Copyright © 1994-2009 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 The Gambit-C system

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

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

For the most up to date information on Gambit and add-on packages please check the Gambit web page at http://gambit.iro.umontreal.ca. The web page has links to the Gambit mailing list, the bug reporting system, and the source code repository.

1.1 Accessing the system files

Files related to Gambit, such as executables, libraries and header files, are stored in multiple Gambit installation directories. Gambit may be installed on a system according to two different installation models.

In the first model there is a single directory where all the Gambit installation directories are stored. This central installation directory is typically /usr/local/Gambit-C under UNIX, /Library/Gambit-C under Mac OS X and C:/Program Files/Gambit-C under Microsoft Windows. This may have been overridden when the system was built with the command ‘configure --prefix=/my/Gambit-C’. If the system was built with the command ‘configure --enable-multiple-versions’ then the central installation directory is prefix/version, where version is the system version string (e.g. v4.6.1 for Gambit v4.6.1). Moreover, prefix/current will be a symbolic link which points to the central installation directory. In this model, the Gambit installation directory named X is simply the subdirectory X of the central installation directory.

In the second model some or all of the Gambit installation directories are stored in installation specific directories. The location of these directories is assigned when the system is built using the command ‘configure --bindir=/my/bin --includedir=/my/include --libdir=/my/lib’.

The advantage of the first model is that it is easy to have multiple versions of Gambit coexist and to remove all the files of a given version. However, the second model may be necessary to conform to the package installation conventions of some operating systems.

Executable programs such as the interpreter gsi and compiler gsc can be found in the bin installation directory. Adding this directory to the PATH environment variable allows these programs to be started by simply entering their name. This is done automatically by the Mac OS X and Microsoft Windows installers.

The runtime library is located in the lib installation directory. 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
Chapter 1: The Gambit-C system

of UNIX, LIBPATH on AIX, SHLIB_PATH on HPX, DYLD_LIBRARY_PATH on Mac OS X, and PATH on Microsoft Windows. If the shell is sh, 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 installation directory. This header file is needed for compiling Scheme programs with the Gambit-C compiler. When the C compiler is being called explicitly 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
$ ln -s /usr/local/Gambit-C/include/gambit.h /usr/include

Alternatively these files can be copied or linked in the directory where the C compiler is invoked (this requires no special privileges).

Another approach is to set some environment variables which are used to tell the C compiler where to find header files and libraries. For example, the following settings can be used for the gcc C compiler:

$ export LIBRARY_PATH=/usr/local/Gambit-C/lib
$ export CPATH=/usr/local/Gambit-C/include

Note that this may have been done by the installation process. In particular, the Mac OS X and Microsoft Windows prebuilt installers set up the environment so that the gcc compiler finds these files automatically.
2 The Gambit Scheme interpreter

Synopsis:

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

The interpreter is executed in interactive mode when no file or `-i` or `-e` option is given on the command line. Otherwise the interpreter is executed in batch mode. The `-i` option is ignored by the interpreter. The initialization file will be examined unless the `-f` option is present (see Section 2.3 [GSI customization], page 4). The `-v` option prints the system version string, system time stamp, operating system type, and configure script options on standard output and exits. Runtime options are explained in Chapter 4 [Runtime options], page 23.

2.1 Interactive mode

In interactive mode a read-eval-print loop (REPL) is started for the user to interact with the interpreter. At each iteration of this loop the interpreter displays a prompt, reads a command and executes it. The commands can be expressions to evaluate (the typical case) or special commands related to debugging, for example `;q` to terminate the process (for a complete list of commands see Chapter 5 [Debugging], page 27). Most commands produce some output, such as the value or error message resulting from an evaluation.

The input and output of the interaction is done on the interaction channel. The interaction channel can be specified through the runtime options but if none is specified the system uses a reasonable default that depends on the system’s configuration. When the system’s runtime library was built with support for GUIDE, the Gambit Universal IDE (with the command `configure --enable-guide`) the interaction channel corresponds to the console window of the primordial thread (for details see Section 5.7 [GUIDE], page 41), otherwise the interaction channel is the user’s console, also known as the controlling terminal in the UNIX world. When the REPL starts, the ports associated with `(current-input-port)`, `(current-output-port)` and `(current-error-port)` all refer to the interaction channel.

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

Here is a sample interaction with gsi:

\[
\begin{align*}
\$ & \texttt{gsi} \\
& \text{Gambit v4.6.1} \\
> & \texttt{(define (fact n) (if (< n 2) 1 (* n (fact (- n 1)))))} \\
> & \texttt{(map fact '(1 2 3 4 5 6))} \\
& (1 2 6 24 120 720) \\
> & \texttt{(values (fact 10) (fact 40))} \\
& 3628800 \\
& 815915283247897734345611269596115894272000000000000 \\
> & ;q
\end{align*}
\]
What happens when errors occur is explained in Chapter 5 [Debugging], page 27.

