This documentation is incomplete!

(make-will *testator action*) [procedure]
(will? *obj*) [procedure]
(will-testator *will*) [procedure]
(will-execute! *will*) [procedure]

These procedures implement the *will* data type. Will objects provide support for finalization. A will is an object that contains a reference to a *testator* object (the object attached to the will), and an *action* procedure which is a one parameter procedure which is called when the will is executed.

make-will creates a will object with the given *testator* object and *action* procedure. will? tests if *obj* is a will object. will-testator gets the testator object attached to the *will*. will-execute! executes *will*.

An object is *finalizable* if all paths to the object from the roots (i.e. continuations of runnable threads, global variables, etc) pass through a will object. Note that by this definition an object that is not reachable at all from the roots is finalizable. Some objects, including symbols, small integers (fixnums), booleans and characters, are considered to be always reachable and are therefore never finalizable.

When the runtime system detects that a will's testator "T" is finalizable the current computation is interrupted, the will's testator is set to #f and the will's action procedure is called with "T" as the sole argument. Currently only the garbage collector detects when objects become finalizable but this may change in future versions of Gambit (for example the compiler could perform an analysis to infer finalizability

at compile time). The garbage collector builds a list of all wills whose testators are finalizable. Shortly after a garbage collection, the action procedures of these wills will be called. The link from the will to the action procedure is severed when the action procedure is called.

Note that the testator object will not be reclaimed during the garbage collection that detected finalizability of the testator object. It is only when an object is not reachable from the roots (not even through will objects) that it is reclaimed by the garbage collector.

A remarkable feature of wills is that an action procedure can "resurrect" an object after it has become finalizable, by making it nonfinalizable. An action procedure could for example assign the testator object to a global variable.

For example:

```
> (define a (list 123))
> (set-cdr! a a) ; create a circular list
> (define b (vector a))
> (define c #f)
> (define w
    (let ((obj a))
      (make-will obj
                 (lambda (x) ; x will be eq? to obj
                   (display "executing action procedure")
                   (newline)
                   (set! c x)))))
> (will? w)
#t
> (car (will-testator w))
123
> (##gc)
> (set! a #f)
> (##gc)
> (set! b #f)
> (##gc)
executing action procedure
> (will-testator w)
#f
> (car c)
123
```

(gensym [*prefix*]) [procedure]

`gensym` returns a new *uninterned* symbol. Uninterned symbols are guaranteed to be distinct from the symbols generated by the procedures `read` and `string->symbol`. The symbol *prefix* is the prefix used to generate the new symbol's name. If it is not specified, the prefix defaults to '`g`'.

For example:

```
> (gensym)
#:g0
```

```
> (gensym)
#:g1
> (gensym 'star-trek-)
#:star-trek-2
```

(void) [procedure]

`void` returns the void object. The read-eval-print loop prints nothing when the result is the void object.

(eval *expr* [*env*]) [procedure]

`eval`'s first argument is a datum representing an expression. `eval` evaluates this expression in the global interaction environment and returns the result. If present, the second argument is ignored (it is provided for compatibility with R5RS).

For example:

```
> (eval '(+ 1 2))
3
> ((eval 'car) '(1 2))
1
> (eval '(define x 5))
> x
5
```

(error *string obj...*) [procedure]

`error` signals an error and causes a nested REPL to be started. The error message displayed is *string* followed by the remaining arguments. The continuation of the REPL is the same as the one passed to `error`. Thus, returning from the REPL with the '`,c`' or '`,(c` *expr*`)`' command causes a return from the call to `error`.

For example:

```
> (define (f x)
    (let ((y (if (> x 0) (log x) (error "x must be positive"))))
      (+ y 1)))
> (+ (f -4) 10)
*** ERROR IN (stdin)@2.34 -- x must be positive
1> ,(c 5)
16
```

(time *expr*) [special form]

`time` evaluates *expr* and returns the result. As a side effect it displays a message which indicates how long the evaluation took (in real time and cpu time), how much time was spent in the garbage collector, how much memory was allocated during the evaluation and how many minor and major page faults occured (0 is reported if not running under UNIX).

For example:

```
> (define (f x)
    (let loop ((x x) (lst '()))
      (if (= x 0)
          lst
```

```
          (loop (- x 1) (cons x lst)))))
> (length (time (f 100000)))
(time (f 100000))
    266 ms real time
    260 ms cpu time (260 user, 0 system)
    8 collections accounting for 41 ms real time (30 user, 0 system)
    6400136 bytes allocated
    859 minor faults
    no major faults
100000
```

13.3 Unstable additions

This section contains additional special forms and procedures which are documented only in the interest of experimentation. They may be modified or removed in future releases of Gambit. The procedures in this section do not check the type of their arguments so they may cause the program to crash if called improperly.

`(##gc)` [procedure]

The procedure `##gc` forces a garbage collection of the heap.

`(##add-gc-interrupt-job` *`thunk`*`)` [procedure]
`(##clear-gc-interrupt-jobs)` [procedure]

Using the procedure `##add-gc-interrupt-job` it is possible to add a thunk that is called at the end of every garbage collection. The procedure `##clear-gc-interrupt-jobs` removes all the thunks added with `##add-gc-interrupt-job`.

`(##add-timer-interrupt-job` *`thunk`*`)` [procedure]
`(##clear-timer-interrupt-jobs)` [procedure]

The runtime system sets up a free running timer that raises an interrupt at 10 Hz or faster. Using the procedure `##add-timer-interrupt-job` it is possible to add a thunk that is called every time a timer interrupt is received. The procedure `##clear-timer-interrupt-jobs` removes all the thunks added with `##add-timer-interrupt-job`. It is relatively easy to implement threads by using these procedures in conjunction with `call-with-current-continuation`.

`(cond-expand` *`cond-expand-clause...`*`)` [special form]

*** This documentation is incomplete!

`(parameterize ((`*`parameter value`*`)...)` *`body`*`)` [special form]
`(make-parameter` *`obj`* [*`filter`*]`)` [procedure]

*** This documentation is incomplete!

`(s8vector?` *`obj`*`)` [procedure]
`(make-s8vector` *`k`* [*`fill`*]`)` [procedure]
`(s8vector` *`exact-int8...`*`)` [procedure]
`(s8vector-length` *`s8vector`*`)` [procedure]
`(s8vector-ref` *`s8vector k`*`)` [procedure]
`(s8vector-set!` *`s8vector k exact-int8`*`)` [procedure]

(s8vector->list *s8vector*) [procedure]
(list->s8vector *list-of-exact-int8*) [procedure]
(s8vector-fill! *s8vector fill*) [procedure]
(s8vector-copy *s8vector*) [procedure]
(subs8vector *s8vector start end*) [procedure]

(u8vector? *obj*) [procedure]
(make-u8vector *k* [*fill*]) [procedure]
(u8vector *exact-int8...*) [procedure]
(u8vector-length *u8vector*) [procedure]
(u8vector-ref *u8vector k*) [procedure]
(u8vector-set! *u8vector k exact-int8*) [procedure]
(u8vector->list *u8vector*) [procedure]
(list->u8vector *list-of-exact-int8*) [procedure]
(u8vector-fill! *u8vector fill*) [procedure]
(u8vector-copy *u8vector*) [procedure]
(subu8vector *u8vector start end*) [procedure]

(s16vector? *obj*) [procedure]
(make-s16vector *k* [*fill*]) [procedure]
(s16vector *exact-int16...*) [procedure]
(s16vector-length *s16vector*) [procedure]
(s16vector-ref *s16vector k*) [procedure]
(s16vector-set! *s16vector k exact-int16*) [procedure]
(s16vector->list *s16vector*) [procedure]
(list->s16vector *list-of-exact-int16*) [procedure]
(s16vector-fill! *s16vector fill*) [procedure]
(s16vector-copy *s16vector*) [procedure]
(subs16vector *s16vector start end*) [procedure]

(u16vector? *obj*) [procedure]
(make-u16vector *k* [*fill*]) [procedure]
(u16vector *exact-int16...*) [procedure]
(u16vector-length *u16vector*) [procedure]
(u16vector-ref *u16vector k*) [procedure]
(u16vector-set! *u16vector k exact-int16*) [procedure]
(u16vector->list *u16vector*) [procedure]
(list->u16vector *list-of-exact-int16*) [procedure]
(u16vector-fill! *u16vector fill*) [procedure]
(u16vector-copy *u16vector*) [procedure]
(subu16vector *u16vector start end*) [procedure]

(s32vector? *obj*) [procedure]
(make-s32vector *k* [*fill*]) [procedure]
(s32vector *exact-int32...*) [procedure]
(s32vector-length *s32vector*) [procedure]
(s32vector-ref *s32vector k*) [procedure]
(s32vector-set! *s32vector k exact-int32*) [procedure]
(s32vector->list *s32vector*) [procedure]

(list->s32vector *list-of-exact-int32*) [procedure]
(s32vector-fill! *s32vector fill*) [procedure]
(s32vector-copy *s32vector*) [procedure]
(subs32vector *s32vector start end*) [procedure]

(u32vector? *obj*) [procedure]
(make-u32vector *k* [*fill*]) [procedure]
(u32vector *exact-int32...*) [procedure]
(u32vector-length *u32vector*) [procedure]
(u32vector-ref *u32vector k*) [procedure]
(u32vector-set! *u32vector k exact-int32*) [procedure]
(u32vector->list *u32vector*) [procedure]
(list->u32vector *list-of-exact-int32*) [procedure]
(u32vector-fill! *u32vector fill*) [procedure]
(u32vector-copy *u32vector*) [procedure]
(subu32vector *u32vector start end*) [procedure]

(s64vector? *obj*) [procedure]
(make-s64vector *k* [*fill*]) [procedure]
(s64vector *exact-int64...*) [procedure]
(s64vector-length *s64vector*) [procedure]
(s64vector-ref *s64vector k*) [procedure]
(s64vector-set! *s64vector k exact-int64*) [procedure]
(s64vector->list *s64vector*) [procedure]
(list->s64vector *list-of-exact-int64*) [procedure]
(s64vector-fill! *s64vector fill*) [procedure]
(s64vector-copy *s64vector*) [procedure]
(subs64vector *s64vector start end*) [procedure]

(u64vector? *obj*) [procedure]
(make-u64vector *k* [*fill*]) [procedure]
(u64vector *exact-int64...*) [procedure]
(u64vector-length *u64vector*) [procedure]
(u64vector-ref *u64vector k*) [procedure]
(u64vector-set! *u64vector k exact-int64*) [procedure]
(u64vector->list *u64vector*) [procedure]
(list->u64vector *list-of-exact-int64*) [procedure]
(u64vector-fill! *u64vector fill*) [procedure]
(u64vector-copy *u64vector*) [procedure]
(subu64vector *u64vector start end*) [procedure]

(f32vector? *obj*) [procedure]
(make-f32vector *k* [*fill*]) [procedure]
(f32vector *inexact-real...*) [procedure]
(f32vector-length *f32vector*) [procedure]
(f32vector-ref *f32vector k*) [procedure]
(f32vector-set! *f32vector k inexact-real*) [procedure]
(f32vector->list *f32vector*) [procedure]
(list->f32vector *list-of-inexact-real*) [procedure]

(f32vector-fill! *f32vector fill*) [procedure]
(f32vector-copy *f32vector*) [procedure]
(subf32vector *f32vector start end*) [procedure]

(f64vector? *obj*) [procedure]
(make-f64vector *k* [*fill*]) [procedure]
(f64vector *inexact-real...*) [procedure]
(f64vector-length *f64vector*) [procedure]
(f64vector-ref *f64vector k*) [procedure]
(f64vector-set! *f64vector k inexact-real*) [procedure]
(f64vector->list *f64vector*) [procedure]
(list->f64vector *list-of-inexact-real*) [procedure]
(f64vector-fill! *f64vector fill*) [procedure]
(f64vector-copy *f64vector*) [procedure]
(subf64vector *f64vector start end*) [procedure]

Bytevectors are uniform vectors containing raw numbers (signed or unsigned exact integers or inexact reals). There are 10 types of bytevectors: '`s8vector`' (vector of exact integers in the range -2^7 to 2^7-1), '`u8vector`' (vector of exact integers in the range 0 to 2^8-1), '`s16vector`' (vector of exact integers in the range -2^15 to 2^15-1), '`u16vector`' (vector of exact integers in the range 0 to 2^16-1), '`s32vector`' (vector of exact integers in the range -2^31 to 2^31-1), '`u32vector`' (vector of exact integers in the range 0 to 2^32-1), '`s64vector`' (vector of exact integers in the range -2^63 to 2^63-1), '`u64vector`' (vector of exact integers in the range 0 to 2^64-1), '`f32vector`' (vector of 32 bit floating point numbers), and '`f64vector`' (vector of 64 bit floating point numbers). These procedures are the analog of the normal vector procedures for each of the bytevector types.

