

Gambit-C, version 4.0 beta 1

A portable implementation of Scheme
Edition 4.0 beta 1, July 2002

Marc Feeley

Copyright © 1994-2002 Marc Feeley.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the copyright holder.

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, R5RS 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 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.

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>' or send mail to 'gambit@iro.umontreal.ca'.

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

1.1 Accessing the Gambit system files

Unless the default is overridden, all files of the Gambit-C system are installed in '`/usr/local/Gambit-C`' under UNIX 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. 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/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.

2 The Gambit Scheme interpreter

Synopsis:

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

The interpreter is executed in *interactive mode* when no command line argument is given other than options. When command line arguments are present the interpreter is executed in *batch mode*. 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 a read-eval-print loop (REPL) is started for interacting with the interpreter. In 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. Commands can produce 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*. Normally the interaction channel is the standard input (‘`stdin`’) and standard output (‘`stdout`’) of the interpreter (for details see Chapter 4 [runtime options], page 18). So unless I/O redirection is used, the interaction channel will correspond to the user’s *console*, also known as the *controlling terminal* in the UNIX world. The standard error (‘`stderr`’) of the interpreter is not used for REPL interaction.

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`).

When execution starts, the ports associated with (‘`(current-input-port)`’), (‘`(current-output-port)`’) and (‘`(current-error-port)`’) all refer to the interaction channel.

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

- An evaluation error has occurred, such as attempting to divide by zero.
- The user has interrupted the evaluation (usually by typing `⌘C`).
- A breakpoint has been reached or (`step`) was evaluated.
- 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 (‘`(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. This allows the inspection of the

evaluation context, which is particularly useful to determine the exact location and cause of an error.

The prompt of nested REPLs includes the nesting level. An end of file (usually `⏏` on UNIX and `⏏` on MSDOS and Microsoft Windows) will cause the current REPL to be aborted 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. 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 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:

<code>,?</code>	Give a summary of the REPL commands.
<code>,q</code>	Quit the program (i.e. terminate abruptly).
<code>,t</code>	Return to the outermost REPL, also known as the “top-level REPL”.
<code>,d</code>	Leave the current REPL and resume the enclosing REPL. This command does nothing in the top-level REPL.
<code>,(c <i>expr</i>)</code>	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 <i>expr</i> 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 <code>(+ (/ x y) 1)</code> signaled an error because ‘y’ is zero, then in the nested REPL a <code>,(c (* 2 3))</code> will resume the computation of <code>(+ (/ x y) 1)</code> as though the value of <code>(/ x y)</code> was 6. This command must be used carefully because the context where the error occurred may rely on the result being of a particular type. For instance a <code>,(c #f)</code> in the previous example will cause ‘+’ to signal a type error (this problem is extremely serious when debugging Scheme code that was compiled with type checking turned off so be careful).
<code>,c</code>	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.
<code>,s</code>	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 <code>step-level-set!</code>) 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 <code>trace</code>) 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.

- `,1` 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. Frames are numbered with nonnegative integers. Frame 0 is the most recently created frame in the chain of continuation frames. 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. 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 than 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. Com-

piled procedures will only be pretty printed if compiled with the ‘-debug’ option.

,e Display the environment which is accessible from the current frame. The environment contains the value of local variables (only if the frame was created by interpreted code) and the dynamically-bound parameters.

Here is a sample interaction with `gsi`:

```
% gsi
Gambit Version 4.0 beta 1

> (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 IN (stdin)@1.32 -- NUMBER expected
(* 'oops 10)
1> ,i
#<procedure f> =
(lambda (x) (let ((y 10)) (let ((z (* x y))) (- x z))))
1> ,b
0 f                (stdin)@1:32          (* x y)
1 g                (stdin)@2:32          (g (/ n 2))
2 g                (stdin)@2:32          (g (/ n 2))
3 g                (stdin)@2:32          (g (/ n 2))
4 (interaction)    (stdin)@3:1           (g 8)
5 ##initial-continuation
1> ,e
y = 10
x = oops
1> ,+
1 g                (stdin)@2.32          (g (/ n 2))
1-1> ,e
n = 2
1-1> ,+
2 g                (stdin)@2.32          (g (/ n 2))
1-2> ,e
n = 4
1-2> ,+
3 g                (stdin)@2.32          (g (/ n 2))
1-3> ,e
n = 8
1-3> ,0
0 f                (stdin)@1.32          (* x y)
1> (set! x 1)
1> ,e
y = 10
x = 1
1> ,(c (* x y))
-6
```

```
> ,q
```

2.2 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 `‘.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 Chapter 7 [Scheme extensions], page 23).

When execution starts, the ports associated with `‘(current-input-port)’`, `‘(current-output-port)’` and `‘(current-error-port)’` refer respectively to the standard input (`‘stdin’`), standard output (`‘stdout’`) and the standard error (`‘stderr’`) of the interpreter.

The interpreter exits after loading the files or as soon as an error occurs.

For example, under UNIX:

```
% cat m1.scm
(display "hello") (newline)
% cat m2.scm
(display "world") (newline)
% gsi m1 m2
hello
world
```

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 Chapter 5 [file names], page 20). 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: `‘gambcini’` and `‘~/gambcini’` (with a `‘.scm’`, `‘.six’` or no extension). 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

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

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 `'#!'` or `'@;'`. 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 `'#!'` or `'@;'` the system will search for the first substring matching one of the following language specifying tokens:

<code>scheme-r4rs</code>	R4RS language with prefix syntax, case insensitivity, keyword syntax not supported
<code>scheme-r5rs</code>	R5RS language with prefix syntax, case insensitivity, keyword syntax not supported
<code>scheme-ieee-1178-1990</code>	IEEE 1178-1990 language with prefix syntax, case insensitivity, keyword syntax not supported
<code>scheme-srfi-0</code>	Full Gambit Scheme language with prefix syntax, case sensitivity, keyword syntax supported
<code>scheme-six</code>	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 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.
- After the script is loaded the procedure `main` is called with the command-line arguments. The way this is done depends on the format of the script's first line. If there is a space after the two characters `'#!'` or `'@;'` then `main` 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. If there is no space after the two characters `'#!'` or `'@;'` then `main` is called with the equivalent of `(apply main (cdr`

(`command-line`))) and the process exit status code is 0 (`main`'s result is ignored). Note that 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 terminates. That is the command-line arguments after a script file are not taken to be other files to load.

Under UNIX, the Gambit-C installation process will have created the executable '`gsi`' and also the executables '`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 '`#!`', possibly a space, 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 the files in the current directory:

```
#!/usr/local/Gambit-C/bin/scheme-srfi-0
(display "files:\n")
(pp (directory-files (current-directory)))
```

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

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

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

Under UNIX, 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 scheme-r5rs

(define (main arguments)
  (for-each display-file (cdr arguments))
  0)

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

Under Microsoft Windows, the Gambit-C installation process will have created the executable '`gsi.exe`' and also the batch files '`scheme-r5rs.bat`', '`scheme-srfi-0.bat`', etc which simply invoke '`gsi.exe`'. A Scheme script need only start with the name of the desired Scheme language variant prefixed with '`@;`' and possibly a space. 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:

```
@;scheme-srfi-0 %~f0 %*  
(display "files:\n")  
(pp (directory-files (current-directory)))
```

Note that Microsoft Windows always searches executables in the user's 'PATH', so there is no need for an indirection through '`/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,...] [-f] [-i] [-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 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’, ‘.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 Chapter 4 [runtime options], page 18). 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 *expressions* Evaluate expressions in the interaction environment.
- prelude *expressions* Add expressions to the top of the source code being compiled.