### 2.2 Batch mode

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

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 will search for both a source file and an object file. The object file’s name is obtained by adding to the file name a ‘.on’ extension with the highest consecutive version number starting with 1. The source file’s name is obtained by adding to the file name the file extensions ‘.scm’ and ‘.six’ (the first found is the source file). If both a source file and an object file exist, then the one with the latest modification time is loaded. Otherwise the file that is found is loaded. When the file name has an extension, the `load` procedure will only attempt to load the file with that specific name.

When the extension of the file loaded is ‘.scm’ the content of the file will be parsed using the normal Scheme prefix syntax. When the extension of the file loaded is ‘.six’ the content of the file will be parsed using the Scheme infix syntax extension (see Section 18.12 [Scheme infix syntax extension], page 191). Otherwise, `gsi` will parse the file using the normal Scheme prefix syntax.

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

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

```
$ cat h.scm
(display "hello") (newline)
$ cat w.six
display("world"); newline();
$ gsi h.scm - w.six -e "(pretty-print 1)(pretty-print 2)"
hello
> (define (display x) (write (reverse (string->list x))))
> ,(c 0)
  (\d \l \r \o \w)
  1
  2
```

### 2.3 Customization

There are two ways to customize the interpreter. When the interpreter starts off it tries to execute a ‘(load "˜lib/gambcext")’ (for an explanation of how file names are
Chapter 2: The Gambit Scheme interpreter

interpreted see Chapter 16 [Host environment], page 134). An error is not signaled when the file does not exist. Interpreter extensions and patches that are meant to apply to all users and all modes should go in that file.

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

2.4 Process exit status

The status is zero when the interpreter exits normally and is nonzero when the interpreter exits due to an error. Here is the meaning of the exit statuses:

0  The execution of the primordial thread (i.e. the main thread) did not encounter any error. It is however possible that other threads terminated abnormally (by default threads other than the primordial thread terminate silently when they raise an exception that is not handled).
64  The runtime options or the environment variable ‘GAMBCOPT’ contained a syntax error or were invalid.
70  This normally indicates that an exception was raised in the primordial thread and the exception was not handled.
71  There was a problem initializing the runtime system, for example insufficient memory to allocate critical tables.

For example, if the shell is sh:

```
$ gsi -:d0 -e "(pretty-print (expt 2 100))"
1267650600228229401496703205376
$ echo $?
0
$ gsi -:d0,unknown # try to use an unknown runtime option
$ echo $?
64
$ gsi -:d0 nonexistent.scm # try to load a file that does not exist
$ echo $?
70
$ gsi nonexistent.scm
*** ERROR IN ##main -- No such file or directory
(load "nonexistent.scm")
$ echo $?
70
$ gsi -:m4000000 # ask for a 4 gigabyte heap
*** malloc: vm_allocate(size=528384) failed (error code=3)
*** malloc[15068]: error: Can’t allocate region
$ echo $?
```
Note the use of the runtime option ‘-:d0’ that prevents error messages from being output, and the runtime option ‘-:m4000000’ which sets the minimum heap size to 4 gigabytes.

### 2.5 Scheme scripts

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

- **scheme-r4rs**: R4RS language with prefix syntax, case-insensitivity, keyword syntax not supported
- **scheme-r5rs**: R5RS language with prefix syntax, case-insensitivity, keyword syntax not supported
- **scheme-ieee-1178-1990**: IEEE 1178-1990 language with prefix syntax, case-insensitivity, keyword syntax not supported
- **scheme-srfi-0**: R5RS language with prefix syntax and SRFI 0 support (i.e. `cond`-`expand` special form), case-insensitivity, keyword syntax not supported
- **gsi-script**: Full Gambit Scheme language with prefix syntax, case-sensitivity, keyword syntax supported
- **gsc-script**: Full Gambit Scheme language with prefix syntax, case-sensitivity, keyword syntax supported
- **six-script**: Full Gambit Scheme language with infix syntax, case-sensitivity, keyword syntax supported

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

After processing the script line, `load` will parse the rest of the file (using the syntax of the language indicated) and then execute it. When the file is being loaded because it is an argument on the interpreter’s command line, the interpreter will:

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

- When main returns, the interpreter exits. The command-line arguments after a script file are consequently not processed (however they do appear in the list returned by the command-line procedure, after the script file’s expanded file name, so it is up to the script to process them).

