Gambit-C, version 2.7

A portable implementation of Scheme
Edition 2.7, September 1997

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 programs: `gsi`, the Gambit Scheme interpreter, and `gsc`, the Gambit Scheme compiler.

Gambit-C is a version of the Gambit 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.

For the most up to date information on Gambit please check the Gambit web page at '`http://www.iro.umontreal.ca/~gambit`' or send mail to '`gambit@iro.umontreal.ca`'.

Bug reports should be sent to '`gambit@iro.umontreal.ca`'.

1.1 Accessing the Gambit runtime library

When Gambit is installed, its runtime library is normally compiled as a shared-library which is put in '`/usr/local/lib/`' under UNIX. 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, and '`PATH`' on Windows-NT/95. 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/lib gsi
```

Note that this is not a concern if Gambit was linked statically (i.e. Gambit was built with the command '`make FORCE_STATIC_LINK=yes`').

2 The Gambit Scheme interpreter

Synopsis:

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

The interpreter is executed in *interactive mode* when no command line argument is given other than options and the input does not come from a pipe. *Pipe mode* is when no command line argument is given and the input comes from a pipe. Finally, *batch mode* is when command line arguments are present. The '`-i`' option is ignored by the interpreter. The '`-e`' option may appear multiple times and must be after the '`-f`' and '`-i`' options. In all modes the expressions specified after each '`-e`' are evaluated from left to right in the interaction environment.

2.1 Interactive mode

In this mode the interpreter starts a read-eval-print loop (REPL) to interact with the user. The system prompts the user for a command, reads the command from standard input and executes it, sending any output generated including error messages to standard output.

The commands entered by the user are typically expressions that are to be evaluated. These expressions are evaluated in the global *interaction environment*. The REPL adds to this environment any definition entered using the `define` and `define-macro` special forms.

The result of evaluation is written to standard output unless it is the special "void" object. This object is returned by most procedures and special forms which the standard defines as returning an unspecified value (e.g. `write`, `set!`, `define`).

When an evaluation error occurs or the user interrupts the system (usually by typing `^C`), a *nested REPL* is initiated at the point of error, making it possible to inspect the context of the error. The prompt of nested REPLs includes the nesting level. An end of file (usually `^D` on UNIX and `^Z` on MSDOS and Windows-NT/95) will cause the current REPL to be aborted and the enclosing REPL (one nesting level less) to be resumed.

Gambit combines the standard REPL functions with those of the debugger. At any time the user can examine the frames in the REPL's continuation (which is the continuation of the error or the initial continuation). This is useful to determine which part of the program triggered an error and which chain of calls lead to the error.

Expressions entered at a nested REPL are evaluated in the environment 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 get 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` Terminate abruptly (i.e. quit the program).

`,t` Return to the outermost REPL.

`,d` Return to the enclosing REPL.

`,r` Return from the REPL with a specific value (the user is prompted for an expression to evaluate). This can also be used to resume a computation that was interrupted by the user (in this case the expression's value is ignored).

`,`*n* Move to frame number *n* of the continuation. Frames are numbered with non-positive integers. Frame 0 is the most recent frame of the continuation. Frame -1 is the next to most recent and so on. When it is different from 0, the frame number appears in the prompt after the REPL nesting level.

`,+` Move to the next frame of the continuation (i.e. towards the most recent).

`,-` Move to the previous frame of the continuation (i.e. towards the least recent).

`,b` Display a summary of each frame in the continuation starting with the current frame. There are 3 columns. The first is the frame number. The second is the procedure that created the frame or '`(interaction)`' if the frame was created by an expression entered at the REPL. The last column is the subproblem associated with the frame, that is the expression whose value is being computed. The third column is missing if the frame was created by a compiled procedure not compiled with the '`-debug`' option.

`,l` List all non-global variables in the current frame's environment. This command is only supported if the current frame was created by interpreted code.

`,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.

`,y` Display the subproblem associated with the current frame. The subproblem is not displayed if the frame was created by a compiled procedure not compiled with the '`-debug`' option.

Here is a sample interaction with gsi:

```
% gsi
Gambit Version 2.7

> (define (f x) (let* ((y 10) (z (* x y))) (- x z)))
> (define (g n) (if (> n 1) (+ 1 (g (/ n 2))) (f 'oops)))
> (g 8)
*** ERROR -- NUMBER expected
(* 'oops 10)
1> ,b
0    f                        (* x y)
-1   g                        (g (/ n 2))
-2   g                        (g (/ n 2))
-3   g                        (g (/ n 2))
-4   (interaction)            (g 8)
-5   ##initial-continuation
1> ,i
#<procedure f> =
(lambda (x) (let ((y 10)) (let ((z (* x y))) (- x z))))
1> ,y
(* x y)
1> ,l
y = 10
x = oops
1> (set! x 1)
1> ,l
y = 10
x = 1
1> ,r
Return value: (* x y)
-6
> ,q
```

2.2 Pipe mode

In pipe mode the interpreter evaluates the expressions read from standard input in the global interaction environment and writes each result on a separate line on standard output. Evaluation errors cause the interpreter to exit. Error messages are sent to standard error.

For example, under UNIX:

```
% echo "(sqrt (read)) 9 (expt 2 100)" | gsi
3
1267650600228229401496703205376
```

2.3 Batch mode

In batch mode the command line arguments designate files to be loaded. The interpreter loads these files in left-to-right order using the `load` procedure. The files can have no extension, or the extension '`.scm`' or '`.o`n' where n is a positive integer that acts as a version number (the '`.o`n' 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 '`.o`n' 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 name is completed with a '`.scm`' extension and the file is loaded as a Scheme source file.

A special case is the argument '`-`' which designates the standard input. If the standard input comes from a pipe, it is treated as a Scheme source file. Otherwise, a REPL is started so that the user can interact with the interpreter. Note that '`-`' can appear multiple times on the command line, which is useful for debugging.

The interpreter exits after loading the files or as soon as an error occurs. Input is taken from standard input and any output generated is sent to standard output except for error messages which go to standard error.

For example, under UNIX:

```
% cat m1.scm
(display "hello") (newline)
% cat m2.scm
(display "world") (newline)
% gsi m1 m2
hello
world
% gsi m1 - m2 -
hello
> (define display write)
> ,r
Return value: 0
"world"
> (+ 1 2)
3
> ,q
% echo "(write 123)(newline)" | gsi -
123
% echo "(write 123)(newline)" | gsi
123#<void>

#<void>
```

2.4 Customization

There are two ways to customize the interpreter. When the interpreter starts off it tries to execute a '`(load "~~/gambc")`' (for an explanation of how file names are interpreted see Chapter 5 [file names], page 17). 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 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: '`gambc.scm`' and '`~/gambc.scm`'. 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.

2.5 Process exit status

Under UNIX, the status is 0 when the interpreter exits normally and is 1 when the interpreter exits due to an error.

For example, if the shell is `sh`:

```
% echo "(/ 1 0)" | gsi
*** ERROR -- Division by zero
(/ 1 0)
% echo $?
1
```

2.6 Scheme scripts

Gambit's `load` procedure treats specially any Scheme source file beginning with the token '`#!`'. The `load` procedure discards the rest of the line and then loads the rest of the file normally. If this file is being loaded because it is an argument on the interpreter's command line, then the interpreter is terminated after loading the file.

This feature can be used under UNIX to write Scheme scripts by simply prefixing a file of Scheme code with a line containing '`#! /usr/local/bin/gsi`' (note the space between the '`#!`' and the '`/usr/local/bin/gsi`' so that the '`#!`' token is read properly by `gsi`). When such a script is executed, the script's file name followed by the script's command line arguments are added to the arguments passed to the interpreter. Thus, the interpreter will be run in batch mode and the interpreter will call `load` with the script's file name as argument. The script's arguments can be accessed by calling the procedure `argv`. This nullary procedure returns the script's file name and its arguments as a list of strings.

For example:

```
% cat upto
#! /usr/local/bin/gsi -f
(define (usage) (display "usage: upto n") (newline))
(if (not (= (length (argv)) 2))
  (usage)
  (let ((n (string->number (list-ref (argv) 1))))
    (if (and n (exact? n) (integer? n))
      (let loop ((i 1))
        (if (<= i n)
          (begin (write i) (newline) (loop (+ i 1)))))
      (usage))))
% upto 3
1
2
3
```

On some versions of UNIX it is necessary to prefix the Scheme source code in the following way:

```
#! /bin/sh
":";exec gsi -f $0 $*
(write (argv)) (newline)
```

An interesting application of Scheme scripts is to implement CGI scripts. Here is a sample CGI script that maintains a counter that is incremented each time the CGI script is accessed:

```
#! /usr/local/bin/gsi -f

(define n (+ 1 (with-input-from-file "counter" read)))
(with-output-to-file "counter" (lambda () (write n)))
(display "Content-type: text/html") (newline)
(newline)
(display "Access #") (display n) (newline)
```

3 The Gambit Scheme compiler

Synopsis:

gsc [-:*runtimeoption*,...] [-f] [-i] [-e *expression*] [-verbose] [-report] [-expansion]
[-gvm] [-debug] [-o *output*] [-c] [-dynamic] [-flat] [-l *base*] [*file*...]

3.1 Interactive and pipe modes

When no command line argument is present other than options the compiler behaves like the interpreter. This means that interactive mode is selected if the input does not come from a pipe, otherwise pipe mode is selected. In these modes, the only difference with the interpreter is that some additional predefined procedures are available (notably `compile-file`).

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', '.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', the C file '*file*.c' will be produced.

The C files produced by the compiler serve two purposes. They will have to be compiled by a C compiler to generate object files, and also they 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 Chapter 4 [runtime options], page 15). If present, the '-f' and '-i' compiler options must come first. The available options are:

`-f`	Do not examine initialization file.
`-i`	Force interpreter mode.
`-e` *expression*	Evaluate an expression in the interaction environment.
`-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.
`-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 an expression in the interaction environment.

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. This makes it possible for the REPL to display a more precise backtrace and for `pp` to display the source code of procedures. The debugging information is very large (it typically increases the size of the object file by a factor of 5).

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 ('.o*n*' 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. '~~/_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 all the Scheme modules. 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.

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 150 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 2.7

> (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.

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.

Some C compilers don't automatically search '`/usr/local/include`' for header files. In this case the flag '`-I/usr/local/include`' should be passed to the C compiler. Similarly, some C compilers/linkers don't automatically search '/usr/local/lib' for libraries. In this case the flag '-L/usr/local/lib' should be passed to the C compiler/linker.

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.

Under Digital UNIX, formerly DEC OSF/1, on DEC Alpha (a 64 bit processor) the Gambit runtime library is linked using the '-taso' C linker flag. This allows the use of 32 bit pointers instead of the usual 64 bit pointers, which roughly reduces the memory usage for data by a factor of two. The '-taso' flag must thus be passed to the C linker when linking a program. Gambit can be compiled to use 64 bit pointers by removing the definition '#define ___FORCE_32' from the file 'gambit.h'. The '-taso' C linker flag can then be omitted.

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:

`s`	Select standard Scheme mode.
`d`	Display debugging information.
`t`	Treat `stdin`, `stdout` and `stderr` as terminals.
`m`*heapsize*	Set minimum heap size in kilobytes.
`h`*heapsize*	Set maximum heap size in kilobytes.
`c`	Select native character encoding for I/O.
`1`	Select '`LATIN-1`' character encoding for I/O.
`8`	Select '`UTF-8`' character encoding for I/O.

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

The '`d`' option selects debugging mode which displays a trace on standard error to monitor the activity of the runtime system.

The '`t`' option forces the standard input and output to be treated like a terminal (i.e. as though `isatty` was true on `stdin`, `stdout` and `stderr`). This is useful in situations, such as running emacs under Windows-NT/95, where running the interpreter as a subprocess invokes pipe mode. By using the '`t`' option in this situation, the interpreter will enter interactive mode.

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 '`c`' option selects the native character encoding as the default character encoding for I/O. This is used by default if no default encoding is specified.