<code>-postlude</code>	<i>expressions</i>	Add expressions to the bottom of the source code being compiled.
<code>-cc-options</code>	<i>options</i>	Add options to the command that invokes the C compiler.
<code>-ld-options</code>	<i>options</i>	Add options to the command that invokes the C linker.
<code>-warnings</code>		Display warnings.
<code>-verbose</code>		Display a trace of the compiler's activity.
<code>-report</code>		Display a global variable usage report.
<code>-expansion</code>		Display the source code after expansion.
<code>-gvm</code>		Generate a listing of the GVM code.
<code>-debug</code>		Include debugging information in the code generated.
<code>-track-scheme</code>		Generate <code>#line</code> directives referring back to the Scheme code.
<code>-o</code>	<i>output</i>	Set name of output file.
<code>-c</code>		Only compile Scheme source files to C (no link file generated).
<code>-dynamic</code>		Only compile Scheme source files to dynamically loadable object files (no link file generated).
<code>-flat</code>		Generate a flat link file instead of an incremental link file.
<code>-l</code>	<i>base</i>	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 1

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


The warnings indicate that there are no definitions (`defines` 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 [\[Top\]](#), page [\[Top\]](#).

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.

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:

<code>mheapsize</code>	Set minimum heap size in kilobytes.
<code>hheapsize</code>	Set maximum heap size in kilobytes.
<code>llivepercent</code>	Set heap occupation after garbage collection.
<code>s</code>	Select standard Scheme mode.
<code>S</code>	Select Gambit Scheme mode.
<code>d[OPT...]</code>	Set debugging options.
<code>t[OPT...]</code>	Set terminal options.
<code>cencoding</code>	Set default character encoding for I/O.
<code>=directory</code>	Override the Gambit installation directory.
<code>+argument</code>	Add <i>argument</i> 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 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 will resize 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.

<code>p</code>	Uncaught exceptions will be treated as “errors” in primordial thread only.
<code>a</code>	Uncaught exceptions will be treated as “errors” in all threads.
<code>r</code>	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 <i>level</i> (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 various 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 '(pp (path-expand "~~")) (/ 1 0)'
"/u/feeley/my-gambit2/"
% echo $?
1
% gsi -:d1 -e '(pp (path-expand "~~")) (/ 1 0)'
"/u/feeley/my-gambit2/"
*** ERROR IN string@1.25 -- Divide by zero
(/ 1 0)
```

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 Microsoft Windows, and by a ‘:’ under Classic Mac OS. A leading separator indicates an absolute path under UNIX, 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 drive 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 and must not be an alias.

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 Microsoft Windows. Under Classic Mac OS 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, Microsoft Windows, and Classic Mac OS.

A file name which starts with the characters ‘~~/’ designates a file in the Gambit installation directory. This directory is normally ‘/usr/local/Gambit-C/’ under UNIX, ‘C:\Gambit-C\’ under MSDOS and Microsoft Windows, and under Classic Mac OS the ‘Gambit-C’ folder. To override this binding under UNIX, MSDOS and Microsoft Windows, define the ‘GAMBCOPT’ environment variable.

6 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):

<code>C-x C-e</code>	Evaluate the expression which is before the cursor (the expression will be copied to the Gambit interaction buffer).
<code>C-c C-z</code>	Switch to Gambit interaction buffer.
<code>C-c C-l</code>	Load a file (file attached to current buffer is default) using (<code>load file</code>).
<code>C-c C-k</code>	Compile a file (file attached to current buffer is default) using (<code>compile-file file</code>).

The file ‘`gambit.el`’ provides these additional commands:

<code>C-c c</code>	Continue the computation (same as typing ‘ <code>,c</code> ’ to the REPL).
<code>C-c s</code>	Step the computation (same as typing ‘ <code>,s</code> ’ to the REPL).
<code>C-c l</code>	Leap the computation (same as typing ‘ <code>,l</code> ’ to the REPL).
<code>C-c [</code>	Move to older frame (same as typing ‘ <code>,+</code> ’ to the REPL).
<code>C-c]</code>	Move to newer frame (same as typing ‘ <code>,-</code> ’ to the REPL).
<code>C-c _</code>	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)
```


7 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.

7.1 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.

<code>(open-file <i>path</i> [<i>settings</i>])</code>	procedure
<code>(open-input-file <i>path</i> [<i>settings</i>])</code>	procedure
<code>(open-output-file <i>path</i> [<i>settings</i>])</code>	procedure
<code>(call-with-input-file <i>path</i> <i>proc</i> [<i>settings</i>])</code>	procedure
<code>(call-with-output-file <i>path</i> <i>proc</i> [<i>settings</i>])</code>	procedure
<code>(with-input-from-file <i>path</i> <i>thunk</i> [<i>settings</i>])</code>	procedure
<code>(with-output-to-file <i>path</i> <i>thunk</i> [<i>settings</i>])</code>	procedure

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

*** This documentation is incomplete!

<code>(transcript-on <i>file</i>)</code>	procedure
<code>(transcript-off)</code>	procedure

These procedures do nothing.

<code>(read [<i>port</i> [<i>readtable</i>]])</code>	procedure
<code>(write <i>obj</i> [<i>port</i> [<i>readtable</i>]])</code>	procedure
<code>(display <i>obj</i> [<i>port</i> [<i>readtable</i>]])</code>	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 Section 7.2 [procedure `keyword?`], page 29).
- Extended character names:

<code>#\newline</code>	newline character
<code>#\space</code>	space character
<code>#\nul</code>	Unicode character 0
<code>#\bel</code>	Unicode character 7
<code>#\backspace</code>	Unicode character 8
<code>#\tab</code>	Unicode character 9
<code>#\linefeed</code>	Unicode character 10

<code>#\vt</code>	Unicode character 11
<code>#\page</code>	Unicode character 12
<code>#\return</code>	Unicode character 13
<code>#\rubout</code>	Unicode character 127
<code>#\n</code>	Unicode character <i>n</i> (<i>n</i> must start with a “#” character and it must represent an exact integer, for example <code>#\#x20</code> is the space character, <code>#\#d9</code> is the tab character, and <code>#\#e1.2e2</code> is the lower case character “x”)

- Escape sequences inside character strings:

<code>\n</code>	newline character
<code>\a</code>	Unicode character 7
<code>\b</code>	Unicode character 8
<code>\t</code>	Unicode character 9
<code>\v</code>	Unicode character 11
<code>\f</code>	Unicode character 12
<code>\r</code>	Unicode character 13
<code>\"</code>	"
<code>\\</code>	\
<code>\ooo</code>	character encoded in octal (1 to 3 octal digits)
<code>\xhh</code>	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.
- Multiline comments are delimited by the tokens ‘#|’ and ‘|#’. These comments can be nested.
- Special “#!” objects:

<code>#!eof</code>	end-of-file object
<code>#!void</code>	void object
<code>#!optional</code>	optional object
<code>#!rest</code>	rest object
<code>#!key</code>	key object

- Special inexact real numbers:

<code>+inf.</code>	positive infinity
--------------------	-------------------

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>) <stat>	(six.for \$3 \$5 \$7 \$9)
<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> ::=	