2.5.1 Scripts under UNIX and Mac OS X

Under UNIX and Mac OS X, the Gambit-C installation process creates the executable ‘gsi’ and also the executables ‘six’, ‘gsi-script’, ‘six-script’, ‘scheme-r5rs’, ‘scheme-srfi-0’, etc as links to ‘gsi’. A Scheme script need only start with the name of the desired Scheme language variant prefixed with ‘#!’ and the directory where the Gambit-C executables are stored. This script should be made executable by setting the execute permission bits (with a ‘chmod +x script’). Here is an example of a script which lists on standard output the files in the current directory:

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

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

```scheme
#!/usr/local/Gambit-C/bin/six-script
void main (obj n_str)
{
int n = \string->number(n_str);
for (int i=1; i<=n; i++)
\pretty-print(i);
}
```

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

```scheme
#!/usr/bin/env gsi-script
(define (display-file filename)
(display (call-with-input-file filename
(lambda (port)
(read-line port #f))))))
(for-each display-file (cdr (command-line)))
```

2.5.2 Scripts under Microsoft Windows

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

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

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

### 2.5.3 Compiling scripts

A script file can be compiled using the Gambit Scheme compiler (see Chapter 3 [GSC],
page 9) into a standalone executable. The script line will provide information to the compiler
on which language to use. The script line also provides information on which runtime options
to use when executing the compiled script. This is useful to set the default runtime options
of an executable program.

The compiled script will be executed similarly to an interpreted script (i.e. the list of
command line arguments returned by the command-line procedure and the invocation of
the main procedure).

For example:

```
$ cat square.scm
#!/usr/local/Gambit-C/bin/gsi-script -:d0
(define (main arg)
  (pretty-print (expt (string->number arg) 2)))
$ gsi square 30
   # gsi will load square.scm
900
$ gsc -exe square
   # compile the script to a standalone program
$ ./square 30
900
$ ./square 1 2 3
   # too many arguments to main
$ echo $?
70
$ ./square -:d1 1 2 3
   # ask for error message
*** ERROR -- Wrong number of arguments passed to procedure
(main "1" "2" "3")
```
3 The Gambit Scheme compiler

Synopsis:

```
gsc [-:runtimeoption, ...] [-i] [-f] [-v]
    [-prelude expressions] [-postlude expressions]
    [-dynamic] [-exe] [-obj] [-cc-options options]
    [-ld-options-prelude options] [-ld-options options]
    [-debug] [-debug-location] [-debug-source]
    [-debug-environments] [-track-scheme]
    [[-] [-e expressions] [file]]...
```

3.1 Interactive mode

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

3.2 Customization

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 (with either no extension, or a ‘.c’ extension, or some other extension) on the command line and compiles each Scheme file into a C file. The extension can be omitted from file when the Scheme file has a ‘.scm’ or ‘.six’ extension. When the extension of the Scheme file is ‘.six’ the content of the file will be parsed using the Scheme infix syntax extension (see Section 18.12 [Scheme infix syntax extension], page 191). Otherwise, gsc will parse the Scheme file using the normal Scheme prefix syntax. Files with a ‘.c’ extension must have been previously produced by gsc, with the ‘-c’ option, and are used by Gambit’s linker.

For each Scheme file a C file ‘file.c’ will be produced. The C file’s name is the same as the Scheme file, but the extension is changed to ‘.c’. By default the C file is created in the same directory as the Scheme file. This default can be overridden with the compiler’s ‘-o’ option.

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 only invoked when the ‘-link’ or ‘-exe’ 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 23). If present, the ‘-i’, ‘-f’, and ‘-v’ compiler options must come first. The available options are:
-i                Force interpreter mode.
-f                Do not examine the initialization file.
-v                Print the system version string, system time stamp, operating system
type, and configure script options on standard output and exit.

-prelude expressions
                Add expressions to the top of the source code being compiled.
-postlude expressions
                Add expressions to the bottom of the source code being compiled.
-cc-options options
                Add options to the command that invokes the C compiler.
-ld-options-prelude options
                Add options to the command that invokes the C linker.
-ld-options options
                Add options to the command that invokes the C linker.
-warnings        Display warnings.
-verbose          Display a trace of the compiler's activity.
-report           Display a global variable usage report.
-expansion        Display the source code after expansion.
-gvm              Generate a listing of the GVM code.
-debug            Include all debugging information in the code generated.
-debug-location   Include source code location debugging information in the code generated.
-debug-source     Include the source code debugging information in the code generated.
-debug-environments Include environment debugging information in the code generated.
-track-scheme     Generate ‘#line’ directives referring back to the Scheme code.
-o output         Set name of output file or directory where output file(s) are written.
-dynamic          Compile Scheme source files to dynamically loadable object files (this is
                the default).
-exe               Compile Scheme source files into an executable program.
-obj               Compile Scheme source files to object files.
-keep-c            Keep any intermediate ‘.c’ files that are generated.
-c                 Compile Scheme source files to C without generating link file.
-link              Compile Scheme source files to C and generate a link file.
-flat              Generate a flat link file instead of the default incremental link file.
-l base  Specify the link file of the base library to use for the link.
-e expressions  Evaluate expressions in the interaction environment.