The '`1`' option selects '`LATIN-1`' as the default character encoding for I/O.

The '`8`' option selects '`UTF-8`' (variable length Unicode) as the default character encoding for I/O.

5 Handling of file names

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

A file is designated using a *path*. Each component of a path is separated by a '`/`' under UNIX, by a '`/`' or '`\`' under MSDOS and Windows-NT/95, and by a '`:`' under MACOS. A leading separator indicates an absolute path under UNIX, MSDOS and Windows-NT/95 but indicates a relative path under MACOS. A path which does not contain a path separator is relative to the *current working directory* on all operating systems (including MACOS). A drive specifier such as '`C:`' may prefix a file name under MSDOS and Windows-NT/95.

Under MACOS the folder '`Gambit-C`' must exist in the '`Preferences`' folder and contain the folder '`gambc`' (the Gambit installation directory). The '`Gambit-C`' and '`gambc`' folders must not be aliases.

In this document and the rest of this section in particular, '`/`' has been used to represent the path separator.

A path which starts with the characters '`~/`' designates a file in the user's home directory. The user's home directory is contained in the '`HOME`' environment variable under UNIX, MSDOS and Windows-NT/95. Under MACOS this designates the folder which contains the application.

A file name which starts with the characters '`~`*user*`/`' designates a file in the home directory of the given user. Under UNIX this is found using the password file. There is no equivalent under MSDOS, Windows-NT/95, and MACOS.

A file name which starts with the characters '`~~/`' designates a file in the Gambit installation directory. This directory is normally '`/usr/local/share/gambc/`' under UNIX, '`C:\GAMBC\`' under MSDOS and Windows-NT/95, and under MACOS the folder '`gambc`' in the '`Gambit-C`' folder. To override this binding under UNIX, MSDOS and Windows-NT/95, define the '`GAMBCDIR`' environment variable.

6 Extensions to Scheme

The Gambit Scheme system conforms to the R4RS 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.

6.1 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.

`(open-input-file` *file* [*char-encoding*]`)` procedure
`(open-output-file` *file* [*char-encoding*]`)` procedure
`(call-with-input-file` *file proc* [*char-encoding*]`)` procedure
`(call-with-output-file` *file proc* [*char-encoding*]`)` procedure
`(with-input-from-file` *file thunk* [*char-encoding*]`)` procedure
`(with-output-to-file` *file thunk* [*char-encoding*]`)` procedure
`(load` *file* [*char-encoding*]`)` procedure

These procedures take an optional argument which specifies the character encoding to use for I/O operations on the port. *char-encoding* must be one of the following symbols:

`char` the file is opened in text mode and the native character encoding is used

`latin1` the file is opened in text mode and the '`LATIN-1`' character encoding is used

`utf8` the file is opened in text mode and the '`UTF-8`' character encoding (1 to 6 bytes per character) is used

`byte` the file is opened in binary mode and the '`LATIN-1`' character encoding (1 byte per character) is used

`ucs2` the file is opened in binary mode and the '`UCS-2`' character encoding (2 bytes per character) is used

`ucs4` the file is opened in binary mode and the '`UCS-4`' character encoding (4 bytes per character) is used

If *char-encoding* is not specified, the default character encoding is used (see Chapter 4 [runtime options], page 15).

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

These procedures do nothing.

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

The `read` and `write` procedures support the following features.

- Keyword objects (see Section 6.2 [procedure `keyword?`], page 20).
- Extended character names:

`#\newline`	newline character
`#\space`	space character
`#\nul`	Unicode character 0
`#\backspace`	Unicode character 8
`#\tab`	Unicode character 9
`#\linefeed`	Unicode character 10
`#\page`	Unicode character 12
`#\return`	Unicode character 13
`#\rubout`	Unicode character 127

- 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.
- Special "`#!`" objects:

`#!`	script object
`#!eof`	end-of-file object
`#!optional`	optional object
`#!rest`	rest object
`#!key`	key object

- Bytevectors are uniform vectors containing raw numbers (non-negative exact integers or inexact reals). There are 5 types of bytevectors: '`u8vector`' (vector of 8 bit unsigned integers), '`u16vector`' (vector of 16 bit unsigned integers), '`u32vector`' (vector of 32 bit unsigned integers), '`f32vector`' (vector of 32 bit floating point numbers), and '`f64vector`' (vector of 64 bit floating point numbers). Bytevectors are represented similarly to normal vectors but with the '`#(`' prefix replaced respectively with '`#u8(`', '`#u16(`', '`#u32(`', '`#f32(`', and '`#f64(`'. The elements of the integer bytevectors must be signed or unsigned integers fitting in the given precision. The elements of the floating point bytevectors must be inexact reals.

6.2 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)))
```