<code><expr12> ? <expr> : <expr13></code> <code> <expr12></code>	<code>(six.x?y:z \$1 \$3 \$5)</code> <code>\$1</code>
<code><expr12> ::=</code> <code><expr12> <expr11></code> <code> <expr11></code>	<code>(six.x\\ \\ y \$1 \$3)</code> <code>\$1</code>
<code><expr11> ::=</code> <code><expr11> && <expr10></code> <code> <expr10></code>	<code>(six.x&&y \$1 \$3)</code> <code>\$1</code>
<code><expr10> ::=</code> <code><expr10> <expr9></code> <code> <expr9></code>	<code>(six.x\\ y \$1 \$3)</code> <code>\$1</code>
<code><expr9> ::=</code> <code><expr9> ^ <expr8></code> <code> <expr8></code>	<code>(six.x^y \$1 \$3)</code> <code>\$1</code>
<code><expr8> ::=</code> <code><expr8> & <expr7></code> <code> <expr7></code>	<code>(six.x&y \$1 \$3)</code> <code>\$1</code>
<code><expr7> ::=</code> <code><expr7> != <expr6></code> <code> <expr7> == <expr6></code> <code> <expr6></code>	<code>(six.x!=y \$1 \$3)</code> <code>(six.x==y \$1 \$3)</code> <code>\$1</code>
<code><expr6> ::=</code> <code><expr6> < <expr5></code> <code> <expr6> <= <expr5></code> <code> <expr6> > <expr5></code> <code> <expr6> >= <expr5></code> <code> <expr5></code>	<code>(six.x<y \$1 \$3)</code> <code>(six.x<=y \$1 \$3)</code> <code>(six.x>y \$1 \$3)</code> <code>(six.x>=y \$1 \$3)</code> <code>\$1</code>
<code><expr5> ::=</code> <code><expr5> << <expr4></code> <code> <expr5> >> <expr4></code> <code> <expr4></code>	<code>(six.x<<y \$1 \$3)</code> <code>(six.x>>y \$1 \$3)</code> <code>\$1</code>
<code><expr4> ::=</code> <code><expr4> + <expr3></code> <code> <expr4> - <expr3></code> <code> <expr3></code>	<code>(six.x+y \$1 \$3)</code> <code>(six.x-y \$1 \$3)</code> <code>\$1</code>
<code><expr3> ::=</code> <code><expr3> % <expr2></code> <code> <expr3> * <expr2></code> <code> <expr3> / <expr2></code> <code> <expr2></code>	<code>(six.x%y \$1 \$3)</code> <code>(six.x*y \$1 \$3)</code> <code>(six.x/y \$1 \$3)</code> <code>\$1</code>

```

<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-declaration
  <block> $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.

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

7.2 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:

- | | |
|-------------------------|--|
| <code>(dialect)</code> | Use the given dialect's semantics. <i>dialect</i> can be: <code>'ieee-scheme'</code> or <code>'r4rs-scheme'</code> . |
| <code>(strategy)</code> | 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. <i>strategy</i> can be: <code>'block'</code> or <code>'separate'</code> . |

- `([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 <i>lambda-formals body</i>)	special form
(define (<i>variable define-formals</i>) <i>body</i>)	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)

```

<code>(c-declare <i>c-declaration</i>)</code>	special form
<code>(c-initialize <i>c-code</i>)</code>	special form
<code>(c-lambda (<i>type1...</i>) <i>result-type c-name-or-code</i>)</code>	special form
<code>(c-define (<i>variable define-formals</i>) (<i>type1...</i>) <i>result-type</i> <i>c-name scope body</i>)</code>	special form
<code>(c-define-type <i>name type</i> [<i>c-to-scheme scheme-to-c</i> <i>cleanup</i>])</code>	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 9 [C-interface], page 59.

`(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>
```

`(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 procedure
> (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 procedure
> (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.

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

`(force-output [port])` procedure
`force-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])` 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 `uppercase`, 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 needed
#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:
```

<code>(gc-report-set! report?)</code>	procedure
<code>gc-report-set!</code> 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.	

<code>(port? obj)</code>	procedure
<code>(close-port port)</code>	procedure
<code>(input-port-timeout-set! port timeout [think])</code>	procedure
<code>(output-port-timeout-set! port timeout [think])</code>	procedure

<code>(input-port-line port)</code>	procedure
<code>(input-port-column port)</code>	procedure
<code>(output-port-line port)</code>	procedure
<code>(output-port-column port)</code>	procedure
<code>(output-port-width port)</code>	procedure

<code>(read-all [port [reader]])</code>	procedure
<code>(read-line [port [separator [include-separator?]]])</code>	procedure
<code>(read-byte [port])</code>	procedure
<code>(write-byte n [port])</code>	procedure
<code>(read-subu8vector u8vector start end [port])</code>	procedure
<code>(write-subu8vector u8vector start end [port])</code>	procedure

<code>(tty? port)</code>	procedure
<code>(tty-type-set! port term-type emacs-bindings)</code>	procedure
<code>(tty-text-attributes-set! port input output)</code>	procedure
<code>(tty-history port)</code>	procedure
<code>(tty-history-set! port history)</code>	procedure
<code>(tty-max-history-length-set! port max-length)</code>	procedure
<code>(tty-paren-balance-duration-set! port duration)</code>	procedure
<code>(tty-mode-set! port mode)</code>	procedure
<code>(open-vector [vector [settings]])</code>	procedure
<code>(open-input-vector vector [settings])</code>	procedure
<code>(open-output-vector [settings])</code>	procedure
<code>(get-output-vector port)</code>	procedure
<code>(call-with-input-vector vector proc [settings])</code>	procedure
<code>(call-with-output-vector proc [settings])</code>	procedure
<code>(with-input-from-vector vector thunk [settings])</code>	procedure
<code>(with-output-to-vector thunk [settings])</code>	procedure
<code>(open-string [string [settings]])</code>	procedure
<code>(open-input-string string [settings])</code>	procedure
<code>(open-output-string [settings])</code>	procedure
<code>(get-output-string port)</code>	procedure
<code>(call-with-input-string string proc [settings])</code>	procedure
<code>(call-with-output-string proc [settings])</code>	procedure
<code>(with-input-from-string string thunk [settings])</code>	procedure
<code>(with-output-to-string thunk [settings])</code>	procedure
<code>(open-u8vector [u8vector [settings]])</code>	procedure
<code>(open-input-u8vector u8vector [settings])</code>	procedure
<code>(open-output-u8vector [settings])</code>	procedure
<code>(get-output-u8vector port)</code>	procedure
<code>(call-with-input-u8vector u8vector proc [settings])</code>	procedure
<code>(call-with-output-u8vector proc [settings])</code>	procedure
<code>(with-input-from-u8vector u8vector thunk [settings])</code>	procedure
<code>(with-output-to-u8vector thunk [settings])</code>	procedure
<code>(directory-files path)</code>	procedure
<code>(create-directory path [settings])</code>	procedure
<code>(create-fifo path [settings])</code>	procedure
<code>(create-link source-path destination-path)</code>	procedure
<code>(create-symbolic-link source-path destination-path)</code>	procedure
<code>(rename-file source-path destination-path)</code>	procedure
<code>(copy-file source-path destination-path)</code>	procedure
<code>(delete-file path)</code>	procedure
<code>(delete-directory path)</code>	procedure