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

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 a dynamically loadable object file is being generated (neither the ‘-c’ or ‘-link’ options are 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 -cc-options "-U___SINGLE_HOST -O2 -I../include" bench.scm

The ‘-ld-options-prelude’ and ‘-ld-options’ options are only meaningful when a dynamically loadable object file is being generated (neither the ‘-c’ or ‘-link’ options are used). The ‘-ld-options-prelude’ and ‘-ld-options’ options add the specified options to the command that invokes the C linker (the options in ld-options-prelude are passed to the C linker before the input file and the options in ld-options are passed after). 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 -ld-options "-L/usr/X11R6/lib -lX11 -dynamic" 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 whether 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 all kinds of debugging information to be saved in the code generated. See the documentation of the ‘debug’ declaration for details.
The ‘-debug-location’ option causes source code location debugging information to be saved in the code generated. See the documentation of the ‘debug-location’ declaration for details.

The ‘-debug-source’ option causes source code debugging information to be saved in the code generated. See the documentation of the ‘debug-source’ declaration for details.

The ‘-debug-environments’ option causes environment debugging information to be saved in the code generated. See the documentation of the ‘debug-environments’ declaration for details.

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 or profiler to debug Scheme code.

The ‘-o’ option sets the filename of the output file, or the directory in which the output file(s) generated by the compiler are written.

If the ‘-link’ or ‘-exe’ options 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. By default the link file is ‘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 ‘-exe’ option is specified, the generated C files and link file are compiled and linked using the C compiler to produce an executable program whose name defaults to ‘last.exe’. When the ‘-obj’ option is specified, the generated C files are compiled using the C compiler to produce object files (‘.o’ or ‘.obj’ extensions). If neither the ‘-link’, ‘-c’, ‘-exe’, ‘-obj’ options appear on the command line, the Scheme source files are compiled to dynamically loadable object files (‘.on’ extension). The ‘-keep-c’ option will prevent the deletion of any intermediate ‘.c’ file that is generated. Note that in this case the intermediate ‘.c’ file will be generated in the same directory as the Scheme source file even if the ‘-o’ option is used.

The ‘-flat’ option is only meaningful when a link file is being generated (i.e. the ‘-link’ or ‘-exe’ options also appear on the command line). 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 when an incremental link file is being generated (i.e. the ‘-link’ or ‘-exe’ options appear on the command line and the ‘-flat’ option is 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’).

The ‘-e’ option starts a REPL interaction.

The ‘-e’ option evaluates the specified expressions in the interaction environment.

### 3.4 Link files

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

In general, a program 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 program 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 though 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 a program 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 program 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 programs (this linking information typically takes hundreds of kilobytes).

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, a program 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 program is dynamically linked with this shared-library and only contains the user modules and the incremental link file. For small programs this approach greatly reduces the size of the program 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 invoke `gsc` with the ‘-exe’ option. The compiler will transparently perform all the steps necessary, including compiling Scheme source files to C files, generating the link file, compiling the C files generated to object files, and creating the final executable file using the C linker. The following example shows how to build the executable program ‘hello.exe’ which contains the two Scheme modules ‘h.scm’ and ‘w.six’.

```
$ cat h.scm
(display "hello") (newline)
$ cat w.six
(display "world"); newline();
$ gsc -o hello.exe -exe h.scm w.six
```

h.scm:
/Users/feeley/gambit/doc/h.c:
w.six:
/Users/feeley/gambit/doc/w.c:
The detailed steps which are performed can be viewed by setting the ‘GAMBC_CC_VERBOSE’ environment variable to a nonnull value. For example:

```
$ export GAMBC_CC_VERBOSE=yes
$ gsc -o hello.exe -exe h.scm w.six
```

Using a single invocation of gsc with the ‘-exe’ option is sometimes inappropriate when the build process is more complex, for example when the program is composed of several separately compiled modules. In such a case it is useful to decompose the build process into smaller compilation steps. The ‘hello.exe’ executable program could have been built by separating the generation of C files from the C compilation and linking:

```
$ gsc -c h.scm
$ gsc -c w.six
$ gsc -o hello.exe -exe h.c w.c
```

When even finer control is desired the build process can be decomposed into smaller steps that invoke the C compiler and linker explicitly. This is described in the rest of this section.

The gsc compiler can be invoked to compile each Scheme module into a C file and to 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. The content of the three source files (‘m1.c’, ‘m2.scm’ and ‘m3.scm’) is:

```
/* File: "m1.c" */
int power_of_2 (int x) { return 1<<x; }

; File: "m2.scm"
(c-declare "extern int power_of_2 ();")
(define pow2 (c-lambda (int) int "power_of_2")(define (twice x) (cons x x))

; File: "m3.scm"
```
The compilation of the two Scheme source files can be done with three invocations of gsc:

\$ gsc -c m2.scm  # create m2.c (note: .scm is optional)
\$ gsc -c m3.scm  # create m3.c (note: .scm is optional)
\$ gsc -link m2.c m3.c  # create the incremental link file m3_.c

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

\$ gsc -link m2 m3
m2:
m3:

At this point there will be 4 C files: ‘m1.c’, ‘m2.c’, ‘m3.c’, and ‘m3_.c’. To produce an executable program these files must be compiled with a C compiler and linked with the Gambit-C runtime library. The C compiler options needed will depend on the C compiler and the operating system (in particular it may be necessary to add the options ‘-I/usr/local/Gambit-C/include -L/usr/local/Gambit-C/lib’ to access the ‘gambit.h’ header file and the Gambit-C runtime library).