For example:

```
> (define v (u8vector 10 255 13))
> (u8vector-set! v 2 99)
> v
#u8(10 255 99)
> (u8vector-ref v 1)
255
> (u8vector->list v)
(10 255 99)
```

13.4 Other extensions

Gambit supports the Unicode character encoding standard (ISO/IEC-10646-1). Scheme characters can be any of the characters in the 16 bit subset of Unicode known as UCS-2. Scheme strings can contain any character in UCS-2. Source code can also contain any character in UCS-2. However, to read such source code properly `gsi` and `gsc` must be told which character encoding to use for reading the source code (i.e. UTF-8, UCS-2, or UCS-4). This can be done by passing a character encoding parameter to `load` or by specifying the runtime option '`-:c`' when `gsi` and `gsc` are started.

14 Threads

Gambit supports the execution of multiple Scheme threads. These threads are managed entirely by Gambit's runtime and are not related to the host operating system's threads. Gambit's runtime does not currently take advantage of multiprocessors (i.e. at most one thread is running).

14.1 Introduction

Multithreading is a paradigm that is well suited for building complex systems such as: servers, GUIs, and high-level operating systems. Gambit's thread system offers mechanisms for creating new threads of execution and for synchronizing them. The thread system also supports features which are useful in a real-time context, such as priorities, priority inheritance and a time datatype (which is useful on its own and also for specifying synchronization timeouts). Mechanisms to handle exceptions and some multithreading exception datatypes are explained in this chapter because the handling of exceptions is closely tied to the multithreading model.

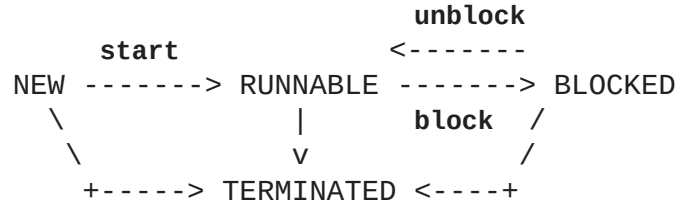
The thread system provides the following data types:

- Thread (a virtual processor which shares object space with all other threads)
- Thread group (a collection of threads)
- Mutex (a mutual exclusion device, also known as a lock and binary semaphore)
- Condition variable (a set of blocked threads)
- Time (an absolute point on the time line)

Some multithreading exception datatypes are also specified, and a general mechanism for handling exceptions.

14.2 Threads

A *running thread* is a thread that is currently executing. There can be more than one running thread on a multiprocessor machine. A *runnable thread* is a thread that is ready to execute or running. A thread is *blocked* if it is waiting for a mutex to become unlocked, an I/O operation to become possible, the end of a "sleep" period, etc. A *new thread* is a thread that has not yet become runnable. A new thread becomes runnable when it is started. A *terminated thread* is a thread that can no longer become runnable (but *deadlocked threads* are not considered terminated). The only valid transitions between the thread states are from new to runnable, between runnable and blocked, and from any state to terminated as indicated in the following diagram:



Each thread has a *base priority*, which is a real number (where a higher numerical value means a higher priority), a *priority boost*, which is a non-negative real number representing

the priority increase applied to a thread when it blocks, and a *quantum*, which is a non-negative real number representing a duration in seconds.

Each thread has a *specific field* which can be used in an application specific way to associate data with the thread (some thread systems call this "thread local storage").

14.3 Mutexes

A mutex can be in one of four states: *locked* (either *owned* or *not owned*) and *unlocked* (either *abandoned* or *not abandoned*).

An attempt to lock a mutex only succeeds if the mutex is in an unlocked state, otherwise the current thread will wait. A mutex in the locked/owned state has an associated *owner thread*, which by convention is the thread that is responsible for unlocking the mutex (this case is typical of critical sections implemented as "lock mutex, perform operation, unlock mutex"). A mutex in the locked/not-owned state is not linked to a particular thread.

A mutex becomes locked when a thread locks it using the '`mutex-lock!`' primitive. A mutex becomes unlocked/abandoned when the owner of a locked/owned mutex terminates. A mutex becomes unlocked/not-abandoned when a thread unlocks it using the '`mutex-unlock!`' primitive.

The mutex primitives do not implement *recursive mutex semantics*. An attempt to lock a mutex that is locked implies that the current thread waits even if the mutex is owned by the current thread (this can lead to a deadlock if no other thread unlocks the mutex).

Each mutex has a *specific field* which can be used in an application specific way to associate data with the mutex.

14.4 Condition variables

A condition variable represents a set of blocked threads. These blocked threads are waiting for a certain condition to become true. When a thread modifies some program state that might make the condition true, the thread unblocks some number of threads (one or all depending on the primitive used) so they can check if the condition is now true. This allows complex forms of interthread synchronization to be expressed more conveniently than with mutexes alone.

Each condition variable has a *specific field* which can be used in an application specific way to associate data with the condition variable.

14.5 Fairness

In various situations the scheduler must select one thread from a set of threads (e.g. which thread to run when a running thread blocks or expires its quantum, which thread to unblock when a mutex becomes unlocked or a condition variable is signaled). The constraints on the selection process determine the scheduler's *fairness*. The selection depends on the order in which threads become runnable or blocked and on the *priority* attached to the threads.

The definition of fairness requires the notion of time ordering, i.e. "event A occured before event B". For the purpose of establishing time ordering, the scheduler uses a clock with a discrete, usually variable, resolution (a "tick"). Events occuring in a given tick can be considered to be simultaneous (i.e. if event A occured before event B in real time, then

the scheduler will claim that event A occured before event B unless both events fall within the same tick, in which case the scheduler arbitrarily chooses a time ordering).

Each thread T has three priorities which affect fairness; the *base priority*, the *boosted priority*, and the *effective priority*.

- The *base priority* is the value contained in T's *base priority* field (which is set with the '`thread-base-priority-set!`' primitive).
- T's *boosted flag* field contains a boolean that affects T's *boosted priority*. When the boosted flag field is false, the boosted priority is equal to the base priority, otherwise the boosted priority is equal to the base priority plus the value contained in T's *priority boost* field (which is set with the '`thread-priority-boost-set!`' primitive). The boosted flag field is set to false when a thread is created, when its quantum expires, and when *thread-yield!* is called. The boosted flag field is set to true when a thread blocks. By carefully choosing the base priority and priority boost, relatively to the other threads, it is possible to set up an interactive thread so that it has good I/O response time without being a CPU hog when it performs long computations.
- The *effective priority* is equal to the maximum of T's boosted priority and the effective priority of all the threads that are blocked on a mutex owned by T. This *priority inheritance* avoids priority inversion problems that would prevent a high priority thread blocked at the entry of a critical section to progress because a low priority thread inside the critical section is preempted for an arbitrary long time by a medium priority thread.

Let $P(T)$ be the effective priority of thread T and let $R(T)$ be the most recent time when one of the following events occurred for thread T, thus making it runnable: T was started by calling '`thread-start!`', T called '`thread-yield!`', T expired its quantum, or T became unblocked. Let the relation $NL(T1,T2)$, "*T1* no later than *T2*", be true if $P(T1)<P(T2)$ or $P(T1)=P(T2)$ and $R(T1)>R(T2)$, and false otherwise. The scheduler will schedule the execution of threads in such a way that whenever there is at least one runnable thread, 1) within a finite time at least one thread will be running, and 2) there is never a pair of runnable threads *T1* and *T2* for which $NL(T1,T2)$ is true and *T1* is not running and *T2* is running.

A thread T expires its quantum when an amount of time equal to T's quantum has elapsed since T entered the running state and T did not block, terminate or call '`thread-yield!`'. At that point T exits the running state to allow other threads to run. A thread's quantum is thus an indication of the rate of progress of the thread relative to the other threads of the same priority. Moreover, the resolution of the timer measuring the running time may cause a certain deviation from the quantum, so a thread's quantum should only be viewed as an approximation of the time it can run before yielding to another thread.

Threads blocked on a given mutex or condition variable will unblock in an order which is consistent with decreasing priority and increasing blocking time (i.e. the highest priority thread unblocks first, and among equal priority threads the one that blocked first unblocks first).

14.6 Memory coherency and lack of atomicity**********

Read and write operations on the store (such as reading and writing a variable, an element of a vector or a string) are not required to be atomic. It is an error for a thread to write

a location in the store while some other thread reads or writes that same location. It is the responsibility of the application to avoid write/read and write/write races through appropriate uses of the synchronization primitives.

Concurrent reads and writes to ports are allowed. It is the responsibility of the implementation to serialize accesses to a given port using the appropriate synchronization primitives.

14.7 Dynamic environments, continuations and ‘dynamic-wind’

The "dynamic environment" is a structure which allows the system to find the value returned by ‘current-input-port’, ‘current-output-port’, etc. The procedures ‘with-input-from-file’, ‘with-output-to-file’, etc extend the dynamic environment to produce a new dynamic environment which is in effect for the duration of the call to the thunk passed as the last argument. Some Scheme systems generalize the dynamic environment by providing procedures and special forms to define new "dynamic variables" and bind them in the dynamic environment (e.g. ‘make-parameter’ and ‘parameterize’).

Each thread has its own dynamic environment. When a thread's dynamic environment is extended this does not affect the dynamic environment of other threads. When a thread creates a continuation, the thread's dynamic environment and the ‘dynamic-wind’ stack are saved within the continuation (an alternate but equivalent point of view is that the ‘dynamic-wind’ stack is part of the dynamic environment). When this continuation is invoked the required ‘dynamic-wind’ before and after thunks are called and the saved dynamic environment is reinstated as the dynamic environment of the current thread. During the call to each required ‘dynamic-wind’ before and after thunk, the dynamic environment and the ‘dynamic-wind’ stack in effect when the corresponding ‘dynamic-wind’ was executed are reinstated. Note that this specification clearly defines the semantics of calling ‘call-with-current-continuation’ or invoking a continuation within a before or after thunk. The semantics are well defined even when a continuation created by another thread is invoked. Below is an example exercising the subtleties of this semantics.

```
(with-output-to-file
 "foo"
 (lambda ()
   (let ((k (call-with-current-continuation
             (lambda (exit)
               (with-output-to-file
                "bar"
                (lambda ()
                  (dynamic-wind
                   (lambda () (write '(b1)))
                   (lambda ()
                     (let ((x (call-with-current-continuation
                               (lambda (cont) (exit cont)))))
                       (write '(t1))
                       x))
                   (lambda () (write '(a1)))))))))))
     (if k
```

```
(dynamic-wind
 (lambda () (write '(b2)))
 (lambda ()
   (with-output-to-file
    "baz"
    (lambda ()
      (write '(t2))
      ; go back inside (with-output-to-file "bar" ...)
      (k #f))))
 (lambda () (write '(a2)))))))
```

In an implementation of Scheme where '`with-output-to-file`' only closes the port it opened when the thunk returns normally, then the following actions will occur: `(b1)(a1)` is written to "bar", `(b2)` is written to "foo", `(t2)` is written to "baz", `(a2)` is written to "foo", and `(b1)(t1)(a1)` is written to "bar".

When the scheduler stops the execution of a running thread T1 (whether because it blocked, expired its quantum, was terminated, etc) and then resumes the execution of a thread T2, there is in a sense a transfer of control between T1's current continuation and the continuation of T2. This transfer of control by the scheduler does not cause any '`dynamic-wind`' before and after thunks to be called. It is only when a thread itself transfers control to a continuation that '`dynamic-wind`' before and after thunks are called.