`(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. *Variables* in *required-formal-arguments* are bound to successive actual arguments starting with the first actual argument. It shall be an error if there are fewer actual arguments than *required-formal-arguments*.

b. Next *variables* in *optional-formal-arguments* are bound to remaining actual arguments. If there are fewer remaining actual arguments than *optional-formal-arguments*, 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-arguments*. If there is no *rest-formal-argument* and there are no *keyword-formal-arguments*, 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-define-type` *name type*) special form
(`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

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 Chapter 7 [C-interface], page 41.

(`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 are printed out as '`#s(`*name* `(`*field value*`)...)`', where the *field*/*value* pair appears for each field and *value* is the value contained in the corresponding 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                            => #s(point (x 3) (y 5) (color red))
(point-x p)                  => 3
(point-color p)              => red
(point-color-set! p 'black) => #<void>
p                            => #s(point (x 3) (y 5) (color black))
```

`(trace var...)` special form

`(untrace var...)` special form

`trace` starts tracing calls to the specified procedures. `untrace` stops the tracing. The form `(trace)` returns the names of the currently traced procedures. The void object is returned by `trace` if it is passed one or more arguments. The form `(untrace)` stops the tracing on all those procedures and returns the void object.

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
```

`(file-exists? file)` procedure

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

`(flush-output [port])` procedure

`flush-output` causes all data buffered on the output port *port* to be written out. If *port* is not specified, the current output port is used.

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

`pretty-print` and `pp` are similar to `write` except that the result is nicely formatted. The argument *width* specifies the width of the page. 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.

(`open-input-pipe` *command* [*char-encoding*]) procedure
(`open-output-pipe` *command* [*char-encoding*]) procedure

These procedures open a pipe for input or output, and return a port. *Command* must be a string containing a shell command line. The command line is passed to the underlying shell (`/bin/sh` on most versions of UNIX) and the command's output (in the case of an input pipe) or its input (in the case of an output pipe) are available respectively for reading from the port and writing to the port.

The second argument, which is optional, specifies the character encoding to use for I/O operations on the port (see `open-input-file` for possible encodings).

Under MACOS these procedures always fail because there is no direct equivalent of pipes.

(`open-input-string` *string*) procedure
(`open-output-string`) procedure

These procedures implement string ports. 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. When the port is closed with a call to `close-output-port`, a string containing the characters accumulated is returned.

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)
```

```
        (close-output-port o)) => "\"cloud\"9"
```

(call-with-input-string *string proc*) procedure

(call-with-output-string *proc*) 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 string>"
```

(with-input-from-string *string thunk*) procedure

(with-output-to-string *thunk*) 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 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.