Here is an example under Mac OS X:

\$ uname -srmp
Darwin 8.1.0 Power Macintosh powerpc
\$ gsc -obj m1.c m2.c m3.c m3_.c
m1.c:
m2.c:
m3.c:
m3_.c:
\$ gcc m1.o m2.o m3.o m3_.o -lgambc
\$ ./a.out
((2 . 2) (4 . 4) (8 . 8) (16 . 16))

Here is an example under Linux:

\$ uname -srmp
Linux 2.6.8-1.521 i686 athlon
\$ gsc -obj m1.c m2.c m3.c m3_.c
m1.c:
m2.c:
m3.c:
m3_.c:
\$ gcc m1.o m2.o m3.o m3_.o -lgambc -lm -ldl -lutil
\$ ./a.out
((2 . 2) (4 . 4) (8 . 8) (16 . 16))

3.4.2 Building a loadable library

To bundle multiple modules into a single object file that can be dynamically loaded with the load procedure, a flat link file is needed. The compiler’s ‘-o’ option must be used to name the C file generated as follows. If the dynamically loadable object file is to be named ‘myfile.on’ then the ‘-o’ option must set the name of the link file generated to ‘myfile.on.c’ (note that the ‘.c’ extension could also be ‘.cc’, ‘.cpp’ or whatever extension is appropriate for C/C++ source files). The three modules of the previous example can be bundled by generating a link file in this way:

\$ gsc -link -flat -o foo.01.c m2 m3
m2:
m3:
*** WARNING -- "cons" is not defined,
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.

When compiling the C files and link file generated, the flag ‘-D___DYNAMIC’ must be passed to the C compiler and the C compiler and linker must be told to generate a dynamically loadable shared library.

Here is an example under Mac OS X:

```
$ uname -srmp
Darwin 10.5.0 i386 i386
$ gsc -link -flat -o foo.o1.c m2 m3 > /dev/null
m2:
m3:
$ gsc -cc-options "-D___DYNAMIC" -obj m1.c m2.c m3.c foo.o1.c
m1.c:
m2.c:
m3.c:
foo.o1.c:
$ gcc -bundle m1.o m2.o m3.o foo.o1.o -o foo.o1
$ gsi foo.o1
((2 . 2) (4 . 4) (8 . 8) (16 . 16))
```

Here is an example under Linux:

```
$ uname -srmp
Linux 2.6.8-1.521 i686 athlon
$ gsc -link -flat -o foo.o1.c m2 m3 > /dev/null
m2:
m3:
$ gsc -cc-options "-D___DYNAMIC" -obj m1.c m2.c m3.c foo.o1.c
m1.c:
m2.c:
m3.c:
foo.o1.c:
$ gcc -shared m1.o m2.o m3.o foo.o1.o -o foo.o1
$ gsi foo.o1
((2 . 2) (4 . 4) (8 . 8) (16 . 16))
```

Here is a more complex example, under Solaris, which shows how to build a loadable library ‘mymod.o1’ composed of the files ‘m4.scm’, ‘m5.scm’ and ‘x.c’ that links to system shared libraries (for X-windows):

```
$ uname -srmp
SunOS ungava 5.6 Generic_105181-05 sun4m Sparc SUNW,Sparcstation-20
$ gsc -link -flat -o mymod.o1.c m4 m5
m4:
m5:
*** WARNING -- "*" is not defined,
*** referenced in: ("m4.c")
*** WARNING -- "+" is not defined,
```

Here is an example under Solaris, which shows how to build a loadable library ‘mymod.o1’ composed of the files ‘m4.scm’, ‘m5.scm’ and ‘x.c’ that links to system shared libraries (for X-windows):

```
$ uname -srmp
SunOS ungava 5.6 Generic_105181-05 sun4m Sparc SUNW,Sparcstation-20
$ gsc -link -flat -o mymod.o1.c m4 m5
m4:
m5:
*** WARNING -- "*" is not defined,
*** referenced in: ("m4.c")
*** WARNING -- "+" is not defined,
```
Chapter 3: The Gambit Scheme compiler

*** referenced in: ("m5.c")
*** WARNING -- "display" is not defined,
*** referenced in: ("m5.c" "m4.c")
*** WARNING -- "newline" is not defined,
*** referenced in: ("m5.c" "m4.c")
*** WARNING -- "write" is not defined,
*** referenced in: ("m5.c")

$ gsc -cc-options "-D___DYNAMIC" -obj m4.c m5.c x.c mymod.o1.c

m4.c:
m5.c:
x.c:
mymod.o1.c:

$/usr/ccs/bin/ld -G -o mymod.o1 mymod.o1.o m4.o m5.o x.o -lX11 -lsocket

gsi mymod.o1

hello from m4
hello from m5
(f1 10) = 22

$ cat m4.scm
(define (f1 x) (* 2 (f2 x)))
(display "hello from m4")
newline

(c-declare #<<c-declare-end
#include "x.h"
c-declare-end)
(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 m5.scm
(define (f2 x) (+ x 1))
(display "hello from m5")
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 -srmp
Linux bailey 1.2.13 #2 Wed Aug 28 16:29:41 GMT 1996 i586
$ gsc -link -o mylib.c m2
$ gsc -obj -cc-options "-D___SHARED" m1.c m2.c mylib.c
m1.c:
m2.c:
mylib.c:
$ gcc -shared m1.o m2.o mylib.o -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 -link -l mylib m3
$ gsc -obj m3.c m3_.c
m3.c:
m3_.c:
$ gcc m3.o m3_.o 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

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. It has been observed that lower levels of optimization (e.g. ‘-O1’) often give faster compilation and also generate faster code. It is a good idea to experiment.

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
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: \texttt{-shared}', \texttt{-call\_shared}', \texttt{-rdynamic}', \texttt{-fpic}', \texttt{-fPIC}', \texttt{-Kpic}', \texttt{-KPIC}', \texttt{-pic}', \texttt{+z}', and \texttt{-G}. Check your compiler’s documentation to see which flag you need.

### 3.5 Procedures specific to compiler

The Gambit Scheme compiler features the following procedures that are not available in the Gambit Scheme interpreter.

\begin{verbatim}
(compile-file-to-c file [options: options] [output: output] [module-name: module-name])

The \texttt{file} parameter must be a string naming an existing file containing Scheme source code. The extension can be omitted from \texttt{file} when the Scheme file has a \texttt{.scm}' or \texttt{.six}' extension. This procedure compiles the source file into a file containing C code. By default, this file is named after \texttt{file} with the extension replaced with \texttt{.c}'. The name of the generated file can be specified with the \texttt{output} parameter. If \texttt{output} is a string naming a directory then the C file is created in that directory. Otherwise the name of the C file is \texttt{output}. The name of the generated module can be specified with the \texttt{module-name} parameter. If \texttt{module-name} is \#f or is not specified, then the name of the module is derived from the name of the C file generated, without the extension.

Compilation options are specified through the \texttt{options} parameter which must be a list of symbols. Any combination of the following options can be used: ‘verbose’, ‘report’, ‘expansion’, ‘gvm’, and ‘debug’.

When the compilation is successful, \texttt{compile-file-to-c} returns the name of the C file generated. When there is a compilation error, \#f is returned.

\$ \texttt{cat h.scm}
\texttt{(display }"\text{hello}\text{")} \ (\texttt{newline})
\$ \texttt{gsc}
\texttt{Gambit v4.6.1}

\>
\texttt{(compile-file-to-c "h")}
\texttt{"/Users/feeley/gambit/doc/h.c"}
\end{verbatim}

\begin{verbatim}
(compile-file file [options: options] [output: output] [cc-options: cc-options] [ld-options-prelude: ld-options-prelude] [ld-options: ld-options])

The \texttt{file}, \texttt{options}, and \texttt{output} parameters have the same meaning as for the \texttt{compile-file-to-c} procedure. The \texttt{cc-options} parameter is a string containing the options to pass to the C compiler and the \texttt{ld-options-prelude} and \texttt{ld-options} parameters are strings containing the options to pass to the C linker (the options in \texttt{ld-options-prelude} are passed to the C linker before the input file and the options in \texttt{ld-options} are passed after).

The \texttt{compile-file} procedure compiles the source file \texttt{file} into an object file, which is either a file dynamically loadable using the \texttt{load} procedure, or a C linkable object file.
destined to be linked with the C linker (for example to create a standalone executable program). The \texttt{file} may be a Scheme source file or a C file possibly generated by the Gambit Scheme compiler (for example with the \texttt{compile-file-to-c} procedure). The presence of the \texttt{obj} option in \texttt{options} will cause the creation of a C linkable object file and therefore the options \texttt{ld-options-prelude} and \texttt{ld-options} are ignored, otherwise a dynamically loadable file is created. In both cases, if \texttt{file} is a Scheme source file, the compiler first compiles \texttt{file} to a C file which is created in the same directory as \texttt{file} regardless of the \texttt{output} parameter. Then the C file is compiled with the C compiler.

When the compilation is successful, \texttt{compile-file} returns the name of the object file generated. When there is a compilation error, \texttt{#f} is returned.

The name of the object file can be specified with the \texttt{output} parameter. If \texttt{output} is a string naming a directory then the object file is created in that directory. Otherwise the name of the object file is \texttt{output}.

In the case of a dynamically loadable object file, by default the object file is named after \texttt{file} with the extension replaced with \texttt{'.on'}, where \texttt{n} is a positive integer that acts as a version number. The next available version number is generated automatically by \texttt{compile-file}.