14.8 Time objects and timeouts

A time object represents a point on the time line. Its resolution is implementation dependent (implementations are encouraged to implement at least millisecond resolution so that precise timing is possible). Using `time->seconds` and `seconds->time`, a time object can be converted to and from a real number which corresponds to the number of seconds from a reference point on the time line. The reference point is implementation dependent and does not change for a given execution of the program (e.g. the reference point could be the time at which the program started).

All synchronization primitives which take a timeout parameter accept three types of values as a timeout, with the following meaning:

- a time object represents an absolute point in time
- an exact or inexact real number represents a relative time in seconds from the moment the primitive was called
- '`#f`' means that there is no timeout

When a timeout denotes the current time or a time in the past, the synchronization primitive claims that the timeout has been reached only after the other synchronization conditions have been checked. Moreover the thread remains running (it does not enter the blocked state). For example, `(mutex-lock! m 0)` will lock mutex `m` and return '`#t`' if `m` is currently unlocked, otherwise '`#f`' is returned because the timeout is reached.

14.9 Primitives and exceptions

When one of the primitives defined in this SRFI raises an exception defined in this SRFI, the exception handler is called with the same continuation as the primitive

(i.e. it is a tail call to the exception handler). This requirement avoids having to use '`call-with-current-continuation`' to get the same effect in some situations.

14.10 Primordial thread

The execution of a program is initially under the control of a single thread known as the "primordial thread". The primordial thread has an unspecified base priority, priority boost, boosted flag, quantum, name, specific field, dynamic environment, '`dynamic-wind`' stack, and exception handler. All threads are terminated when the primordial thread terminates (normally or not).

14.11 Procedures

`(current-thread)` [procedure]

Returns the current thread. For example:

```
> (current-thread)
#<thread #1 primordial>
> (eq? (current-thread) (current-thread))
#t
```

`(thread? obj)` [procedure]

Returns '`#t`' if *`obj`* is a thread, otherwise returns '`#f`'. If any of the predicates listed in Section 3.2 of the R5RS is true of *`obj`*, then `thread?` is false of *`obj`*.

For example:

```
> (thread? (current-thread))
#t
> (thread? 'foo)
#f
```

`(make-thread` *`thunk`* [*`name`* [*`thread-group`*]]`)` [procedure]

Returns a new thread. This thread is not automatically made runnable (the procedure `thread-start!` must be used for this). A thread has the following fields: base priority, priority boost, boosted flag, quantum, name, specific, end-result, end-exception, and a list of locked/owned mutexes it owns. The thread's execution consists of a call to *thunk* with the "initial continuation". This continuation causes the (then) current thread to store the result in its end-result field, abandon all mutexes it owns, and finally terminate. The '`dynamic-wind`' stack of the initial continuation is empty. The optional *`name`* is an arbitrary Scheme object which identifies the thread (useful for debugging); it defaults to an unspecified value. The specific field is set to an unspecified value. The optional *`thread-group`* indicates which thread group this thread belongs to; it defaults to the thread group of the current thread. The base priority, priority boost, and quantum of the thread are set to the same value as the current thread and the boosted flag is set to false. The thread inherits the dynamic environment from the current thread. Moreover, in this dynamic environment the exception handler is bound to the "initial exception handler" which is a unary procedure which causes the (then) current thread to store in its end-exception field an "uncaught exception" object whose "reason" is the argument of the handler, abandon all mutexes it owns, and finally terminate.

For example:

```
> (make-thread (lambda () (write 'hello)))
#<thread #2>
> (make-thread (lambda () (write 'world)) 'a-name)
#<thread #3 a-name>
```

(thread-name *thread*) [procedure]

Returns the name of the *thread*.

For example:

```
> (thread-name (make-thread (lambda () #f) 'foo))
foo
```

(thread-specific *thread*) [procedure]

Returns the content of the *thread*'s specific field.

(thread-specific-set! *thread obj*) [procedure]

Stores *obj* into the *thread*'s specific field. thread-specific-set! returns an unspecified value.

For example:

```
> (thread-specific-set! (current-thread) "hello")
> (thread-specific (current-thread))
"hello"
```

(thread-base-priority *thread*) [procedure]

Returns a real number which corresponds to the base priority of the *thread*.

(thread-base-priority-set! *thread priority*) [procedure]

Changes the base priority of the *thread* to *priority*. The *priority* must be a real number. thread-base-priority-set! returns an unspecified value.

For example:

```
> (thread-base-priority-set! (current-thread) 12.3)
> (thread-base-priority (current-thread))
12.3
```

(thread-priority-boost *thread*) [procedure]

Returns a real number which corresponds to the priority boost of the *thread*.

(thread-priority-boost-set! *thread priority-boost*) [procedure]

Changes the priority boost of the *thread* to *priority-boost*. The *priority-boost* must be a non-negative real. thread-priority-boost-set! returns an unspecified value.

For example:

```
> (thread-priority-boost-set! (current-thread) 2.5)
> (thread-priority-boost (current-thread))
2.5
```

(thread-quantum *thread*) [procedure]

Returns a real number which corresponds to the quantum of the *thread*.

`(thread-quantum-set! thread quantum)` [procedure]

Changes the quantum of the *thread* to *quantum*. The *quantum* must be a non-negative real. A value of zero selects the smallest quantum supported by the implementation. `thread-quantum-set!` returns an unspecified value.

For example:

```
> (thread-quantum-set! (current-thread) 1.5)
> (thread-quantum (current-thread))
1.5
> (thread-quantum-set! (current-thread) 0)
> (thread-quantum (current-thread))
.01
```

`(thread-start! thread)` [procedure]

Makes *thread* runnable. The *thread* must be a new thread. `thread-start!` returns the *thread*.

For example:

```
> (let ((t (thread-start! (make-thread (lambda () (write 'a))))))
    (write 'b)
    (thread-join! t))
ab> or ba>
```

NOTE: It is useful to separate thread creation and thread activation to avoid the race condition that would occur if the created thread tries to examine a table in which the current thread stores the created thread. See the last example of `thread-terminate!` which contains mutually recursive threads.

`(thread-yield!)` [procedure]

The current thread exits the running state as if its quantum had expired. `thread-yield!` returns an unspecified value.

For example:

```
; a busy loop that avoids being too wasteful of the CPU

(let loop ()
  (if (mutex-lock! m 0) ; try to lock m but don't block
      (begin
        (display "locked mutex m")
        (mutex-unlock! m))
      (begin
        (do-something-else)
        (thread-yield!) ; relinquish rest of quantum
        (loop))))
```

`(thread-sleep! timeout)` [procedure]

The current thread waits until the timeout is reached. This blocks the thread only if *timeout* represents a point in the future. It is an error for *timeout* to be '`#f`'. `thread-sleep!` returns an unspecified value.

For example:

```
; a clock with a gradual drift:

(let loop ((x 1))
  (thread-sleep! 1)
  (write x)
  (loop (+ x 1)))

; a clock with no drift:

(let ((start (time->seconds (current-time)))
  (let loop ((x 1))
    (thread-sleep! (seconds->time (+ x start)))
    (write x)
    (loop (+ x 1))))
```

`(thread-terminate!` *`thread`*`)` [procedure]

Causes an abnormal termination of the *`thread`*. If the *`thread`* is not already terminated, all mutexes owned by the *`thread`* become unlocked/abandoned and a "terminated thread exception" object is stored in the *`thread`*'s end-exception field. If *`thread`* is the current thread, `thread-terminate!` does not return. Otherwise `thread-terminate!` returns an unspecified value; the termination of the *`thread`* will occur before `thread-terminate!` returns. at some point between the calling of `thread-terminate!` and a finite time in the future (an explicit thread synchronization is needed to detect termination, see `thread-join!`).

For example:

```
(thread-terminate! (current-thread))  ==>  does not return

(define (amb thunk1 thunk2)
  (let ((result #f)
        (result-mutex (make-mutex))
        (done-mutex (make-mutex)))
    (letrec ((child1
              (make-thread
                (lambda ()
                  (let ((x (thunk1)))
                    (mutex-lock! result-mutex #f #f)
                    (set! result x)
                    (thread-terminate! child2)
                    (mutex-unlock! done-mutex)))))
             (child2
              (make-thread
                (lambda ()
                  (let ((x (thunk2)))
                    (mutex-lock! result-mutex #f #f)
                    (set! result x)
                    (thread-terminate! child1)
                    (mutex-unlock! done-mutex))))))
```

```
      (mutex-lock! done-mutex #f #f)
      (thread-start! child1)
      (thread-start! child2)
      (mutex-lock! done-mutex #f #f)
      result)))
```

NOTE: This operation must be used carefully because it terminates a thread abruptly and it is impossible for that thread to perform any kind of cleanup. This may be a problem if the thread is in the middle of a critical section where some structure has been put in an inconsistent state. However, another thread attempting to enter this critical section will raise an "abandoned mutex exception" because the mutex is unlocked/abandoned. This helps avoid observing an inconsistent state. Clean termination can be obtained by polling, as shown in the example below.

For example:

```
(define (spawn thunk)
  (let ((t (make-thread thunk)))
    (thread-specific-set! t #t)
    (thread-start! t)
    t))

(define (stop! thread)
  (thread-specific-set! thread #f)
  (thread-join! thread))

(define (keep-going?)
  (thread-specific (current-thread)))

(define count!
  (let ((m (make-mutex))
        (i 0))
    (lambda ()
      (mutex-lock! m)
      (let ((x (+ i 1)))
        (set! i x)
        (mutex-unlock! m)
        x))))

(define (increment-forever!)
  (let loop () (count!) (if (keep-going?) (loop))))

(let ((t1 (spawn increment-forever!))
      (t2 (spawn increment-forever!)))
  (thread-sleep! 1)
  (stop! t1)
  (stop! t2)
  (count!))  ==>  377290
```

`(thread-join! thread [timeout [timeout-val]])` [procedure]

The current thread waits until the *thread* terminates (normally or not) or until the timeout is reached if *timeout* is supplied. If the timeout is reached, *thread-join!* returns *timeout-val* if it is supplied, otherwise a "join timeout exception" is raised. If the *thread* terminated normally, the content of the end-result field is returned, otherwise the content of the end-exception field is raised.

For example:

```
(let ((t (thread-start! (make-thread (lambda () (expt 2 100))))))
  (do-something-else)
  (thread-join! t))  ==>  1267650600228229401496703205376

(let ((t (thread-start! (make-thread (lambda () (raise 123))))))
  (do-something-else)
  (with-exception-handler
    (lambda (exc)
      (if (uncaught-exception? exc)
          (* 10 (uncaught-exception-reason exc))
          99999))
    (lambda ()
      (+ 1 (thread-join! t)))))  ==>  1231

(define thread-alive?
  (let ((unique (list 'unique)))
    (lambda (thread)
      ; Note: this procedure raises an exception if
      ; the thread terminated abnormally.
      (eq? (thread-join! thread 0 unique) unique))))

(define (wait-for-termination! thread)
  (let ((eh (current-exception-handler)))
    (with-exception-handler
      (lambda (exc)
        (if (not (or (terminated-thread-exception? exc)
                     (uncaught-exception? exc)))
            (eh exc))) ; unexpected exceptions are handled by eh
      (lambda ()
        ; The following call to thread-join! will wait until the
        ; thread terminates.  If the thread terminated normally
        ; thread-join! will return normally.  If the thread
        ; terminated abnormally then one of these two exceptions
        ; is raised by thread-join!:
        ;   - terminated thread exception
        ;   - uncaught exception
        (thread-join! thread)
        #f)))) ; ignore result of thread-join!
```

`(mutex? obj)` [procedure]

Returns '`#t`' if *obj* is a mutex, otherwise returns '`#f`'. If any of the predicates listed in Section 3.2 of the R5RS is true of *obj*, then `mutex?` is false of *obj*.

For example:

```
> (mutex? (make-mutex))
#t
> (mutex? 'foo)
#f
```

`(make-mutex [name])` [procedure]

Returns a new mutex in the unlocked/not-abandoned state. The optional *name* is an arbitrary Scheme object which identifies the mutex (useful for debugging); it defaults to an unspecified value. The mutex's specific field is set to an unspecified value.