<code>(current-console-port [new-value])</code>	procedure
<code>(current-directory [new-value])</code>	procedure
<code>(current-error-port [new-value])</code>	procedure
<code>(current-exception-handler [new-value])</code>	procedure
<code>(current-input-port [new-value])</code>	procedure
<code>(current-output-port [new-value])</code>	procedure
<code>(current-readtable [new-value])</code>	procedure
<code>(current-user-interrupt-handler [new-value])</code>	procedure
<code>(file-info path)</code>	procedure
<code>(file-info? object)</code>	procedure
<code>(file-info-attributes file-info)</code>	procedure
<code>(file-info-creation-time file-info)</code>	procedure
<code>(file-info-device file-info)</code>	procedure
<code>(file-info-group file-info)</code>	procedure
<code>(file-info-inode file-info)</code>	procedure
<code>(file-info-last-access-time file-info)</code>	procedure
<code>(file-info-last-change-time file-info)</code>	procedure
<code>(file-info-last-modification-time file-info)</code>	procedure
<code>(file-info-mode file-info)</code>	procedure
<code>(file-info-number-of-links file-info)</code>	procedure
<code>(file-info-owner file-info)</code>	procedure
<code>(file-info-size file-info)</code>	procedure
<code>(file-info-type file-info)</code>	procedure
<code>(file-attributes path)</code>	procedure
<code>(file-creation-time path)</code>	procedure
<code>(file-device path)</code>	procedure
<code>(file-group path)</code>	procedure
<code>(file-inode path)</code>	procedure
<code>(file-last-access-time path)</code>	procedure
<code>(file-last-change-time path)</code>	procedure
<code>(file-last-modification-time path)</code>	procedure
<code>(file-mode path)</code>	procedure
<code>(file-number-of-links path)</code>	procedure
<code>(file-owner path)</code>	procedure
<code>(file-size path)</code>	procedure
<code>(file-type path)</code>	procedure
<code>(group-info group-name-or-id)</code>	procedure
<code>(group-info? object)</code>	procedure
<code>(group-info-gid group-info)</code>	procedure
<code>(group-info-members group-info)</code>	procedure
<code>(group-info-name group-info)</code>	procedure

<code>(user-info user-name-or-id)</code>	procedure
<code>(user-info? object)</code>	procedure
<code>(user-info-gid user-info)</code>	procedure
<code>(user-info-home user-info)</code>	procedure
<code>(user-info-name user-info)</code>	procedure
<code>(user-info-shell user-info)</code>	procedure
<code>(user-info-uid user-info)</code>	procedure
<code>(host-info host-name-or-address)</code>	procedure
<code>(host-info? object)</code>	procedure
<code>(host-info-addresses host-info)</code>	procedure
<code>(host-info-aliases host-info)</code>	procedure
<code>(host-info-name host-info)</code>	procedure
<code>(open-file path [settings])</code>	procedure
<code>(open-input-file path [settings])</code>	procedure
<code>(open-output-file path [settings])</code>	procedure
<code>(open-string string [settings])</code>	procedure
<code>(open-input-string string [settings])</code>	procedure
<code>(open-output-string string [settings])</code>	procedure
<code>(open-vector vector [settings])</code>	procedure
<code>(open-input-vector vector [settings])</code>	procedure
<code>(open-output-vector vector [settings])</code>	procedure
<code>(open-u8vector u8vector [settings])</code>	procedure
<code>(open-input-u8vector u8vector [settings])</code>	procedure
<code>(open-output-u8vector u8vector [settings])</code>	procedure
<code>(open-directory path [settings])</code>	procedure
<code>(open-process path-and-arguments [environment [settings]])</code>	procedure
<code>(open-tcp-client host-name-or-address port-number [settings])</code>	procedure
<code>(open-tcp-server port-number [settings])</code>	procedure
<code>(readtable? object)</code>	procedure
<code>(readtable-copy readtable)</code>	procedure
<code>(readtable-case-conversion?-set! readtable new-value)</code>	procedure
<code>(readtable-eval-allowed?-set! readtable new-value)</code>	procedure
<code>(readtable-keywords-allowed?-set! readtable new-value)</code>	procedure
<code>(readtable-max-write-length-set! readtable new-value)</code>	procedure
<code>(readtable-max-write-level-set! readtable new-value)</code>	procedure
<code>(readtable-sharing-allowed?-set! readtable new-value)</code>	procedure
<code>(readtable-sharp-quote-keyword-set! readtable new-value)</code>	procedure
<code>(readtable-start-syntax-set! readtable new-value)</code>	procedure
<code>(serial-number->object integer)</code>	procedure
<code>(with-exception-catcher handler thunk)</code>	procedure
<code>(with-exception-handler handler thunk)</code>	procedure

<code>(bitwise-ior n...)</code>	procedure
<code>(bitwise-xor n...)</code>	procedure
<code>(bitwise-and n...)</code>	procedure
<code>(bitwise-not n)</code>	procedure
<code>(arithmetic-shift n1 n2)</code>	procedure
<code>(bit-count n)</code>	procedure
<code>(integer-length n)</code>	procedure
<code>(bitwise-merge n1 n2 n3)</code>	procedure
<code>(bit-set? n1 n2)</code>	procedure
<code>(any-bits-set? n1 n2)</code>	procedure
<code>(all-bits-set? n1 n2)</code>	procedure
<code>(first-set-bit n)</code>	procedure
<code>(extract-bit-field n1 n2 n3)</code>	procedure
<code>(test-bit-field? n1 n2 n3)</code>	procedure
<code>(clear-bit-field n1 n2 n3)</code>	procedure
<code>(insert-bit-field n1 n2 n3 n4)</code>	procedure
<code>(copy-bit-field n1 n2 n3 n4)</code>	procedure
<code>(box? obj)</code>	procedure
<code>(box obj)</code>	procedure
<code>(unbox box)</code>	procedure
<code>(set-box! box val)</code>	procedure

*** This documentation is incomplete!

<code>(make-will testator action)</code>	procedure
<code>(will? obj)</code>	procedure
<code>(will-testator will)</code>	procedure
<code>(will-execute! will)</code>	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
```

(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 `‘.on’`, 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 `‘.on’` 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 1

> (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 1

> (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 1

> (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 (`defines` 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 `'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

```

(exit [status]) procedure
exit causes the program 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.

(command-line) 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 file name followed by the remaining command line arguments.

(getenv name [default]) procedure
(setenv name value) 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 *default* is returned if it is specified, otherwise an exception is raised.
setenv changes the binding of the environment variable *name*. If *value* is **#f** the binding is removed.

(current-time) procedure
(time->seconds time) procedure
(seconds->time x) procedure
(process-times) procedure
(cpu-time) procedure
(real-time) procedure

current-time returns a “time” object representing the current point in real time. **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). **time->seconds** converts the real number *x* representing the number of seconds elapsed since the “epoch” into a time object.

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.

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

real-time returns the real time that has elapsed since the program was started. 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). Time is computed internally using floating

point numbers which means that there is a gradual loss of resolution as time moves away from the “epoch”. Moreover, some operating systems report time in number of ticks using a 32 bit integer so the time returned by the above procedures may wraparound much before any significant loss of resolution occurs (for example 2.7 years if ticks are 1/50th of a second).