`(set-case-sensitive` *case-sensitive?* `)` procedure
`(set-read-keywords` *enabled?* `)` procedure

These procedures control the behavior of the reader (i.e. the `read` procedure and the parser used by the `load` procedure and the interpreter and compiler). If the argument to `set-case-sensitive` is `#f`, the reader will convert to lower case the letters in the tokens that are read, otherwise the case is preserved by the reader. The default is to preserve the case. The reader will only recognize keyword objects if the argument to `set-read-keywords` is not `#f`. Normally, the reader expects keywords to end with a colon (as in DSSSL) but if the argument to `set-read-keywords` is the symbol `prefix` then it expects keywords to start with a colon (as in Common Lisp). The default is to recognize keyword objects that end in a colon.

For example:

```
(set-case-sensitive #f)
(eq? 'TeX 'tex) => #t

(set-case-sensitive #t)
(eq? 'TeX 'tex) => #f

(set-read-keywords #f)
(symbol? 'foo:) => #t

(set-read-keywords #t)
(keyword? 'foo:) => #t ; quote not really needed

(set-read-keywords 'prefix)
(keyword? ':foo) => #t ; quote not really needed
```

`(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-read-keywords '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:
```

`(set-gc-report` *report?* `)` procedure

`set-gc-report` 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 proportion of the heap that contains live data, the size of the heap in kilobytes, and the number of bytes allocated to movable and non-movable objects.

`(make-will` *testator* [*action*]`)` procedure
`(will?` *obj*`)` procedure
`(will-testator` *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 nullary procedure. If no action procedure is supplied when `make-will` is called, the will contains an action procedure that does nothing.

An object is *finalizable* if all paths to the object from the roots (i.e. current continuation and global variables) pass through a will object. Note that by this definition an object that is not reachable 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 is finalizable the current computation is interrupted, the will's testator is set to `#f` and the will's action procedure is called. 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 action procedure may be a closure which retains a reference to the will's testator object. In such a case or if the testator object is reachable from another will object, 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 (even through will objects) that it is reclaimed by the garbage collector.

An counterintuitive feature of wills is that an action procedure can resurect an object after it has become finalizable. 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 ()
                   (display "executing action procedure")
                   (newline)
                   (set! c obj)))))
> (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
(eq? 'g2 (gensym))     => #f
(gensym 'star-trek-) => star-trek-3
```

`(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
```

`(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 2.7

> (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 2.7

> (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 2.7

> (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.

`(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 '`,r`' command causes a return from the call to `error`.

For example:

```
> (define (f x) (if (> x 0) (log x) (error "x must be positive")))
> (+ (f -4) 10)
*** ERROR -- x must be positive
1> ,r
Return value: 5
15
```

`(exit` [*status*]`)` procedure

`exit` causes the program to terminate with the status *status*. If it is not specified, the status defaults to 0.

`(argv)` procedure

`argv` 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 `argv` contains the script's file name followed by the remaining command line arguments.

`(getenv` *name*`)` procedure

`getenv` returns the value of the environment variable *name* (a string) of the current process. A string is returned if the environment variable is bound, otherwise `#f` is returned. Under MACOS `#f` is always returned.

`(runtime)` procedure

`runtime` returns the amount of time in seconds since the program was started. On most platforms process time is returned (user time plus system time) on the others real time is returned.

`(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 and how much time was spent in the garbage collector.

6.3 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 approximately 10 Hz. 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`.

`(##shell-command` *command*`)` procedure

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

`(##path-expand` *path*`)` procedure
`(##path-absolute?` *path*`)` procedure
`(##path-extension` *path*`)` procedure
`(##path-strip-extension` *path*`)` procedure
`(##path-directory` *path*`)` procedure
`(##path-strip-directory` *path*`)` procedure

These procedures manipulate file paths. `##path-expand` takes the relative or absolute path of a file or directory and returns the absolute path of the file or directory. The expanded path of a directory will always end with a path separator (i.e. '`/`', '`\`', or '`:`' depending on the operating system). If the path is the empty string, the current working directory is returned. `#f` is returned if the path is invalid.

The procedure `##path-absolute?` tests if the given path is absolute.

The remaining procedures extract various parts of a path. `##path-extension` returns the file extension (including the period) or the empty string if there is no extension. `##path-strip-extension` returns the path with the extension stripped off. `##path-directory` returns the file's directory (including the last path separator) or the empty string if no directory is specified in the path. `##path-strip-directory` returns the path with the directory stripped off.

`(dynamic-define` *var val*`)` special form
`(dynamic-ref` *var*`)` special form
`(dynamic-set!` *var val*`)` special form
`(dynamic-let ((`*var val*`)...)` *body*`)` special form

These special forms provide support for "dynamic variables" which have dynamic scope. Dynamic variables and normal (lexically scoped) variables are in different namespaces so there is no possible naming conflict between them. In all these special forms *var* is an identifier which names the dynamic variable. `dynamic-define` defines the global dynamic variable *var* (if it doesn't already exist) and assigns to it the value of *val*. `dynamic-let` has a syntax similar to `let`. It creates bindings of the given dynamic variables which are accessible for the duration of the evaluation of *body*. `dynamic-ref` returns the value currently bound to the dynamic variable *var*. `dynamic-set!` assigns the value of *val* to the dynamic variable *var*. The dynamic environment that was in effect when a continuation was created by `call-with-current-continuation` is restored when that continuation is invoked.

For example:

```
(dynamic-define radix 10)

(define (f x) (number->string x (dynamic-ref radix)))

(list (f 5) (f 15)) => ("5" "15")

(dynamic-let ((radix 2))
  (list (f 5) (f 15))) => ("101" "1111")
```

`(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

`(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

`(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

(`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

(`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

Bytevectors are uniform vectors containing raw numbers (non-negative exact integers or inexact reals). There are 5 types of bytevectors: '`u8vector`' (vector of 8 bit unsigned integers), '`u16vector`' (vector of 16 bit unsigned integers), '`u32vector`' (vector of 32 bit unsigned integers), '`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 -1 13))
(u8vector-set! v 2 99)

v => #u8(10 255 99)

(u8vector-ref v 1) => 255

(u8vector->list v) => (10 255 99)
```

6.4 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 '`-:8`' when `gsi` and `gsc` are started.

7 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 datatypes between C and Scheme is discussed in the next section. The remaining sections of this chapter describe each special form of the C-interface.

7.1 The mapping of types between C and Scheme

Scheme and C do not provide the same set of built-in datatypes 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. For the sake of explaining the mapping, we assume that Scheme and C have been augmented with some new datatypes. To Scheme is added the datatype '`C-pointer`' to support the C concept of pointer. The following datatypes are added to C:

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

`boolean`
: denotes the C '`int`' type when used as a boolean

`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

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

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

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

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

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

Scheme notation	C type
`void`	`void`
`boolean`	`boolean`
`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`
`float`	`float`
`double`	`double`
`(struct "`*name*`")`	`struct` *name*
`(union "`*name*`")`	`union` *name*
`(pointer` *type*`)`	*T*`*` (where *T* is the C equivalent of *type* which must be the Scheme notation of a C type)
`(function (`*type1*...`)` *result-type*`)`	function with the given argument types and result type
`char-string`	`char-string`
`latin1-string`	`latin1-string`
`ucs2-string`	`ucs2-string`
`ucs4-string`	`ucs4-string`
`utf8-string`	`utf8-string`

`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* (where *c-type-id* is an identifier naming a C type, for example: `"FILE"` and `"time_t"`)

Note that not all of these types can be used in all contexts. In particular the arguments and result of functions defined with `c-lambda` and `c-define` can not be `(struct "`*name*`")` or `(union "`*name*`")` or `"`*c-type-id*`"`. On the other hand, pointers to these types are acceptable.