For example:

```
> (make-mutex)
#<mutex #2>
> (make-mutex 'foo)
#<mutex #3 foo>
```

`(mutex-name mutex)` [procedure]

Returns the name of the *mutex*. For example:

```
> (mutex-name (make-mutex 'foo))
foo
```

`(mutex-specific mutex)` [procedure]

Returns the content of the *mutex*'s specific field.

`(mutex-specific-set! mutex obj)` [procedure]

Stores *obj* into the *mutex*'s specific field. `mutex-specific-set!` returns an unspecified value.

For example:

```
(define m (make-mutex))
(mutex-specific-set! m "hello")  ==>  unspecified
(mutex-specific m)               ==>  "hello"

(define (mutex-lock-recursively! mutex)
  (if (eq? (mutex-state mutex) (current-thread))
      (let ((n (mutex-specific mutex)))
        (mutex-specific-set! mutex (+ n 1)))
      (begin
        (mutex-lock! mutex)
        (mutex-specific-set! mutex 0))))

(define (mutex-unlock-recursively! mutex)
  (let ((n (mutex-specific mutex)))
    (if (= n 0)
        (mutex-unlock! mutex)
        (mutex-specific-set! mutex (- n 1)))))
```

`(mutex-state` *`mutex`*`)` [procedure]

Returns information about the state of the *`mutex`*. The possible results are:

- **`thread`** *`T`*: the *`mutex`* is in the locked/owned state and thread T is the owner of the *`mutex`*
- **`symbol`** `not-owned`: the *`mutex`* is in the locked/not-owned state
- **`symbol`** `abandoned`: the *`mutex`* is in the unlocked/abandoned state
- **`symbol`** `not-abandoned`: the *`mutex`* is in the unlocked/not-abandoned state

For example:

```
(mutex-state (make-mutex))  ==>  not-abandoned

(define (thread-alive? thread)
  (let ((mutex (make-mutex)))
    (mutex-lock! mutex #f thread)
    (let ((state (mutex-state mutex)))
      (mutex-unlock! mutex) ; avoid space leak
      (eq? state thread))))
```

`(mutex-lock!` *`mutex`* [*`timeout`* [*`thread`*]]`)` [procedure]

If the *`mutex`* is currently locked, the current thread waits until the *`mutex`* is unlocked, or until the timeout is reached if *`timeout`* is supplied. If the timeout is reached, `mutex-lock!` returns '`#f`'. Otherwise, the state of the *`mutex`* is changed as follows:

- if *`thread`* is '`#f`' the *`mutex`* becomes locked/not-owned,
- otherwise, let T be *`thread`* (or the current thread if *`thread`* is not supplied),
  - if T is terminated the *`mutex`* becomes unlocked/abandoned,
  - otherwise *`mutex`* becomes locked/owned with T as the owner.

After changing the state of the *`mutex`*, an "abandoned mutex exception" is raised if the *`mutex`* was unlocked/abandoned before the state change, otherwise `mutex-lock!` returns '`#t`'. It is not an error if the *`mutex`* is owned by the current thread (but the current thread will have to wait).

For example:

```
; an implementation of a mailbox object of depth one; this
; implementation does not behave well in the presence of forced
; thread terminations using thread-terminate! (deadlock can occur
; if a thread is terminated in the middle of a put! or get! operati

(define (make-empty-mailbox)
  (let ((put-mutex (make-mutex)) ; allow put! operation
        (get-mutex (make-mutex))
        (cell #f))

    (define (put! obj)
      (mutex-lock! put-mutex #f #f) ; prevent put! operation
      (set! cell obj)
      (mutex-unlock! get-mutex)) ; allow get! operation
```

```
    (define (get!)
      (mutex-lock! get-mutex #f #f) ; wait until object in mailbox
      (let ((result cell))
        (set! cell #f) ; prevent space leaks
        (mutex-unlock! put-mutex) ; allow put! operation
        result))

    (mutex-lock! get-mutex #f #f) ; prevent get! operation

    (lambda (msg)
      (case msg
        ((put!) put!)
        ((get!) get!)
        (else (error "unknown message"))))))

(define (mailbox-put! m obj) ((m 'put!) obj))
(define (mailbox-get! m) ((m 'get!)))

; an alternate implementation of thread-sleep!

(define (sleep! timeout)
  (let ((m (make-mutex)))
    (mutex-lock! m #f #f)
    (mutex-lock! m timeout #f)))

; a procedure that waits for one of two mutexes to unlock

(define (lock-one-of! mutex1 mutex2)
  ; this procedure assumes that neither mutex1 or mutex2
  ; are owned by the current thread
  (let ((ct (current-thread))
        (done-mutex (make-mutex)))
    (mutex-lock! done-mutex #f #f)
    (let ((t1 (thread-start!
               (make-thread
                (lambda ()
                  (mutex-lock! mutex1 #f ct)
                  (mutex-unlock! done-mutex)))))
          (t2 (thread-start!
               (make-thread
                (lambda ()
                  (mutex-lock! mutex2 #f ct)
                  (mutex-unlock! done-mutex))))))
      (mutex-lock! done-mutex #f #f)
      (thread-terminate! t1)
      (thread-terminate! t2)
```

```
(if (eq? (mutex-state mutex1) ct)
    (begin
      (if (eq? (mutex-state mutex2) ct)
          (mutex-unlock! mutex2)) ; don't lock both
      mutex1)
    mutex2))))
```

(mutex-unlock! *mutex* [*condition-variable* [*timeout*]]) [procedure]

Unlocks the *mutex* by making it unlocked/not-abandoned. It is not an error to unlock an unlocked mutex and a mutex that is owned by any thread. If *condition-variable* is supplied, the current thread is blocked and added to the *condition-variable* before unlocking *mutex*; the thread can unblock at any time but no later than when an appropriate call to `condition-variable-signal!` or `condition-variable-broadcast!` is performed (see below), and no later than the timeout (if *timeout* is supplied). If there are threads waiting to lock this *mutex*, the scheduler selects a thread, the mutex becomes locked/owned or locked/not-owned, and the thread is unblocked. `mutex-unlock!` returns '`#f`' when the timeout is reached, otherwise it returns '`#t`'.

NOTE: The reason the thread can unblock at any time (when *condition-variable* is supplied) is to allow extending this SRFI with primitives that force a specific blocked thread to become runnable. For example a primitive to interrupt a thread so that it performs a certain operation, whether the thread is blocked or not, may be useful to handle the case where the scheduler has detected a serious problem (such as a deadlock) and it must unblock one of the threads (such as the primordial thread) so that it can perform some appropriate action. After a thread blocked on a condition-variable has handled such an interrupt it would be wrong for the scheduler to return the thread to the blocked state, because any calls to `condition-variable-broadcast!` during the interrupt will have gone unnoticed. It is necessary for the thread to remain runnable and return from the call to `mutex-unlock!` with a result of '`#t`'.

NOTE: `mutex-unlock!` is related to the "wait" operation on condition variables available in other thread systems. The main difference is that "wait" automatically locks *mutex* just after the thread is unblocked. This operation is not performed by `mutex-unlock!` and so must be done by an explicit call to `mutex-lock!`. This has the advantages that a different timeout and exception handler can be specified on the `mutex-lock!` and `mutex-unlock!` and the location of all the mutex operations is clearly apparent. A typical use with a condition variable is:

For example:

```
(let loop ()
  (mutex-lock! m)
  (if (condition-is-true?)
      (begin
        (do-something-when-condition-is-true)
        (mutex-unlock! m))
      (begin
        (mutex-unlock! m cv)
```

```
                    (loop))))
```

(condition-variable? *obj*) [procedure]

Returns '#t' if *obj* is a condition variable, otherwise returns '#f'. If any of the predicates listed in Section 3.2 of the R5RS is true of *obj*, then condition-variable? is false of *obj*.

For example:

```
> (condition-variable? (make-condition-variable))
#t
> (condition-variable? 'foo)
#f
```

(make-condition-variable [*name*]) [procedure]

Returns a new empty condition variable. The optional *name* is an arbitrary Scheme object which identifies the condition variable (useful for debugging); it defaults to an unspecified value. The condition variable's specific field is set to an unspecified value.

For example:

```
> (make-condition-variable)
an empty condition variable
```

(condition-variable-name *condition-variable*) [procedure]

Returns the name of the *condition-variable*. For example:

```
> (condition-variable-name (make-condition-variable 'foo))
foo
```

(condition-variable-specific *condition-variable*) [procedure]

Returns the content of the *condition-variable*'s specific field.

(condition-variable-specific-set! *condition-variable obj*) [procedure]

Stores *obj* into the *condition-variable*'s specific field. condition-variable-specific-set! returns an unspecified value.

For example:

```
(define cv (make-condition-variable))
(condition-variable-specific-set! cv "hello")  ==>  unspecified
(condition-variable-specific cv)               ==>  "hello"
```

(condition-variable-signal! *condition-variable*) [procedure]

If there are threads blocked on the *condition-variable*, the scheduler selects a thread and unblocks it. condition-variable-signal! returns an unspecified value.

For example:

```
; an implementation of a mailbox object of depth one; this
; implementation behaves gracefully when threads are forcibly
; terminated using thread-terminate! (the "abandoned mutex"
; exception will be raised when a put! or get! operation is attempt
```

```
; after a thread is terminated in the middle of a put! or get!
; operation)

(define (make-empty-mailbox)
  (let ((mutex (make-mutex))
        (put-condvar (make-condition-variable))
        (get-condvar (make-condition-variable))
        (full? #f)
        (cell #f))

    (define (put! obj)
      (mutex-lock! mutex)
      (if full?
          (begin
            (mutex-unlock! mutex put-condvar)
            (put! obj))
          (begin
            (set! cell obj)
            (set! full? #t)
            (condition-variable-signal! get-condvar)
            (mutex-unlock! mutex))))

    (define (get!)
      (mutex-lock! mutex)
      (if (not full?)
          (begin
            (mutex-unlock! mutex get-condvar)
            (get!))
          (let ((result cell))
            (set! cell #f) ; avoid space leaks
            (set! full? #f)
            (condition-variable-signal! put-condvar)
            (mutex-unlock! mutex))))

    (lambda (msg)
      (case msg
        ((put!) put!)
        ((get!) get!)
        (else (error "unknown message"))))))

(define (mailbox-put! m obj) ((m 'put!) obj))
(define (mailbox-get! m) ((m 'get!)))
```

`(condition-variable-broadcast!` *`condition-variable`*`)` [procedure]
Unblocks all the threads blocked on the *`condition-variable`*. `condition-variable-broadcast!` returns an unspecified value.

For example:

```
(define (make-semaphore n)
  (vector n (make-mutex) (make-condition-variable)))

(define (semaphore-wait! sema)
  (mutex-lock! (vector-ref sema 1))
  (let ((n (vector-ref sema 0)))
    (if (> n 0)
        (begin
          (vector-set! sema 0 (- n 1))
          (mutex-unlock! (vector-ref sema 1)))
        (begin
          (mutex-unlock! (vector-ref sema 1) (vector-ref sema 2))
          (semaphore-wait! sema)))))

(define (semaphore-signal-by! sema increment)
  (mutex-lock! (vector-ref sema 1))
  (let ((n (+ (vector-ref sema 0) increment)))
    (vector-set! sema 0 n)
    (if (> n 0)
        (condition-variable-broadcast! (vector-ref sema 2)))
    (mutex-unlock! (vector-ref sema 1))))
```

`(current-time)` [procedure]

Returns the time object corresponding to the current time. For example:

```
> (current-time)
#<time #2>
```

`(time? obj)` [procedure]

Returns '`#t`' if *`obj`* is a time object, otherwise returns '`#f`'. If any of the predicates listed in Section 3.2 of the R5RS is true of *`obj`*, then `time?` is false of *`obj`*.

For example:

```
> (time? (current-time))
#t
> (time? 123)
#f
```

`(time->seconds time)` [procedure]

Converts the time object *`time`* into an exact or inexact real number representing the number of seconds elapsed since some implementation dependent reference point.