(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 occurred (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
```

7.3 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 possi-

ble 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.

`(current-directory [new-current-directory])` procedure
`(path-expand path)` procedure
`(path-normalize path)` procedure
`(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 manipulate file paths.

`current-directory` returns (or sets) the current working directory.

`path-expand` takes the path of a file or directory and returns an absolute path of the file or directory. Relative paths are relative to the current working directory. If the path is the empty string, the current working directory is returned.

`path-normalize` takes a path of a file or directory and returns a normalized path of the file or directory. A normalized path is a path containing no redundant parts and which is coherent with the filesystem. A normalized path of a directory will always end with a path separator (i.e. `'/'`, `'\'`, or `':'` depending on the operating system).

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. `path-volume` returns the file's volume (including the last path separator) or the empty string if no volume is specified in the path. `path-strip-volume` returns the path with the volume stripped off.

`(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!

<code>(s8vector? obj)</code>	procedure
<code>(make-s8vector k [fill])</code>	procedure
<code>(s8vector exact-int8...)</code>	procedure
<code>(s8vector-length s8vector)</code>	procedure
<code>(s8vector-ref s8vector k)</code>	procedure
<code>(s8vector-set! s8vector k exact-int8)</code>	procedure
<code>(s8vector->list s8vector)</code>	procedure
<code>(list->s8vector list-of-exact-int8)</code>	procedure
<code>(s8vector-fill! s8vector fill)</code>	procedure
<code>(s8vector-copy s8vector)</code>	procedure
<code>(subs8vector s8vector start end)</code>	procedure
<code>(u8vector? obj)</code>	procedure
<code>(make-u8vector k [fill])</code>	procedure
<code>(u8vector exact-int8...)</code>	procedure
<code>(u8vector-length u8vector)</code>	procedure
<code>(u8vector-ref u8vector k)</code>	procedure
<code>(u8vector-set! u8vector k exact-int8)</code>	procedure
<code>(u8vector->list u8vector)</code>	procedure
<code>(list->u8vector list-of-exact-int8)</code>	procedure
<code>(u8vector-fill! u8vector fill)</code>	procedure
<code>(u8vector-copy u8vector)</code>	procedure
<code>(subu8vector u8vector start end)</code>	procedure
<code>(s16vector? obj)</code>	procedure
<code>(make-s16vector k [fill])</code>	procedure
<code>(s16vector exact-int16...)</code>	procedure
<code>(s16vector-length s16vector)</code>	procedure
<code>(s16vector-ref s16vector k)</code>	procedure
<code>(s16vector-set! s16vector k exact-int16)</code>	procedure
<code>(s16vector->list s16vector)</code>	procedure
<code>(list->s16vector list-of-exact-int16)</code>	procedure
<code>(s16vector-fill! s16vector fill)</code>	procedure
<code>(s16vector-copy s16vector)</code>	procedure
<code>(subs16vector s16vector start end)</code>	procedure
<code>(u16vector? obj)</code>	procedure
<code>(make-u16vector k [fill])</code>	procedure
<code>(u16vector exact-int16...)</code>	procedure
<code>(u16vector-length u16vector)</code>	procedure
<code>(u16vector-ref u16vector k)</code>	procedure
<code>(u16vector-set! u16vector k exact-int16)</code>	procedure
<code>(u16vector->list u16vector)</code>	procedure
<code>(list->u16vector list-of-exact-int16)</code>	procedure
<code>(u16vector-fill! u16vector fill)</code>	procedure
<code>(u16vector-copy u16vector)</code>	procedure
<code>(subu16vector u16vector start end)</code>	procedure

<code>(s32vector? obj)</code>	procedure
<code>(make-s32vector k [fill])</code>	procedure
<code>(s32vector exact-int32...)</code>	procedure
<code>(s32vector-length s32vector)</code>	procedure
<code>(s32vector-ref s32vector k)</code>	procedure
<code>(s32vector-set! s32vector k exact-int32)</code>	procedure
<code>(s32vector->list s32vector)</code>	procedure
<code>(list->s32vector list-of-exact-int32)</code>	procedure
<code>(s32vector-fill! s32vector fill)</code>	procedure
<code>(s32vector-copy s32vector)</code>	procedure
<code>(subs32vector s32vector start end)</code>	procedure
<code>(u32vector? obj)</code>	procedure
<code>(make-u32vector k [fill])</code>	procedure
<code>(u32vector exact-int32...)</code>	procedure
<code>(u32vector-length u32vector)</code>	procedure
<code>(u32vector-ref u32vector k)</code>	procedure
<code>(u32vector-set! u32vector k exact-int32)</code>	procedure
<code>(u32vector->list u32vector)</code>	procedure
<code>(list->u32vector list-of-exact-int32)</code>	procedure
<code>(u32vector-fill! u32vector fill)</code>	procedure
<code>(u32vector-copy u32vector)</code>	procedure
<code>(subu32vector u32vector start end)</code>	procedure
<code>(s64vector? obj)</code>	procedure
<code>(make-s64vector k [fill])</code>	procedure
<code>(s64vector exact-int64...)</code>	procedure
<code>(s64vector-length s64vector)</code>	procedure
<code>(s64vector-ref s64vector k)</code>	procedure
<code>(s64vector-set! s64vector k exact-int64)</code>	procedure
<code>(s64vector->list s64vector)</code>	procedure
<code>(list->s64vector list-of-exact-int64)</code>	procedure
<code>(s64vector-fill! s64vector fill)</code>	procedure
<code>(s64vector-copy s64vector)</code>	procedure
<code>(subs64vector s64vector start end)</code>	procedure
<code>(u64vector? obj)</code>	procedure
<code>(make-u64vector k [fill])</code>	procedure
<code>(u64vector exact-int64...)</code>	procedure
<code>(u64vector-length u64vector)</code>	procedure
<code>(u64vector-ref u64vector k)</code>	procedure
<code>(u64vector-set! u64vector k exact-int64)</code>	procedure
<code>(u64vector->list u64vector)</code>	procedure
<code>(list->u64vector list-of-exact-int64)</code>	procedure
<code>(u64vector-fill! u64vector fill)</code>	procedure
<code>(u64vector-copy u64vector)</code>	procedure
<code>(subu64vector u64vector start end)</code>	procedure

<code>(f32vector? obj)</code>	procedure
<code>(make-f32vector k [fill])</code>	procedure
<code>(f32vector inexact-real...)</code>	procedure
<code>(f32vector-length f32vector)</code>	procedure
<code>(f32vector-ref f32vector k)</code>	procedure
<code>(f32vector-set! f32vector k inexact-real)</code>	procedure
<code>(f32vector->list f32vector)</code>	procedure
<code>(list->f32vector list-of-inexact-real)</code>	procedure
<code>(f32vector-fill! f32vector fill)</code>	procedure
<code>(f32vector-copy f32vector)</code>	procedure
<code>(subf32vector f32vector start end)</code>	procedure
<code>(f64vector? obj)</code>	procedure
<code>(make-f64vector k [fill])</code>	procedure
<code>(f64vector inexact-real...)</code>	procedure
<code>(f64vector-length f64vector)</code>	procedure
<code>(f64vector-ref f64vector k)</code>	procedure
<code>(f64vector-set! f64vector k inexact-real)</code>	procedure
<code>(f64vector->list f64vector)</code>	procedure
<code>(list->f64vector list-of-inexact-real)</code>	procedure
<code>(f64vector-fill! f64vector fill)</code>	procedure
<code>(f64vector-copy f64vector)</code>	procedure
<code>(subf64vector f64vector start end)</code>	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)
```

7.4 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.