When dynamically loaded object files are loaded using the \texttt{load} procedure, the \texttt{'.on'} extension can be specified (to select a particular version) or omitted (to load the file with a \texttt{'.on'} extension with the highest \texttt{n} consecutively from 1). When the \texttt{'.on'} extension is not specified and older versions are no longer needed, all versions must be deleted and the compilation must be repeated (this is necessary because the file name, including the extension, is used to name some of the exported symbols of the object file).

Note that dynamically loadable object files can only be generated on host operating systems that support dynamic loading.

\begin{verbatim}
$ cat h.scm
  (display "hello") (newline)
$ gsc
Gambit v4.6.1

> (compile-file "h")
"/Users/feeley/gambit/doc/h.o1"
> (load "h")
hello
"/Users/feeley/gambit/doc/h.o1"
> (compile-file-to-c "h" output: "h.o99.c")
"/Users/feeley/gambit/doc/h.o99.c"
> (compile-file "h.o99.c")
"/Users/feeley/gambit/doc/h.o99"
> (load "h.o99")
hello
"/Users/feeley/gambit/doc/h.o99"
> (compile-file-to-c "h")
"/Users/feeley/gambit/doc/h.c"
> (compile-file "h.c" options: '(obj))
"/Users/feeley/gambit/doc/h.o"
\end{verbatim}
(link-incremental module-list [output: output] [base: base] procedure
  [warnings?: warnings?])

The first parameter must be a non empty list of strings naming Scheme modules to link
(extensions must be omitted). 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. The name of the generated link file can be specified
with the output parameter. If output is a string naming a directory then the link file
is created in that directory. Otherwise the name of the link file is output.

The base link file is specified by the base parameter, which must be a string. By
default the base link file is the Gambit runtime library link file ‘~lib/_gambc.c’. However, when base is supplied the base link file is named ‘base.c’.

The warnings? parameter controls whether warnings are generated for undefined
references.

The following example shows how to build the executable program ‘hello’ which
contains the two Scheme modules ‘h.scm’ and ‘w.six’.

$ uname -srmp
Darwin 8.1.0 Power Macintosh powerpc
$ cat h.scm
(display "hello") (newline)
$ cat w.six
display("world"); newline();
$ gsc
Gambit v4.6.1

> (compile-file-to-c "h")
"/Users/feeley/gambit/doc/h.c"
> (compile-file-to-c "w")
"/Users/feeley/gambit/doc/w.c"
> (link-incremental '("h" "w") output: "hello.c")
"/Users/feeley/gambit/doc/hello_.c"
> ,q
$ gsc -obj h.c w.c hello.c
h.c:
  w.c:
  hello.c:
$ gcc h.o w.o hello.o -lgambc -o hello
$ ./hello
hello
world

(link-flat module-list [output: output] [warnings?: warnings?]) procedure

The first parameter must be a non empty list of strings naming Scheme modules to link.
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). 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. The name of the generated link file can be specified with the output
parameter. If output is a string naming a directory then the link file is created in that
directory. Otherwise the name of the link file is output. If a dynamically loadable
object file is produced from the link file ‘output’, then the name of the dynamically
loadable object file must be ‘output’ stripped of its file extension.
Chapter 3: The Gambit Scheme compiler

The `warnings?` parameter controls whether warnings are generated for undefined references.