For example:

```
> (time->seconds (current-time))
955039784.928075
```

`(seconds->time x)` [procedure]

Converts into a time object the exact or inexact real number *`x`* representing the number of seconds elapsed since some implementation dependent reference point.

For example:

```
> (seconds->time (+ 10 (time->seconds (current-time))
a time object representing 10 seconds in the future
```

(current-exception-handler) [procedure]

Returns the current exception handler. For example:

```
> (current-exception-handler)
#<procedure #2 primordial-exception-handler>
```

(with-exception-handler *handler thunk*) [procedure]

Returns the result(s) of calling *thunk* with no arguments. The *handler*, which must be a procedure, is installed as the current exception handler in the dynamic environment in effect during the call to *thunk*. Note that the dynamic environment in effect during the call to *handler* has *handler* as the exception handler. Consequently, an exception raised during the call to *handler* may lead to an infinite loop.

For example:

```
> (with-exception-handler
    (lambda (e) (write e) 5)
    (lambda () (+ 1 (* 2 3) 4)))
11
> (with-exception-handler
    (lambda (e) (write e) 5)
    (lambda () (+ 1 (* 'two 3) 4)))
#<type-exception #2>10
> (with-exception-handler
    (lambda (e) (write e 9))
    (lambda () (+ 1 (* 'two 3) 4)))
infinite loop
```

(with-exception-catcher *handler thunk*) [procedure]

Returns the result(s) of calling *thunk* with no arguments. A new exception handler is installed as the current exception handler in the dynamic environment in effect during the call to *thunk*. This new exception handler will call the *handler*, which must be a procedure, with the exception object as an argument and with the same continuation as the call to with-exception-catcher. This implies that the dynamic environment in effect during the call to *handler* is the same as the one in effect at the call to with-exception-catcher. Consequently, an exception raised during the call to *handler* will not lead to an infinite loop.

For example:

```
> (with-exception-catcher
    (lambda (e) (write e) 5)
    (lambda () (+ 1 (* 2 3) 4)))
11
> (with-exception-catcher
    (lambda (e) (write e) 5)
    (lambda () (+ 1 (* 'two 3) 4)))
#<type-exception #2>10
```

```
> (with-exception-catcher
    (lambda (e) (write e 9))
    (lambda () (+ 1 (* 'two 3) 4)))
*** ERROR IN (console)@7.1 -- (Argument 2) OUTPUT PORT expected
(write '#<type-exception #3> 9)
1>
```

(raise *obj*) [procedure]

Calls the current exception handler with *obj* as the single argument. *obj* may be any Scheme object.

For example:

```
(define (f n)
  (if (< n 0) (raise "negative arg") (sqrt n))))

(define (g)
  (call-with-current-continuation
    (lambda (return)
      (with-exception-handler
        (lambda (exc)
          (return
            (if (string? exc)
                (string-append "error: " exc)
                "unknown error")))
        (lambda ()
          (write (f 4.))
          (write (f -1.))
          (write (f 9.)))))))

(g)  ==>  writes 2. and returns "error: negative arg"
```

(join-timeout-exception? *obj*) [procedure]

Returns '#t' if *obj* is a "join timeout exception" object, otherwise returns '#f'. A join timeout exception is raised when thread-join! is called, the timeout is reached and no *timeout-val* is supplied. If any of the predicates listed in Section 3.2 of the R5RS is true of *obj*, then join-timeout-exception? is false of *obj*.

(abandoned-mutex-exception? *obj*) [procedure]

Returns '#t' if *obj* is an "abandoned mutex exception" object, otherwise returns '#f'. An abandoned mutex exception is raised when the current thread locks a mutex that was owned by a thread which terminated (see mutex-lock!). If any of the predicates listed in Section 3.2 of the R5RS is true of *obj*, then abandoned-mutex-exception? is false of *obj*.

(terminated-thread-exception? *obj*) [procedure]

Returns '#t' if *obj* is a "terminated thread exception" object, otherwise returns '#f'. A terminated thread exception is raised when thread-join! is called and the target thread has terminated as a result of a call to thread-terminate!. If any of

the predicates listed in Section 3.2 of the R5RS is true of *obj*, then `terminated-thread-exception?` is false of *obj*.

`(uncaught-exception?` *obj*`)` [procedure]
Returns '`#t`' if *obj* is an "uncaught exception" object, otherwise returns '`#f`'. An uncaught exception is raised when `thread-join!` is called and the target thread has terminated because it raised an exception that called the initial exception handler of that thread. If any of the predicates listed in Section 3.2 of the R5RS is true of *obj*, then `uncaught-exception?` is false of *obj*.

`(uncaught-exception-reason` *exc*`)` [procedure]
exc must be an "uncaught exception" object. `uncaught-exception-reason` returns the object which was passed to the initial exception handler of that thread.

15 Interface to C

The Gambit Scheme system offers a mechanism for interfacing Scheme code and C code called the "C-interface". A Scheme program indicates which C functions it needs to have access to and which Scheme procedures can be called from C, and the C interface automatically constructs the corresponding Scheme procedures and C functions. The conversions needed to transform data from the Scheme representation to the C representation (and back), are generated automatically in accordance with the argument and result types of the C function or Scheme procedure.

The C-interface places some restrictions on the types of data that can be exchanged between C and Scheme. The mapping of data types between C and Scheme is discussed in the next section. The remaining sections of this chapter describe each special form of the C-interface.

15.1 The mapping of types between C and Scheme

Scheme and C do not provide the same set of built-in data types so it is important to understand which Scheme type is compatible with which C type and how values get mapped from one environment to the other. To improve compatibility a new type is added to Scheme, the '`foreign`' object type, and the following data types are added to C:

`scheme-object`
: denotes the universal type of Scheme objects (type `___SCMOBJ` defined in '`gambit.h`')

`bool`
: denotes the C++ '`bool`' type or the C '`int`' type (type `___BOOL` defined in '`gambit.h`')

`int8`
: 8 bit signed integer (type `___S8` defined in '`gambit.h`')

`unsigned-int8`
: 8 bit unsigned integer (type `___U8` defined in '`gambit.h`')

`int16`
: 16 bit signed integer (type `___S16` defined in '`gambit.h`')

`unsigned-int16`
: 16 bit unsigned integer (type `___U16` defined in '`gambit.h`')

`int32`
: 32 bit signed integer (type `___S32` defined in '`gambit.h`')

`unsigned-int32`
: 32 bit unsigned integer (type `___U32` defined in '`gambit.h`')

`int64`
: 64 bit signed integer (type `___S64` defined in '`gambit.h`')

`unsigned-int64`
: 64 bit unsigned integer (type `___U64` defined in '`gambit.h`')

`float32`
: 32 bit floating point number (type `___F32` defined in '`gambit.h`')

`float64`
: 64 bit floating point number (type `___F64` defined in '`gambit.h`')

`latin1`
: denotes LATIN-1 encoded characters (8 bit unsigned integer, type `___LATIN1` defined in '`gambit.h`')

`ucs2` denotes UCS-2 encoded characters (16 bit unsigned integer, type `___UCS2` defined in '`gambit.h`')

`ucs4` denotes UCS-4 encoded characters (32 bit unsigned integer, type `___UCS4` defined in '`gambit.h`')

`char-string`
denotes the C '`char*`' type when used as a null terminated string

`nonnull-char-string`
denotes the nonnull C '`char*`' type when used as a null terminated string

`nonnull-char-string-list`
denotes an array of nonnull C '`char*`' terminated with a null pointer

`latin1-string`
denotes LATIN-1 encoded strings (null terminated string of 8 bit unsigned integers, i.e. `___LATIN1*`)

`nonnull-latin1-string`
denotes nonnull LATIN-1 encoded strings (null terminated string of 8 bit unsigned integers, i.e. `___LATIN1*`)

`nonnull-latin1-string-list`
denotes an array of nonnull LATIN-1 encoded strings terminated with a null pointer

`utf8-string`
denotes UTF-8 encoded strings (null terminated string of `char`, i.e. `char*`)

`nonnull-utf8-string`
denotes nonnull UTF-8 encoded strings (null terminated string of `char`, i.e. `char*`)

`nonnull-utf8-string-list`
denotes an array of nonnull UTF-8 encoded strings terminated with a null pointer

`ucs2-string`
denotes UCS-2 encoded strings (null terminated string of 16 bit unsigned integers, i.e. `___UCS2*`)

`nonnull-ucs2-string`
denotes nonnull UCS-2 encoded strings (null terminated string of 16 bit unsigned integers, i.e. `___UCS2*`)

`nonnull-ucs2-string-list`
denotes an array of nonnull UCS-2 encoded strings terminated with a null pointer

`ucs4-string`
denotes UCS-4 encoded strings (null terminated string of 32 bit unsigned integers, i.e. `___UCS4*`)

`nonnull-ucs4-string`
: denotes nonnull UCS-4 encoded strings (null terminated string of 32 bit unsigned integers, i.e. `___UCS4*`)

`nonnull-ucs4-string-list`
: denotes an array of nonnull UCS-4 encoded strings terminated with a null pointer

To specify a particular C type inside the `c-lambda`, `c-define` and `c-define-type` forms, the following "Scheme notation" is used:

Scheme notation	C type
`void`	`void`
`bool`	`bool`
`char`	`char` (may be signed or unsigned depending on the C compiler)
`signed-char`	`signed char`
`unsigned-char`	`unsigned char`
`latin1`	`latin1`
`ucs2`	`ucs2`
`ucs4`	`ucs4`
`short`	`short`
`unsigned-short`	`unsigned short`
`int`	`int`
`unsigned-int`	`unsigned int`
`long`	`long`
`unsigned-long`	`unsigned long`
`long-long`	`long long`
`unsigned-long-long`	`unsigned long long`
`float`	`float`
`double`	`double`
`int8`	`int8`

`unsigned-int8`
: `unsigned-int8`

`int16` `int16`

`unsigned-int16`
: `unsigned-int16`

`int32` `int32`

`unsigned-int32`
: `unsigned-int32`

`int64` `int64`

`unsigned-int64`
: `unsigned-int64`

`float32` `float32`

`float64` `float64`

`(struct "`*`c-struct-id`*`" [`*`tag`* [`*`release-function`*`]])`
: `struct` *`c-struct-id`* (where *c-struct-id* is the name of a C structure; see below for the meaning of *tag* and *release-function*)

`(union "`*`c-union-id`*`" [`*`tag`* [`*`release-function`*`]])`
: `union` *`c-union-id`* (where *c-union-id* is the name of a C union; see below for the meaning of *tag* and *release-function*)

`(type "`*`c-type-id`*`" [`*`tag`* [`*`release-function`*`]])`
: *`c-type-id`* (where *c-type-id* is an identifier naming a C type; see below for the meaning of *tag* and *release-function*)

`(pointer `*`type`* [*`tag`* [*`release-function`*`]])`
: *`T*`* (where *T* is the C equivalent of *type* which must be the Scheme notation of a C type; see below for the meaning of *tag* and *release-function*)

`(nonnull-pointer `*`type`* [*`tag`* [*`release-function`*`]])`
: same as `(pointer `*`type`* [*`tag`* [*`release-function`*`]])` except the `NULL` pointer is not allowed

`(function (`*`type1`*`...) `*`result-type`*`)`
: function with the given argument types and result type

`(nonnull-function (`*`type1`*`...) `*`result-type`*`)`
: same as `(function (`*`type1`*`...) `*`result-type`*`)` except the `NULL` pointer is not allowed

`char-string`
: `char-string`

`nonnull-char-string`
: `nonnull-char-string`

`nonnull-char-string-list`
: `nonnull-char-string-list`

`latin1-string`
: `latin1-string`

`nonnull-latin1-string`
: `nonnull-latin1-string`

`nonnull-latin1-string-list`
: `nonnull-latin1-string-list`

`utf8-string`
: `utf8-string`

`nonnull-utf8-string`
: `nonnull-utf8-string`

`nonnull-utf8-string-list`
: `nonnull-utf8-string-list`

`ucs2-string`
: `ucs2-string`

`nonnull-ucs2-string`
: `nonnull-ucs2-string`

`nonnull-ucs2-string-list`
: `nonnull-ucs2-string-list`

`ucs4-string`
: `ucs4-string`

`nonnull-ucs4-string`
: `nonnull-ucs4-string`

`nonnull-ucs4-string-list`
: `nonnull-ucs4-string-list`

`scheme-object`
: `scheme-object`

`name`
: appropriate translation of *name* (where *name* is a C type defined with `c-define-type`)

`"`*`c-type-id`*`"`
: *c-type-id* (this form is equivalent to `(type "`*`c-type-id`*`")`)

The `struct`, `union`, `type`, `pointer` and `nonnull-pointer` types are "foreign types" and they are represented on the Scheme side as "foreign objects". A foreign object is internally represented as a pointer. This internal pointer is identical to the C pointer being represented in the case of the `pointer` and `nonnull-pointer` types.