8 Scheme threads

Gambit supports the execution of multiple threads, and conforms to SRFI 18 and SRFI 21. Please refer to the SRFI 18 and SRFI 21 documents for further details. *** The text of those SRFI's will be included here eventually!

9 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.

9.1 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:

<code>scheme-object</code>	denotes the universal type of Scheme objects (type <code>___SCMOBJ</code> defined in ‘ <code>gambit.h</code> ’)
<code>bool</code>	denotes the C++ ‘ <code>bool</code> ’ type or the C ‘ <code>int</code> ’ type (type <code>___BOOL</code> defined in ‘ <code>gambit.h</code> ’)
<code>int8</code>	8 bit signed integer (type <code>___S8</code> defined in ‘ <code>gambit.h</code> ’)
<code>unsigned-int8</code>	8 bit unsigned integer (type <code>___U8</code> defined in ‘ <code>gambit.h</code> ’)
<code>int16</code>	16 bit signed integer (type <code>___S16</code> defined in ‘ <code>gambit.h</code> ’)
<code>unsigned-int16</code>	16 bit unsigned integer (type <code>___U16</code> defined in ‘ <code>gambit.h</code> ’)
<code>int32</code>	32 bit signed integer (type <code>___S32</code> defined in ‘ <code>gambit.h</code> ’)
<code>unsigned-int32</code>	32 bit unsigned integer (type <code>___U32</code> defined in ‘ <code>gambit.h</code> ’)
<code>int64</code>	64 bit signed integer (type <code>___S64</code> defined in ‘ <code>gambit.h</code> ’)
<code>unsigned-int64</code>	64 bit unsigned integer (type <code>___U64</code> defined in ‘ <code>gambit.h</code> ’)
<code>float32</code>	32 bit floating point number (type <code>___F32</code> defined in ‘ <code>gambit.h</code> ’)
<code>float64</code>	64 bit floating point number (type <code>___F64</code> defined in ‘ <code>gambit.h</code> ’)
<code>latin1</code>	denotes LATIN-1 encoded characters (8 bit unsigned integer, type <code>___LATIN1</code> defined in ‘ <code>gambit.h</code> ’)
<code>ucs2</code>	denotes UCS-2 encoded characters (16 bit unsigned integer, type <code>___UCS2</code> defined in ‘ <code>gambit.h</code> ’)
<code>ucs4</code>	denotes UCS-4 encoded characters (32 bit unsigned integer, type <code>___UCS4</code> defined in ‘ <code>gambit.h</code> ’)

<code>char-string</code>	denotes the C <code>'char*'</code> type when used as a null terminated string
<code>nonnull-char-string</code>	denotes the nonnull C <code>'char*'</code> type when used as a null terminated string
<code>nonnull-char-string-list</code>	denotes an array of nonnull C <code>'char*'</code> terminated with a null pointer
<code>latin1-string</code>	denotes LATIN-1 encoded strings (null terminated string of 8 bit unsigned integers, i.e. <code>___LATIN1*</code>)
<code>nonnull-latin1-string</code>	denotes nonnull LATIN-1 encoded strings (null terminated string of 8 bit unsigned integers, i.e. <code>___LATIN1*</code>)
<code>nonnull-latin1-string-list</code>	denotes an array of nonnull LATIN-1 encoded strings terminated with a null pointer
<code>utf8-string</code>	denotes UTF-8 encoded strings (null terminated string of <code>char</code> , i.e. <code>char*</code>)
<code>nonnull-utf8-string</code>	denotes nonnull UTF-8 encoded strings (null terminated string of <code>char</code> , i.e. <code>char*</code>)
<code>nonnull-utf8-string-list</code>	denotes an array of nonnull UTF-8 encoded strings terminated with a null pointer
<code>ucs2-string</code>	denotes UCS-2 encoded strings (null terminated string of 16 bit unsigned integers, i.e. <code>___UCS2*</code>)
<code>nonnull-ucs2-string</code>	denotes nonnull UCS-2 encoded strings (null terminated string of 16 bit unsigned integers, i.e. <code>___UCS2*</code>)
<code>nonnull-ucs2-string-list</code>	denotes an array of nonnull UCS-2 encoded strings terminated with a null pointer
<code>ucs4-string</code>	denotes UCS-4 encoded strings (null terminated string of 32 bit unsigned integers, i.e. <code>___UCS4*</code>)
<code>nonnull-ucs4-string</code>	denotes nonnull UCS-4 encoded strings (null terminated string of 32 bit unsigned integers, i.e. <code>___UCS4*</code>)
<code>nonnull-ucs4-string-list</code>	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
<code>void</code>	<code>void</code>
<code>bool</code>	<code>bool</code>
<code>char</code>	<code>char</code> (may be signed or unsigned depending on the C compiler)
<code>signed-char</code>	<code>signed char</code>
<code>unsigned-char</code>	<code>unsigned char</code>
<code>latin1</code>	<code>latin1</code>
<code>ucs2</code>	<code>ucs2</code>
<code>ucs4</code>	<code>ucs4</code>
<code>short</code>	<code>short</code>
<code>unsigned-short</code>	<code>unsigned short</code>
<code>int</code>	<code>int</code>
<code>unsigned-int</code>	<code>unsigned int</code>
<code>long</code>	<code>long</code>
<code>unsigned-long</code>	<code>unsigned long</code>
<code>long-long</code>	<code>long long</code>
<code>unsigned-long-long</code>	<code>unsigned long long</code>
<code>float</code>	<code>float</code>
<code>double</code>	<code>double</code>
<code>int8</code>	<code>int8</code>
<code>unsigned-int8</code>	<code>unsigned-int8</code>
<code>int16</code>	<code>int16</code>
<code>unsigned-int16</code>	<code>unsigned-int16</code>
<code>int32</code>	<code>int32</code>
<code>unsigned-int32</code>	<code>unsigned-int32</code>
<code>int64</code>	<code>int64</code>
<code>unsigned-int64</code>	<code>unsigned-int64</code>
<code>float32</code>	<code>float32</code>
<code>float64</code>	<code>float64</code>
<code>(struct "c-struct-id" [tag [release-function]])</code>	<code>struct c-struct-id</code> (where <i>c-struct-id</i> is the name of a C structure; see below for the meaning of <i>tag</i> and <i>release-function</i>)

`(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

<code>[unsigned-] int8/int16/int32/int64;</code>	<code>[unsigned] short/int/long</code>
	exact integer
<code>float; double; float32; float64</code>	
	inexact real
<code>char-string; latin1-string; utf8-string; ucs2-string; ucs4-string</code>	
	string or <code>#f</code> if it is equal to 'NULL'
<code>nonnull-char-string; nonnull-latin1-string; nonnull-utf8-string;</code>	
<code>nonnull-ucs2-string; nonnull-ucs4-string</code>	
	string
<code>struct/union/type/pointer/nonnull-pointer</code>	
	foreign object with the appropriate tag or <code>#f</code> in the case of a pointer equal to 'NULL'
<code>function</code>	procedure or <code>#f</code> if it is equal to 'NULL'
<code>nonnull-function</code>	
	procedure
<code>void</code>	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.

9.2 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-declares`, `c-initializes`, and `c-lambdas`, and `c-defines` 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;
")

```

9.3 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 occurrence 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 ();")
```

9.4 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 life-time 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\\n\" ); printf( \"world\\n\\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;
")
```

9.5 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’`.

9.6 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 Section 9.7 [continuations], page 78). 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_ERR)) {
#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_ERR)) {
#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))

```

9.7 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.