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`; `boolean`; any string, pointer or function type
boolean `#t`	`scheme-object`; `boolean`
character	`scheme-object`; `boolean`; [[`un`]`signed`] `char`; `latin1`; `ucs2`; `ucs4`
exact integer	`scheme-object`; `boolean`; [`unsigned`] `short/int/long`
inexact real	`scheme-object`; `boolean`; `float`; `double`
string	`scheme-object`; `boolean`; any string type
'`C-pointer`'	`scheme-object`; `boolean`; any pointer type
vector	`scheme-object`; `boolean`
symbol	`scheme-object`; `boolean`
procedure	`scheme-object`; `boolean`; any function type
other objects	`scheme-object`; `boolean`

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
`boolean`	boolean
character types	character
integer types	exact integer
`float/double`	inexact real
string types	string or `#f` if it is equal to '`NULL`'
pointer types	'`C-pointer`' or `#f` if it is equal to '`NULL`'
function types	procedure or `#f` if it is equal to '`NULL`'

`void` void object

All Scheme types are compatible with the C types `scheme-object` and `boolean`. 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 `boolean` 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 any C string type, C pointer type, or C function type is expected. In this case, `#f` is converted to the '`NULL`' pointer. C `booleans` are extended booleans so any value different from 0 represents true. Thus, a C `boolean` 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 the C types `short`, `int`, and `long` are 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 `short`, `int` and `long` 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` or `double` value. C `float` and `double` 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. Any C string type passed to the Scheme environment causes the creation of a fresh Scheme string containing a copy of the C string.

A C pointer passed to the Scheme environment causes the creation and initialization of a new '`C-pointer`' object. This object is simply a cell containing the pointer to a memory location in the C environment. The pointer is ignored by the garbage collector. As a special case, the '`NULL`'

C pointer is converted to `#f`. A Scheme '`C-pointer`' and `#f` can be passed to the C environment where a C pointer is expected. The conversion simply recreates the original C pointer or '`NULL`' pointer.

Only Scheme procedures defined with the `c-define` special form and `#f` can be passed where a C function is expected. Conversion from C functions to Scheme procedures is not currently implemented.

7.2 The `c-define-type` special form

Synopsis:

(`c-define-type` *name* *type*)

This form defines the type identifier *name* to be equivalent to the C type *type*. After this definition, the use of *name* in a type specification is synonymous to *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.

For example:

```
(c-define-type FILE "FILE")
(c-define-type FILE* (pointer FILE))
(c-define-type time-struct-ptr (pointer (struct "tms")))
```

Note that Scheme identifiers are not case sensitive. Nevertheless it is good programming practice to use a *name* with the same case as in C.

7.3 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;
")
```

7.4 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 ();")
```

7.5 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 (see below for supported conversions).

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`'. If no result needs to be returned, the *result-type* should be `void` and no assignment to the variable '`___result`' 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`'. 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 '`___result`'.

When passed to the Scheme environment, the C `void` type is converted to the void object.

For example:

```
(define fopen
  (c-lambda (char-string char-string) FILE* "fopen"))

(define fgetc
  (c-lambda (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 () char-string "___result = host;")))

(define stdin ((c-lambda () FILE* "___result = stdin;")))

((c-lambda () void
  "printf( \"hello\\n\" ); printf( \"world\\n\" );"))

(define pack-1-char (c-lambda (char) 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) char-string
"
char s[3]; s[0] = ___arg1; s[1] = ___arg2; s[2] = 0; ___result = s;
"))
```

7.6 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 '`check/server.scm`'.

7.7 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.

8 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!`).
- Write/read invariance of inexact numbers is not obeyed. This means that an inexact number written out to a file with `write` may be read back by `read` as a slightly different number.
- Records (defined through `define-structure`) can be written with `write` but can not be read by `read`.
- On MSDOS and Windows-NT/95, `^C` is sometimes interpreted as `^Z` (i.e. an end-of-file).

9 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 non-null 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.

10 Copyright and distribution information

General Index

#

,

-

.

_

+

T

U

V

W

Table of Contents