In the case of the `struct`, `union` and `type` types, the internal pointer points to a copy of the C data type being represented. When an instance of one of these types is converted from C to Scheme, a block of memory is allocated from the C heap and initialized with the instance and then a foreign object is allocated from the Scheme heap and initialized with the pointer to this copy. This approach may appear overly complex, but it allows the conversion of C++ classes that do not have a zero parameter constructor or an assignment

method (i.e. when compiling with a C++ compiler an instance is copied using '`new type (instance)`', which calls the copy-constructor of *type* if it is a class; *type*'s assignment operator is never used). Conversion from Scheme to C simply dereferences the internal pointer (no allocation from the C heap is performed). Deallocation of the copy on the C heap is under the control of the release function attached to the foreign object (see below).

For type checking on the Scheme side, a *tag* can be specified within a foreign type specification. The *tag* must be `#f` or a symbol. When it is not specified the *tag* defaults to a symbol whose name, as returned by `symbol->string`, is the C type declaration for that type. For example the default tag for the type '`(pointer (pointer char))`' is the symbol '`char**`'. Two foreign types are compatible (i.e. can be converted from one to the other) if they have identical tags or if at least one of the tags is `#f`. For the safest code the `#f` tag should be used sparingly, as it completely bypasses type checking. The external representation of Scheme foreign objects (used by the `write` procedure) contains the tag if it is not `#f`, and the hexadecimal address denoted by the internal pointer, for example '`#<char** 1 0x2AAC535C>`'. Note that the hexadecimal address is in C notation, which can be easily transferred to a C debugger with a "cut-and-paste".

A *release-function* can also be specified within a foreign type specification. The *release-function* must be `#f` or a string naming a C function with a single parameter of type '`void*`' (in which the internal pointer is passed) and with a result of type '`___SCMOBJ`' (for returning an error code). When the *release-function* is not specified or is `#f` a default function is constructed by the C-interface. This default function does nothing in the case of the `pointer` and `nonnull-pointer` types (deallocation is not the responsibility of the C-interface) and returns the fixnum '`___FIX(___NO_ERR)`' to indicate no error. In the case of the `struct`, `union` and `type` types, the default function reclaims the copy on the C heap referenced by the internal pointer (when using a C++ compiler this is done using '`delete (type*)internal-pointer`', which calls the destructor of *type* if it is a class) and returns '`___FIX(___NO_ERR)`'. In many situations the default *release-function* will perform the appropriate cleanup for the foreign type. However, in certain cases special operations (such as decrementing a reference count, removing the object from a table, etc) must be performed. For such cases a user supplied *release-function* is needed.

The *release-function* is invoked at most once for any foreign object. After the *release-function* is invoked, the foreign object is considered "released" and can no longer be used in a foreign type conversion. When the garbage collector detects that a foreign object is no longer reachable by the program, it will invoke the *release-function* if the foreign object is not yet released. When there is a need to release the foreign object promptly, the program can explicitly call the Scheme procedure `foreign-release!` which invokes the *release-function* if the foreign object is not yet released, and does nothing otherwise.

The following table gives the C types to which each Scheme type can be converted:

Scheme type
: Allowed target C types

boolean `#f`
: `scheme-object`; `bool`; `pointer`; `function`; `char-string`; `latin1-string`; `utf8-string`; `ucs2-string`; `ucs4-string`

boolean `#t`
: `scheme-object`; `bool`

character `scheme-object`; `bool`; [[un]`signed`] `char`; `latin1`; `ucs2`; `ucs4`

exact integer
: `scheme-object`; `bool`; [`unsigned-`] `int8`/`int16`/`int32`/`int64`; [`unsigned`] `short`/`int`/`long`

inexact real
: `scheme-object`; `bool`; `float`; `double`; `float32`; `float64`

string `scheme-object`; `bool`; `char-string`; `nonnull-char-string`; `latin1-string`; `nonnull-latin1-string`; `utf8-string`; `nonnull-utf8-string`; `ucs2-string`; `nonnull-ucs2-string`; `ucs4-string`; `nonnull-ucs4-string`

foreign object
: `scheme-object`; `bool`; `struct`/`union`/`type`/`pointer`/`nonnull-pointer` with the appropriate tag

vector `scheme-object`; `bool`

symbol `scheme-object`; `bool`

procedure `scheme-object`; `bool`; `function`; `nonnull-function`

other objects
: `scheme-object`; `bool`

The following table gives the Scheme types to which each C type will be converted:

C type Resulting Scheme type

scheme-object
: the Scheme object encoded

bool boolean

[[un]`signed`] `char`; `latin1`; `ucs2`; `ucs4`
: character

[`unsigned-`] `int8`/`int16`/`int32`/`int64`; [`unsigned`] `short`/`int`/`long`
: exact integer

`float`; `double`; `float32`; `float64`
: inexact real

`char-string`; `latin1-string`; `utf8-string`; `ucs2-string`; `ucs4-string`
: string or `#f` if it is equal to '`NULL`'

`nonnull-char-string`; `nonnull-latin1-string`; `nonnull-utf8-string`; `nonnull-ucs2-string`; `nonnull-ucs4-string`
: string

`struct`/`union`/`type`/`pointer`/`nonnull-pointer`
: foreign object with the appropriate tag or `#f` in the case of a `pointer` equal to '`NULL`'

`function`
: procedure or `#f` if it is equal to '`NULL`'

`nonnull-function`
: procedure

`void`
: void object

All Scheme types are compatible with the C types `scheme-object` and `bool`. Conversion to and from the C type `scheme-object` is the identity function on the object encoding. This provides a low-level mechanism for accessing Scheme's object representation from C (with the help of the macros in the '`gambit.h`' header file). When a C `bool` type is expected, an extended Scheme boolean can be passed (`#f` is converted to 0 and all other values are converted to 1).

The Scheme boolean `#f` can be passed to the C environment where a `char-string`, `latin1-string`, `utf8-string`, `ucs2-string`, `ucs4-string`, `pointer` or `function` type is expected. In this case, `#f` is converted to the '`NULL`' pointer. C `bool`s are extended booleans so any value different from 0 represents true. Thus, a C `bool` passed to the Scheme environment is mapped as follows: 0 to `#f` and all other values to `#t`.

A Scheme character passed to the C environment where any C character type is expected is converted to the corresponding character in the C environment. An error is signaled if the Scheme character does not fit in the C character. Any C character type passed to Scheme is converted to the corresponding Scheme character. An error is signaled if the C character does not fit in the Scheme character.

A Scheme exact integer passed to the C environment where a C integer type (other than `char`) is expected is converted to the corresponding integral value. An error is signaled if the value falls outside of the range representable by that integral type. C integer values passed to the Scheme environment are mapped to the same Scheme exact integer. If the value is outside the fixnum range, a bignum is created.

A Scheme inexact real passed to the C environment is converted to the corresponding `float`, `double`, `float32` or `float64` value. C `float`, `double`, `float32` and `float64` values passed to the Scheme environment are mapped to the closest Scheme inexact real.

Scheme's rational numbers and complex numbers are not compatible with any C numeric type.

A Scheme string passed to the C environment where any C string type is expected is converted to a null terminated string using the appropriate encoding. The C string is a fresh copy of the Scheme string. If the C string was created for an argument of a `c-lambda`, the C string will be reclaimed when the `c-lambda` returns. If the C string was created for returning the result of a `c-define` to C, the caller is responsible for reclaiming the C string with a call to the `___release_string` function (see below for an example). Any C string type passed to the Scheme environment causes the creation of a fresh Scheme string containing a copy of the C string (unless the C string is equal to `NULL`, in which case it is converted to `#f`).

A foreign type passed to the Scheme environment causes the creation and initialization of a Scheme foreign object with the appropriate tag (except for the case of a `pointer` equal to `NULL` which is converted to `#f`). A Scheme foreign object can be passed where a foreign type is expected, on the condition that the tags are appropriate (identical or one is `#f`) and the Scheme foreign object is not yet released. The value `#f` is also acceptable for a `pointer` type, and is converted to `NULL`.

Scheme procedures defined with the `c-define` special form can be passed where the `function` and `nonnull-function` types are expected. The value `#f` is also acceptable for a `function` type, and is converted to `NULL`. No other Scheme procedures are acceptable. Conversion from the `function` and `nonnull-function` types to Scheme procedures is not currently implemented.

15.2 The `c-declare` special form

Synopsis:

```
(c-declare c-declaration)
```

Initially, the C file produced by `gsc` contains only an '`#include`' of '`gambit.h`'. This header file provides a number of macro and procedure declarations to access the Scheme object representation. The special form `c-declare` adds *c-declaration* (which must be a string containing the C declarations) to the C file. This string is copied to the C file on a new line so it can start with preprocessor directives. All types of C declarations are allowed (including type declarations, variable declarations, function declarations, '`#include`' directives, '`#define`'s, and so on). These declarations are visible to subsequent `c-declare`s, `c-initialize`s, and `c-lambda`s, and `c-define`s in the same module. The most common use of this special form is to declare the external functions that are referenced in `c-lambda` special forms. Such functions must either be declared explicitly or by including a header file which contains the appropriate C declarations.

The `c-declare` special form does not return a value. It can only appear at top level.

For example:

```
(c-declare
"
#include <stdio.h>

extern char *getlogin ();

#ifdef sparc
char *host = \"sparc\";  /* note backslashes */
#else
char *host = \"unknown\";
#endif

FILE *tfile;
")
```

15.3 The `c-initialize` special form

Synopsis:

```
(c-initialize c-code)
```

Just after the program is loaded and before control is passed to the Scheme code, each C file is initialized by calling its associated initialization function. The body of this function is normally empty but it can be extended by using the `c-initialize` form. Each occurence of the `c-initialize` form adds code to the body of the initialization function in the order

of appearance in the source file. *c-code* must be a string containing the C code to execute. This string is copied to the C file on a new line so it can start with preprocessor directives.

The `c-initialize` special form does not return a value. It can only appear at top level.

For example:

```
(c-initialize "tfile = tmpfile ();")
```

15.4 The `c-lambda` special form

Synopsis:

```
(c-lambda (type1...) result-type c-name-or-code)
```

The `c-lambda` special form makes it possible to create a Scheme procedure that will act as a representative of some C function or C code sequence. The first subform is a list containing the type of each argument. The type of the function's result is given next. Finally, the last subform is a string that either contains the name of the C function to call or some sequence of C code to execute. Variadic C functions are not supported. The resulting Scheme procedure takes exactly the number of arguments specified and delivers them in the same order to the C function. When the Scheme procedure is called, the arguments will be converted to their C representation and then the C function will be called. The result returned by the C function will be converted to its Scheme representation and this value will be returned from the Scheme procedure call. An error will be signaled if some conversion is not possible. The temporary memory allocated from the C heap for the conversion of the arguments and result will be reclaimed whether there is an error or not.