10 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.

11 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 were combined with inlined allocators (e.g. `'list'`, `'vector'`, `'lambda'`).
- Previously nested C to Scheme calls were prohibited. They are now allowed.
- A memory leak occurring 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`.

12 Copyright and distribution information

The Gambit system (including the Gambit-C version) is Copyright © 1994-2002 by Marc Feeley, all rights reserved.

The Gambit system and programs developed with it may be used and distributed only under the following conditions: they must include this copyright and distribution notice and they must not be sold, transferred or used to provide a tool or service (such as a web server) for which there is a direct or indirect revenue. In other words if the software developed with Gambit-C has commercial value a commercial license is required. The system may be used at a school or university for teaching and research even if tuition is charged by the school. For a commercial license please contact `gambit@iro.umontreal.ca`.

General Index

#

##add-gc-interrupt-job..... 52
 ##add-timer-interrupt-job..... 52
 ##clear-gc-interrupt-jobs..... 52
 ##clear-timer-interrupt-jobs..... 52
 ##gc..... 52

,

, (c *expr*) 3
 , - 4
 , ? 3
 , + 4
 , b 4
 , c 3
 , d 3
 , e 5
 , i 4
 , l 4
 , n 4
 , q 3
 , s 3
 , t 3
 , y 4

-

-: = 19
 -: + 19
 -: c 19
 -: d 18
 -: d- 19
 -: da 18
 -: dc 19
 -: di 19
 -: dlevel 19
 -: dp 18
 -: dq 19
 -: dr 18
 -: ds 19
 -: h 18
 -: l 18
 -: m 18
 -: s 18, 41
 -: S 18
 -: t 19
 -c 10, 12
 -call_shared 17
 -cc-options 11

-D__DYNAMIC 14
 -D__LIBRARY 16
 -D__PRIMAL 16
 -D__SHARED 16
 -D__SINGLE_HOST 17
 -debug 12
 -dynamic 10, 12
 -e 11
 -expansion 12
 -flat 12
 -fpic 17
 -fPIC 17
 -gvm 12
 -i 11
 -I/usr/local/Gambit-C/include 17
 -Kpic 17
 -KPIC 17
 -l base 12
 -L/usr/local/Gambit-C/lib 17
 -ld-options 12
 -O 17
 -o *output* 12
 -pic 17
 -postlude 11
 -prelude 11
 -rdynamic 17
 -report 12
 -shared 17
 -track-scheme 12
 -verbose 12
 -warnings 12

•

.c 10
 .scm 10
 .six 10

==

= 29

-

___cleanup 69
 ___setup 69

+

+z 17

- >
- >..... 29
- >=..... 29
- ~
- ^C..... 2, 79
- ^D..... 3
- ^Z..... 3, 79
- <
- <..... 29
- <=..... 29
- A**
- absolute path..... 20, 53
- all-bits-set?..... 46
- and..... 80
- any-bits-set?..... 46
- apply..... 80
- arithmetic-shift..... 46
- B**
- bit-count..... 46
- bit-set?..... 46
- bitwise-and..... 46
- bitwise-ior..... 46
- bitwise-merge..... 46
- bitwise-not..... 46
- bitwise-xor..... 46
- block..... 30
- box..... 46
- box?..... 46
- break..... 37
- bugs fixed..... 80
- bytevectors..... 23, 56
- C**
- c-declare..... 34, 66
- c-define..... 34, 69
- c-define-type..... 34, 70
- c-initialize..... 34, 67
- c-lambda..... 34, 67
- call-with-input-file..... 23
- call-with-input-string..... 40, 43
- call-with-input-u8vector..... 43
- call-with-input-vector..... 43
- call-with-output-file..... 23
- call-with-output-string..... 40, 43
- call-with-output-u8vector..... 43
- call-with-output-vector..... 43
- case-conversion?-set!..... 41
- ceiling..... 80
- char-ci=?..... 29
- char-ci>=?..... 29
- char-ci>?..... 29
- char-ci<=?..... 29
- char-ci<?..... 29
- char=?..... 29
- char>=?..... 29
- char>?..... 29
- char<=?..... 29
- char<?..... 29
- clear-bit-field..... 46
- close-port..... 42
- command-line..... 7, 51
- compile-file..... 48
- compile-file-to-c..... 48
- compiler..... 10
- compiler options..... 10
- cond-expand..... 53
- continuations..... 78, 80
- copy-bit-field..... 46
- copy-file..... 43
- cpu-time..... 51
- create-directory..... 43
- create-fifo..... 43
- create-link..... 43
- create-symbolic-link..... 43
- current working directory..... 20, 53
- current-console-port..... 43
- current-directory..... 44, 53
- current-error-port..... 44
- current-exception-handler..... 44
- current-input-port..... 44
- current-output-port..... 44
- current-readtable..... 41, 44
- current-time..... 51
- current-user-interrupt-handler..... 44
- D**
- debugging..... 35, 36, 38, 39
- declare..... 30
- deficiencies..... 79
- define..... 32, 79
- define-macro..... 30
- define-structure..... 34, 79
- delete-directory..... 43
- delete-file..... 43
- directory-files..... 43

display 23, 80
 display-environment-set! 39

E

Emacs 21
 equal? 80
 error 50
 eval 48
 exact->inexact 80
 exit 51
 extended-bindings 31
 extensions, Scheme 23
 extract-bit-field 46

F

f32vector 56
 f32vector->list 56
 f32vector-copy 56
 f32vector-fill! 56
 f32vector-length 56
 f32vector-ref 56
 f32vector-set! 56
 f32vector? 55
 f64vector 56
 f64vector->list 56
 f64vector-copy 56
 f64vector-fill! 56
 f64vector-length 56
 f64vector-ref 56
 f64vector-set! 56
 f64vector? 56
 FFI 59
 file names 20
 file-attributes 44
 file-creation-time 44
 file-device 44
 file-exists? 39
 file-group 44
 file-info 44
 file-info-attributes 44
 file-info-creation-time 44
 file-info-device 44
 file-info-group 44
 file-info-inode 44
 file-info-last-access-time 44
 file-info-last-change-time 44
 file-info-last-modification-time 44
 file-info-mode 44
 file-info-number-of-links 44
 file-info-owner 44

file-info-size 44
 file-info-type 44
 file-info? 44
 file-inode 44
 file-last-access-time 44
 file-last-change-time 44
 file-last-modification-time 44
 file-mode 44
 file-number-of-links 44
 file-owner 44
 file-size 44
 file-type 44
 file.c 10
 file.scm 10
 file.six 10
 first-set-bit 46
 fixnum 32
 floating point 80
 floating point overflow 79
 flonum 32
 floor 80
 force-output 39
 foreign function interface 59

G

Gambit 1, 23
 Gambit installation directory 20
 Gambit-C 1
 gambit.el 21
 GC 42, 52, 80
 gc-report-set! 42
 generic 32
 gensym 47
 get-output-string 39, 43
 get-output-u8vector 43
 get-output-vector 43
 getenv 51
 group-info 44
 group-info-gid 44
 group-info-members 44
 group-info-name 44
 group-info? 44
 gsc 1, 10, 18, 48, 49
 gsi 1, 2, 18

H

home directory 20
 host-info 45
 host-info-addresses 45
 host-info-aliases 45

host-info-name 45
 host-info? 45

I

ieee-scheme 30
 include 30
 inline 31
 inlining-limit 31
 input-port-column 42
 input-port-line 42
 input-port-timeout-set! 42
 insert-bit-field 46
 integer-length 46
 interpreter 2, 10
 interrupts-enabled 31

K

keyword->string 41
 keyword? 41
 keywords 23, 41
 keywords-allowed?-set! 41

L

lambda 32
 lambda-lift 31
 last_.c 12
 limitations 79
 link-flat 49
 link-incremental 49
 list->f32vector 56
 list->f64vector 56
 list->s16vector 54
 list->s32vector 55
 list->s64vector 55
 list->s8vector 54
 list->u16vector 54
 list->u32vector 55
 list->u64vector 55
 list->u8vector 54
 load 48

M

make-f32vector 56
 make-f64vector 56
 make-parameter 53
 make-s16vector 54
 make-s32vector 55
 make-s64vector 55
 make-s8vector 54
 make-u16vector 54

make-u32vector 55
 make-u64vector 55
 make-u8vector 54
 make-will 46

N

number->string 80

O

object file 48
 open-directory 45
 open-file 23, 45
 open-input-file 23, 45
 open-input-string 39, 43, 45
 open-input-u8vector 43, 45
 open-input-vector 43, 45
 open-output-file 23, 45
 open-output-string 39, 43, 45
 open-output-u8vector 43, 45
 open-output-vector 43, 45
 open-process 45
 open-string 43, 45
 open-tcp-client 45
 open-tcp-server 45
 open-u8vector 43, 45
 open-vector 43, 45
 optimizer 80
 options, compiler 10
 options, runtime 18
 or 80
 output-port-column 42
 output-port-line 42
 output-port-timeout-set! 42
 output-port-width 42
 overflow, floating point 79

P

parameterize 53
 path-directory 53
 path-expand 53
 path-extension 53
 path-normalize 53
 path-strip-directory 53
 path-strip-extension 53
 path-strip-volume 53
 path-volume 53
 port? 42
 pp 39
 pretty-print 39
 problems 79

procedure call	80
procedures	80
process-times	51
proper-tail-calls-set!	38

R

r4rs-scheme	30
read	23, 41, 80
read-all	42
read-byte	42
read-line	42
read-subu8vector	42
readtable	23
readtable-case-conversion?-set!	45
readtable-copy	45
readtable-eval-allowed?-set!	45
readtable-keywords-allowed?-set!	45
readtable-max-write-length-set!	45
readtable-max-write-level-set!	45
readtable-sharing-allowed?-set!	45
readtable-sharp-quote-keyword-set!	45
readtable-start-syntax-set!	45
readtable?	40, 45
real-time	51
relative path	20, 53
rename-file	43
rest parameter	80
round	80
rounding	80
runtime options	18

S

s16vector	54
s16vector->list	54
s16vector-copy	54
s16vector-fill!	54
s16vector-length	54
s16vector-ref	54
s16vector-set!	54
s16vector?	54
s32vector	55
s32vector->list	55
s32vector-copy	55
s32vector-fill!	55
s32vector-length	55
s32vector-ref	55
s32vector-set!	55
s32vector?	54
s64vector	55
s64vector->list	55

s64vector-copy	55
s64vector-fill!	55
s64vector-length	55
s64vector-ref	55
s64vector-set!	55
s64vector?	55
s8vector	54
s8vector->list	54
s8vector-copy	54
s8vector-fill!	54
s8vector-length	54
s8vector-ref	54
s8vector-set!	54
s8vector?	54
safe	31
Scheme	1, 23
scheme-ieee-1178-1990	7
scheme-r4rs	7
scheme-r5rs	7
scheme-six	7
scheme-srfi-0	7
seconds->time	51
separate	30
serial-number->object	45
set!	79
set-box!	46
setenv	51
shell-command	53
SIX	23
standard-bindings	31
step	36
step-level-set!	36
string ports	39, 40, 80
string->keyword	41
string->number	80
string-ci=?	29
string-ci>=?	29
string-ci>?	29
string-ci<=?	29
string-ci<?	29
string=?	29
string>=?	29
string>?	29
string<=?	29
string<?	29
subf32vector	56
subf64vector	56
subs16vector	54
subs32vector	55
subs64vector	55

subs8vector	54
subu16vector	54
subu32vector	55
subu64vector	55
subu8vector	54

T

test-bit-field?	46
threads	58
time	52
time->seconds	51
trace	35
transcript-off	23
transcript-on	23
tty-history	43
tty-history-set!	43
tty-max-history-length-set!	43
tty-mode-set!	43
tty-paren-balance-duration-set!	43
tty-text-attributes-set!	43
tty-type-set!	43
tty?	42

U

u16vector	54
u16vector->list	54
u16vector-copy	54
u16vector-fill!	54
u16vector-length	54
u16vector-ref	54
u16vector-set!	54
u16vector?	54
u32vector	55
u32vector->list	55
u32vector-copy	55
u32vector-fill!	55
u32vector-length	55
u32vector-ref	55
u32vector-set!	55
u32vector?	55
u64vector	55
u64vector->list	55
u64vector-copy	55
u64vector-fill!	55
u64vector-length	55

u64vector-ref	55
u64vector-set!	55
u64vector?	55
u8vector	54
u8vector->list	54
u8vector-copy	54
u8vector-fill!	54
u8vector-length	54
u8vector-ref	54
u8vector-set!	54
u8vector?	54
unbox	46
unbreak	38
untrace	35
user-info	44
user-info-gid	45
user-info-home	45
user-info-name	45
user-info-shell	45
user-info-uid	45
user-info?	45

V

void	48
------------	----

W

will-execute!	46
will-testator	46
will?	46
with-exception-catcher	45
with-exception-handler	45
with-input-from-file	23
with-input-from-port	40
with-input-from-string	40, 43
with-input-from-u8vector	43
with-input-from-vector	43
with-output-to-file	23
with-output-to-port	40
with-output-to-string	40, 43
with-output-to-u8vector	43
with-output-to-vector	43
write	23, 80
write-byte	42
write-subu8vector	42

Table of Contents

1	Gambit-C: a portable version of Gambit.....	1
1.1	Accessing the Gambit system files	1
2	The Gambit Scheme interpreter	2
2.1	Interactive mode	2
2.2	Batch mode.....	6
2.3	Customization	6
2.4	Process exit status	7
2.5	Scheme scripts	7
3	The Gambit Scheme compiler	10
3.1	Interactive mode	10
3.2	Customization	10
3.3	Batch mode.....	10
3.4	Link files	13
3.4.1	Building an executable program	13
3.4.2	Building a loadable library	14
3.4.3	Building a shared-library	16
3.4.4	Other compilation options and flags	17
4	Runtime options for all programs	18
5	Handling of file names	20
6	Emacs interface.....	21
7	Extensions to Scheme	23
7.1	Standard special forms and procedures	23
7.2	Additional special forms and procedures	29
7.3	Unstable additions	52
7.4	Other extensions	57
8	Scheme threads.....	58

9	Interface to C	59
9.1	The mapping of types between C and Scheme	59
9.2	The <code>c-declare</code> special form	66
9.3	The <code>c-initialize</code> special form	67
9.4	The <code>c-lambda</code> special form	67
9.5	The <code>c-define</code> special form	69
9.6	The <code>c-define-type</code> special form	70
9.7	Continuations and the C-interface	78
10	Known limitations and deficiencies	79
11	Bugs fixed	80
12	Copyright and distribution information ...	81
	General Index	82