When *c-name-or-code* is not a valid C identifier, it is treated as an arbitrary piece of C code. Within the C code the variables '`___arg1`', '`___arg2`', etc. can be referenced to access the converted arguments. Similarly, the result to be returned from the call should be assigned to the variable '`___result`' except if the result is of `struct`, `union`, `type`, `pointer` or `nonnull-pointer` type in which case a pointer should be assigned to the variable '`___result_voidstar`' which is of type '`void*`'. If no result needs to be returned, the *result-type* should be `void` and no assignment to the variable '`___result`' or '`___result_voidstar`' should take place. Note that the C code should not contain `return` statements as this is meaningless. Control must always fall off the end of the C code. The C code is copied to the C file on a new line so it can start with preprocessor directives. Moreover the C code is always placed at the head of a compound statement whose lifetime encloses the C to Scheme conversion of the result. Consequently, temporary storage (strings in particular) declared at the head of the C code can be returned by assigning them to '`___result`' or '`___result_voidstar`'. In the *c-name-or-code*, the macro '`___AT_END`' may be defined as the piece of C code to execute before control is returned to Scheme but after the result is converted to its Scheme representation. This is mainly useful to deallocate temporary storage contained in the result.

When passed to the Scheme environment, the C `void` type is converted to the void object.

For example:

```
(define fopen
  (c-lambda (nonnull-char-string nonnull-char-string)
```

```
            (pointer "FILE")
   "fopen"))

(define fgetc
  (c-lambda ((pointer "FILE"))
            int
   "fgetc"))

(let ((f (fopen "datafile" "r")))
  (if f (write (fgetc f))))

(define char-code (c-lambda (char) int "___result = ___arg1;"))

(define host ((c-lambda () nonnull-char-string "___result = host;")))

(define stdin ((c-lambda () (pointer "FILE") "___result = stdin;")))

((c-lambda () void
  "printf( \"hello\\n\" ); printf( \"world\\n\" );"))

(define pack-1-char
  (c-lambda (char)
            nonnull-char-string
   "
   ___result = malloc (2);
   if (___result != NULL) { ___result[0] = ___arg1; ___result[1] = 0; }
   #define ___AT_END if (___result != NULL) free (___result);
   "))

(define pack-2-chars
  (c-lambda (char char)
            nonnull-char-string
   "
   char s[3]; s[0] = ___arg1; s[1] = ___arg2; s[2] = 0; ___result = s;
   "))
```

15.5 The `c-define` special form

Synopsis:

```
(c-define (variable define-formals) (type1...) result-type c-name scope
  body)
```

The `c-define` special form makes it possible to create a C function that will act as a representative of some Scheme procedure. A C function named *c-name* as well as a Scheme procedure bound to the variable *variable* are defined. The parameters of the Scheme procedure are *define-formals* and its body is at the end of the form. The type of each argument of the C function, its result type and *c-name* (which must be a string) are specified after the parameter specification of the Scheme procedure. When the C function

c-name is called from C, its arguments are converted to their Scheme representation and passed to the Scheme procedure. The result of the Scheme procedure is then converted to its C representation and the C function *c-name* returns it to its caller.

The scope of the C function can be changed with the *scope* parameter, which must be a string. This string is placed immediately before the declaration of the C function. So if *scope* is the string `"static"`, the scope of *c-name* is local to the module it is in, whereas if *scope* is the empty string, *c-name* is visible from other modules.

The `c-define` special form does not return a value. It can only appear at top level.

For example:

```
(c-define (proc x #!optional (y x) #!rest z) (int int char float) int "f" "
  (write (cons x (cons y z)))
  (newline)
  (+ x y))

(proc 1 2 #\x 1.5) => 3 and prints (1 2 #\x 1.5)
(proc 1)           => 2 and prints (1 1)

; if f is called from C with the call  f (1, 2, 'x', 1.5)
; the value 3 is returned and (1 2 #\x 1.5) is printed.
; f has to be called with 4 arguments.
```

The `c-define` special form is particularly useful when the driving part of an application is written in C and Scheme procedures are called directly from C. The Scheme part of the application is in a sense a "server" that is providing services to the C part. The Scheme procedures that are to be called from C need to be defined using the `c-define` special form. Before it can be used, the Scheme part must be initialized with a call to the function '`___setup`'. Before the program terminates, it must call the function '`___cleanup`' so that the Scheme part may do final cleanup. A sample application is given in the file '`tests/server.scm`'.

15.6 The `c-define-type` special form

Synopsis:

`(c-define-type` *name type* [*c-to-scheme scheme-to-c* [*cleanup*]]`)`

This form associates the type identifier *name* to the C type *type*. The *name* must not clash with predefined types (e.g. `char-string`, `latin1`, etc.) or with types previously defined with `c-define-type` in the same file. The `c-define-type` special form does not return a value. It can only appear at top level.

If only the two parameters *name* and *type* are supplied then after this definition, the use of *name* in a type specification is synonymous to *type*.

For example:

```
(c-define-type FILE "FILE")
(c-define-type FILE* (pointer FILE))
(c-define-type time-struct-ptr (pointer (struct "tms")))
(define fopen (c-lambda (char-string char-string) FILE* "fopen"))
(define fgetc (c-lambda (FILE*) int "fgetc"))
```

Note that identifiers are not case sensitive in standard Scheme but it is good programming practice to use a *name* with the same case as in C.

If four or more parameters are supplied, then *type* must be a string naming the C type, *c-to-scheme* and *scheme-to-c* must be strings suffixing the C macros that convert data of that type between C and Scheme. If *cleanup* is supplied it must be a boolean indicating whether it is necessary to perform a cleanup operation (such as freeing memory) when data of that type is converted from Scheme to C (it defaults to `#t`). The cleanup information is used when the C stack is unwound due to a continuation invocation (see ⟨undefined⟩ [continuations], page ⟨undefined⟩). Although it is safe to always specify `#t`, it is more efficient in time and space to specify `#f` because the unwinding mechanism can skip C-interface frames which only contain conversions of data types requiring no cleanup. Two pairs of C macros need to be defined for conversions performed by `c-lambda` forms and two pairs for conversions performed by `c-define` forms:

```
___BEGIN_CFUN_scheme-to-c(___SCMOBJ, type, int)
___END_CFUN_scheme-to-c(___SCMOBJ, type, int)

___BEGIN_CFUN_c-to-scheme(type, ___SCMOBJ)
___END_CFUN_c-to-scheme(type, ___SCMOBJ)

___BEGIN_SFUN_c-to-scheme(type, ___SCMOBJ, int)
___END_SFUN_c-to-scheme(type, ___SCMOBJ, int)

___BEGIN_SFUN_scheme-to-c(___SCMOBJ, type)
___END_SFUN_scheme-to-c(___SCMOBJ, type)
```

The macros prefixed with `___BEGIN` perform the conversion and those prefixed with `___END` perform any cleanup necessary (such as freeing memory temporarily allocated for the conversion). The macro `___END_CFUN_scheme-to-c` must free the result of the conversion if it is memory allocated, and `___END_SFUN_scheme-to-c` must not (i.e. it is the responsibility of the caller to free the result).

The first parameter of these macros is the C variable that contains the value to be converted, and the second parameter is the C variable in which to store the converted value. The third parameter, when present, is the index (starting at 1) of the parameter of the `c-lambda` or `c-define` form that is being converted (this is useful for reporting precise error information when a conversion is impossible).

To allow for type checking, the first three `___BEGIN` macros must expand to an unterminated compound statement prefixed by an `if`, conditional on the absence of type check error:

```
if ((___err = conversion_operation) == ___FIX(___NO_ERR)) {
```

The last `___BEGIN` macro must expand to an unterminated compound statement:

```
{ ___err = conversion_operation;
```

If type check errors are impossible then a `___BEGIN` macro can simply expand to an unterminated compound statement performing the conversion:

```
{ conversion_operation;
```

The `___END` macros must expand to a statement, or to nothing if no cleanup is required, followed by a closing brace (to terminate the compound statement started at the corresponding `___BEGIN` macro).

The *conversion_operation* is typically a function call that returns an error code value of type `___SCMOBJ` (the error codes are defined in '`gambit.h`', and the error code `___FIX(___UNKNOWN_ERR)` is available for generic errors). *conversion_operation* can also set the variable `___errmsg` of type `___SCMOBJ` to a specific Scheme string error message.

Below is a simple example showing how to interface to an '`EBCDIC`' character type. Memory allocation is not needed for conversion and type check errors are impossible when converting EBCDIC to Scheme characters, but they are possible when converting from Scheme characters to EBCDIC since Gambit supports Unicode characters.

```
(c-declare
"
typedef char EBCDIC; /* EBCDIC encoded characters */

void put_char (EBCDIC c) { ... } /* EBCDIC I/O functions */
EBCDIC get_char (void) { ... }

char EBCDIC_to_latin1[256] = { ... }; /* conversion tables */
char latin1_to_EBCDIC[256] = { ... };

___SCMOBJ SCMOBJ_to_EBCDIC (___SCMOBJ src, EBCDIC *dst)
{
  int x = ___INT(src); /* convert from Scheme character to int */
  if (x > 255) return ___UNKNOWN_ERR;
  *dst = latin1_to_EBCDIC[x];
  return ___FIX(___NO_ERR);
}

#define ___BEGIN_CFUN_SCMOBJ_to_EBCDIC(src,dst,i) \\
if ((___err = SCMOBJ_to_EBCDIC (src, &dst)) == ___FIX(___NO_ERR)) {
#define ___END_CFUN_SCMOBJ_to_EBCDIC(src,dst,i) }

#define ___BEGIN_CFUN_EBCDIC_to_SCMOBJ(src,dst) \\
{ dst = ___CHR(EBCDIC_to_latin1[src]);
#define ___END_CFUN_EBCDIC_to_SCMOBJ(src,dst) }

#define ___BEGIN_SFUN_EBCDIC_to_SCMOBJ(src,dst,i) \\
{ dst = ___CHR(EBCDIC_to_latin1[src]);
#define ___END_SFUN_EBCDIC_to_SCMOBJ(src,dst,i) }

#define ___BEGIN_SFUN_SCMOBJ_to_EBCDIC(src,dst) \\
{ ___err = SCMOBJ_to_EBCDIC (src, &dst);
#define ___END_SFUN_SCMOBJ_to_EBCDIC(src,dst) }
")
```

```
(c-define-type EBCDIC "EBCDIC" "EBCDIC_to_SCMOBJ" "SCMOBJ_to_EBCDIC" #f)

(define put-char (c-lambda (EBCDIC) void "put_char"))
(define get-char (c-lambda () EBCDIC "get_char"))

(c-define (write-EBCDIC c) (EBCDIC) void "write_EBCDIC" ""
  (write-char c))

(c-define (read-EBCDIC) () EBCDIC "read_EBCDIC" ""
  (read-char))
```

Below is a more complex example that requires memory allocation when converting from C to Scheme. It is an interface to a 2D '`point`' type which is represented in Scheme by a pair of integers. The conversion of the `x` and `y` components is done by calls to the conversion macros for the `int` type (defined in '`gambit.h`'). Note that no cleanup is necessary when converting from Scheme to C (i.e. the last parameter of the `c-define-type` is `#f`).

```
(c-declare
"
typedef struct { int x, y; } point;

void line_to (point p) { ... }
point get_mouse (void) { ... }
point add_points (point p1, point p2) { ... }

___SCMOBJ SCMOBJ_to_POINT (___SCMOBJ src, point *dst, int arg_num)
{
  ___SCMOBJ ___err = ___FIX(___NO_ERR);
  if (!___PAIRP(src))
    ___err = ___FIX(___UNKNOWN_ERR);
  else
    {
      ___SCMOBJ car = ___CAR(src);
      ___SCMOBJ cdr = ___CDR(src);
      ___BEGIN_CFUN_SCMOBJ_TO_INT(car,dst->x,arg_num)
      ___BEGIN_CFUN_SCMOBJ_TO_INT(cdr,dst->y,arg_num)
      ___END_CFUN_SCMOBJ_TO_INT(cdr,dst->y,arg_num)
      ___END_CFUN_SCMOBJ_TO_INT(car,dst->x,arg_num)
    }
  return ___err;
}

___SCMOBJ POINT_to_SCMOBJ (point src, ___SCMOBJ *dst, int arg_num)
{
  ___SCMOBJ ___err = ___FIX(___NO_ERR);
  ___SCMOBJ x_scmobj;
  ___SCMOBJ y_scmobj;
  ___BEGIN_SFUN_INT_TO_SCMOBJ(src.x,x_scmobj,arg_num)
```

```
  ___BEGIN_SFUN_INT_TO_SCMOBJ(src.y,y_scmobj,arg_num)
  *dst = ___EXT(___make_pair) (x_scmobj, y_scmobj, ___STILL);
  if (___FIXNUMP(*dst))
    ___err = *dst; /* return allocation error */
  ___END_SFUN_INT_TO_SCMOBJ(src.y,y_scmobj,arg_num)
  ___END_SFUN_INT_TO_SCMOBJ(src.x,x_scmobj,arg_num)
  return ___err;
}

#define ___BEGIN_CFUN_SCMOBJ_to_POINT(src,dst,i) \\
if ((___err = SCMOBJ_to_POINT (src, &dst, i)) == ___FIX(___NO_ERR)) {
#define ___END_CFUN_SCMOBJ_to_POINT(src,dst,i) }

#define ___BEGIN_CFUN_POINT_to_SCMOBJ(src,dst) \\
if ((___err = POINT_to_SCMOBJ (src, &dst, ___RETURN_POS)) == ___FIX(___NO_ER
#define ___END_CFUN_POINT_to_SCMOBJ(src,dst) \\
___EXT(___release_scmobj) (dst); }

#define ___BEGIN_SFUN_POINT_to_SCMOBJ(src,dst,i) \\
if ((___err = POINT_to_SCMOBJ (src, &dst, i)) == ___FIX(___NO_ERR)) {
#define ___END_SFUN_POINT_to_SCMOBJ(src,dst,i) \\
___EXT(___release_scmobj) (dst); }

#define ___BEGIN_SFUN_SCMOBJ_to_POINT(src,dst) \\
{ ___err = SCMOBJ_to_POINT (src, &dst, ___RETURN_POS);
#define ___END_SFUN_SCMOBJ_to_POINT(src,dst) }
")

(c-define-type point "point" "POINT_to_SCMOBJ" "SCMOBJ_to_POINT" #f)

(define line-to (c-lambda (point) void "line_to"))
(define get-mouse (c-lambda () point "get_mouse"))
(define add-points (c-lambda (point point) point "add_points"))

(c-define (write-point p) (point) void "write_point" ""
  (write p))

(c-define (read-point) () point "read_point" ""
  (read))
```

An example that requires memory allocation when converting from C to Scheme and Scheme to C is shown below. It is an interface to a "null-terminated array of strings" type which is represented in Scheme by a list of strings. Note that some cleanup is necessary when converting from Scheme to C.

```
(c-declare
"
#include <stdlib.h>
```

```
#include <unistd.h>

extern char **environ;

char **get_environ (void) { return environ; }

void free_strings (char **strings)
{
  char **ptr = strings;
  while (*ptr != NULL)
    {
      ___EXT(___release_string) (*ptr);
      ptr++;
    }
  free (strings);
}

___SCMOBJ SCMOBJ_to_STRINGS (___SCMOBJ src, char ***dst, int arg_num)
{
  /*
   * Src is a list of Scheme strings.  Dst will be a null terminated
   * array of C strings.
   */

  int i;
  ___SCMOBJ lst = src;
  int len = 4; /* start with a small result array */
  char **result = (char**) malloc (len * sizeof (char*));

  if (result == NULL)
    return ___FIX(___HEAP_OVERFLOW_ERR);

  i = 0;
  result[i] = NULL; /* always keep array null terminated */

  while (___PAIRP(lst))
    {
      ___SCMOBJ scm_str = ___CAR(lst);
      char *c_str;
      ___SCMOBJ ___err;

      if (i >= len-1) /* need to grow the result array? */
        {
          char **new_result;
          int j;

          len = len * 3 / 2;
```

```
            new_result = (char**) malloc (len * sizeof (char*));
            if (new_result == NULL)
              {
                free_strings (result);
                return ___FIX(___HEAP_OVERFLOW_ERR);
              }
            for (j=i; j>=0; j--)
              new_result[j] = result[j];
            free (result);
            result = new_result;
          }

        ___err = ___EXT(___SCMOBJ_to_CHARSTRING) (scm_str, &c_str, arg_num);

        if (___err != ___FIX(___NO_ERR))
          {
            free_strings (result);
            return ___err;
          }

        result[i++] = c_str;
        result[i] = NULL;
        lst = ___CDR(lst);
      }

    if (!___NULLP(lst))
      {
        free_strings (result);
        return ___FIX(___UNKNOWN_ERR);
      }

    /*
     * Note that the caller is responsible for calling free_strings
     * when it is done with the result.
     */

    *dst = result;
    return ___FIX(___NO_ERR);
  }

  ___SCMOBJ STRINGS_to_SCMOBJ (char **src, ___SCMOBJ *dst, int arg_num)
  {
    ___SCMOBJ ___err = ___FIX(___NO_ERR);
    ___SCMOBJ result = ___NUL; /* start with the empty list */
    int i = 0;

    while (src[i] != NULL)
```

```
  i++;

/* build the list of strings starting at the tail */

while (--i >= 0)
  {
    ___SCMOBJ scm_str;
    ___SCMOBJ new_result;

    /*
     * Invariant: result is either the empty list or a ___STILL pair
     * with reference count equal to 1.  This is important because
     * it is possible that ___CHARSTRING_to_SCMOBJ and ___make_pair
     * will invoke the garbage collector and we don't want the
     * reference in result to become invalid (which would be the
     * case if result was a ___MOVABLE pair or if it had a zero
     * reference count).
     */

    ___err = ___EXT(___CHARSTRING_to_SCMOBJ) (src[i], &scm_str, arg_num);

    if (___err != ___FIX(___NO_ERR))
      {
        ___EXT(___release_scmobj) (result); /* allow GC to reclaim result
        return ___FIX(___UNKNOWN_ERR);
      }

    /*
     * Note that scm_str will be a ___STILL object with reference
     * count equal to 1, so there is no risk that it will be
     * reclaimed or moved if ___make_pair invokes the garbage
     * collector.
     */

    new_result = ___EXT(___make_pair) (scm_str, result, ___STILL);

    /*
     * We can zero the reference count of scm_str and result (if
     * not the empty list) because the pair now references these
     * objects and the pair is reachable (it can't be reclaimed
     * or moved by the garbage collector).
     */

    ___EXT(___release_scmobj) (scm_str);
    ___EXT(___release_scmobj) (result);

    result = new_result;
```

```
      if (___FIXNUMP(result))
        return result; /* allocation failed */
    }

  /*
   * Note that result is either the empty list or a ___STILL pair
   * with a reference count equal to 1.  There will be a call to
   * ___release_scmobj later on (in ___END_CFUN_STRINGS_to_SCMOBJ
   * or ___END_SFUN_STRINGS_to_SCMOBJ) that will allow the garbage
   * collector to reclaim the whole list of strings when the Scheme
   * world no longer references it.
   */

  *dst = result;
  return ___FIX(___NO_ERR);
}

#define ___BEGIN_CFUN_SCMOBJ_to_STRINGS(src,dst,i) \\
if ((___err = SCMOBJ_to_STRINGS (src, &dst, i)) == ___FIX(___NO_ERR)) {
#define ___END_CFUN_SCMOBJ_to_STRINGS(src,dst,i) \\
free_strings (dst); }

#define ___BEGIN_CFUN_STRINGS_to_SCMOBJ(src,dst) \\
if ((___err = STRINGS_to_SCMOBJ (src, &dst, ___RETURN_POS)) == ___FIX(___NO_
#define ___END_CFUN_STRINGS_to_SCMOBJ(src,dst) \\
___EXT(___release_scmobj) (dst); }

#define ___BEGIN_SFUN_STRINGS_to_SCMOBJ(src,dst,i) \\
if ((___err = STRINGS_to_SCMOBJ (src, &dst, i)) == ___FIX(___NO_ERR)) {
#define ___END_SFUN_STRINGS_to_SCMOBJ(src,dst,i) \\
___EXT(___release_scmobj) (dst); }

#define ___BEGIN_SFUN_SCMOBJ_to_STRINGS(src,dst) \\
{ ___err = SCMOBJ_to_STRINGS (src, &dst, ___RETURN_POS);
#define ___END_SFUN_SCMOBJ_to_STRINGS(src,dst) }
")

(c-define-type char** "char**" "STRINGS_to_SCMOBJ" "SCMOBJ_to_STRINGS")

(define execv (c-lambda (char-string char**) int "execv"))
(define get-environ (c-lambda () char** "get_environ"))

(c-define (write-strings x) (char**) void "write_strings" ""
  (write x))

(c-define (read-strings) () char** "read_strings" ""
```

```
  (read))
```

15.7 Continuations and the C-interface

The C-interface allows C to Scheme calls to be nested. This means that during a call from C to Scheme another call from C to Scheme can be performed. This case occurs in the following program:

```
(c-declare
"
int p (char *); /* forward declarations */
int q (void);

int a (char *x) { return 2 * p (x+1); }
int b (short y) { return y + q (); }
")

(define a (c-lambda (char-string) int "a"))
(define b (c-lambda (short) int "b"))

(c-define (p z) (char-string) int "p" ""
  (+ (b 10) (string-length z)))

(c-define (q) () int "q" ""
  123)

(write (a "hello"))
```

In this example, the main Scheme program calls the C function '`a`' which calls the Scheme procedure '`p`' which in turn calls the C function '`b`' which finally calls the Scheme procedure '`q`'.

Gambit-C maintains the Scheme continuation separately from the C stack, thus allowing the Scheme continuation to be unwound independently from the C stack. The C stack frame created for the C function '`f`' is only removed from the C stack when control returns from '`f`' or when control returns to a C function "above" '`f`'. Special care is required for programs which escape to Scheme (using first-class continuations) from a Scheme to C (to Scheme) call because the C stack frame will remain on the stack. The C stack may overflow if this happens in a loop with no intervening return to a C function. To avoid this problem make sure the C stack gets cleaned up by executing a normal return from a Scheme to C call.

16 Known limitations and deficiencies

- On some systems floating point overflows will cause the program to terminate with a floating point exception.
- The compiler will not properly compile files with more than one definition (with `define`) of the same procedure. Replace all but the first `define` with assignments (`set!`).
- Records (defined through `define-structure`) can be written with `write` but can not be read by `read`.
- On MSDOS and Microsoft Windows, ⟨^C⟩ is sometimes interpreted as ⟨^Z⟩ (i.e. an end-of-file).
- On some systems floating point operations involving '`+nan.`' '`+inf.`', '`-inf.`', or '`-0.`' do not return the value required by the IEEE 754 floating point standard.

17 Bugs fixed

- The `floor` and `ceiling` procedures gave incorrect results for negative arguments.
- The `round` procedure did not obey the round to even rule. A value exactly in between two consecutive integers is now correctly rounded to the closest even integer.
- Heap overflow was not tested properly when a nonnull rest parameter was created. This could corrupt the heap in certain situations.
- The procedure `apply` did not check that an implementation limit on the number of arguments was not exceeded. This could corrupt the heap if too many arguments were passed to `apply`.
- The algorithms used by the compiler did not scale well to the compilation of large procedures and modules. Compilation is now faster and takes less memory.
- The compilation of nested `and` and `or` special forms was very slow for deep nestings. This is now much faster. Note that the code generated has not changed.
- On the Macintosh, when compiled with CodeWarrior, floating point computations gave random results.
- Improper allocation could occur when inlined floating point operations where combined with inlined allocators (e.g. '`list`', '`vector`', '`lambda`').
- Previously nested C to Scheme calls were prohibited. They are now allowed.
- A memory leak occuring when long output string ports were created has been fixed.
- `equal?` was not performed properly when the arguments were procedures. This could cause the program to crash.
- Write/read invariance of inexact numbers is now obeyed. An inexact number written out with `display` or `write` will be read back by `read` as the same number.
- The procedures `display`, `write` and `number->string` are more precise and much faster than before (up to a factor of 50).
- The procedure `exact->inexact` convert exact rationals much more precisely than before, in particular when the denominator is more than `1e308`.

18 Copyright and distribution information

General Index

(Index is nonexistent)

Table of Contents