The following example shows how to build the dynamically loadable object file `lib.o1` which contains the two Scheme modules `m6.scm` and `m7.scm`.

```bash
$ uname -srmp
Darwin 8.1.0 Power Macintosh powerpc
$ cat m6.scm
(define (f x) (g (* x x)))
$ cat m7.scm
(define (g y) (+ n y))
$ gsc
Gambit v4.6.1
> (compile-file-to-c "m6")
"/Users/feeley/gambit/doc/m6.c"
> (compile-file-to-c "m7")
"/Users/feeley/gambit/doc/m7.c"
> (link-flat '("m6" "m7") output: "lib.o1.c")
*** WARNING -- "*" is not defined,
*** referenced in: ("m6.c")
*** WARNING -- "+" is not defined,
*** referenced in: ("m7.c")
*** WARNING -- "n" is not defined,
*** referenced in: ("m7.c")
"/Users/feeley/gambit/doc/lib.o1.c"
> ,q
$ gcc -bundle -D___DYNAMIC m6.c m7.c lib.o1.c -o lib.o1
$ gsc
Gambit v4.6.1
> (load "lib")
*** WARNING -- Variable "n" used in module "m7" is undefined
"/Users/feeley/gambit/doc/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.
4 Runtime options

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:

- `mHEAPSIZE` Set minimum heap size in kilobytes.
- `hHEAPSIZE` Set maximum heap size in kilobytes.
- `lLIVEPERCENT` Set heap occupation after garbage collection.
- `s` Select standard Scheme mode.
- `S` Select Gambit Scheme mode.
- `d[OPT...]` Set debugging options.
- `@[INTF]:PORT` Override the configuration of the main RPC server.
- `=DIRECTORY` Override the central installation directory.
- `˜˜DIR=DIRECTORY` Override the DIR installation directory.
- `+ARGUMENT` Add ARGUMENT to the command line before other arguments.
- `f[OPT...]` Set file options.
- `t[OPT...]` Set terminal options.
- `-[OPT...]` Set standard input and output options.

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

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

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

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

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

- `p` Uncaught exceptions will be treated as “errors” in the primordial thread only.
Uncaught exceptions will be treated as “errors” in all threads.

When an “error” occurs a new REPL will be started.

When an “error” occurs a new REPL will be started. Moreover the program starts in single-stepping mode.

When an “error” occurs the program will terminate with a nonzero exit status.

When a user interrupt occurs a new REPL will be started. User interrupts are typically obtained by typing \( \texttt{^C} \). Note that with some system configurations \( \texttt{^C} \) abruptly terminates the process. For example, under Microsoft Windows, \( \texttt{^C} \) works fine with the standard console but with the MSYS terminal window it terminates the process.

When a user interrupt occurs it will be deferred until the parameter \texttt{current-user-interrupt-handler} is bound.

When a user interrupt occurs the program will terminate with a nonzero exit status.

The verbosity level is set to \texttt{LEVEL} (a digit from 0 to 9). At level 0 the runtime system will not display error messages and warnings.

The REPL interaction channel will be the IDE REPL window (if the IDE is available).

The REPL interaction channel will be the console.

The REPL interaction channel will be standard input and standard output.

The REPL interaction channel will be connected to the remote debugger at address \texttt{HOST:PORT} (if there is a remote debugger at that address). The default \texttt{HOST} is 127.0.0.1 and the default \texttt{PORT} is 44555. THIS OPTION IS NOT YET IMPLEMENTED!

The default debugging options are equivalent to \texttt{-:dpq1i} (i.e. an uncaught exception in the primordial thread terminates the program after displaying an error message). When the letter ‘d’ is not followed by suboptions, it is equivalent to \texttt{-:dpr1i} (i.e. a new REPL is started only when an uncaught exception occurs in the primordial thread). When \texttt{gsi} and \texttt{gsc} are running the main REPL, the debugging options are changed to cause errors in the primordial thread and user interrupts to start a nested REPL.

overrides the configuration of the main RPC server. The default \texttt{INTF} is 127.0.0.1 and the default \texttt{PORT} is 44556. THIS OPTION IS NOT YET IMPLEMENTED!

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

The ‘~DIR=DIRECTORY’ option overrides the setting of the \texttt{DIR} installation directory.

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

The ‘f’, ‘t’ and ‘-’ options specify the default settings of the ports created for files, terminals and standard input and output respectively. The default character encoding, end-of-line encoding and buffering can be set. Moreover, for terminals the line-editing feature can be enabled or disabled. The ‘f’, ‘t’ and ‘-’ must be followed by a sequence of these options:
A  ASCII character encoding.
1  ISO-8859-1 character encoding.
2  UCS-2 character encoding.
4  UCS-4 character encoding.
6  UTF-16 character encoding.
8  UTF-8 character encoding.
U  UTF character encoding with fallback to UTF-8 on input if no BOM is present.
UA UTF character encoding with fallback to ASCII on input if no BOM is present.
U1 UTF character encoding with fallback to ISO-8859-1 on input if no BOM is present.
U6 UTF character encoding with fallback to UTF-16 on input if no BOM is present.
U8 UTF character encoding with fallback to UTF-8 on input if no BOM is present.
c End-of-line is encoded as CR (carriage-return).
l End-of-line is encoded as LF (linefeed)
c1 End-of-line is encoded as CR-LF.
u Unbuffered I/O.
n Line buffered I/O (‘\n’ for “at newline”).
f Fully buffered I/O.
r Illegal character encoding is treated as an error (exception raised).
R Silently replace illegal character encodings with Unicode character #xfffd (replacement character).
e Enable line-editing (applies to terminals only).
E Disable line-editing (applies to terminals only).

When a program’s execution starts, the runtime system obtains the runtime options by processing in turn four sources of runtime options: the defaults, the environment variable ‘GAMBCOPT’, the script line of the source code, and the first command line argument of the program. Any runtime option can be overridden by a subsequent source of runtime options. It is sometimes useful to prevent overriding the runtime options of the script line. This can be achieved by starting the script line runtime options with ‘-::’. In this case the environment variable ‘GAMBCOPT’ is ignored, and the first command line argument of the program is not used for runtime options (it is treated like a normal command line argument). For example:
$ \text{GAMBCOPT}=\text{d0,}=^\sim\text{/my-gambit2}
$ \text{export} \ \text{GAMBCOPT}
$ \text{gsi} \ -e \ '(\text{pretty-print} \ (\text{path-expand} \ "^\sim\") \ (/ \ 1 \ 0)\)'
"/\text{Users/feeley/my-gambit2}/"
$ \text{echo} \ \$?
70
$ \text{gsi} \ -d1 \ -e \ '(\text{pretty-print} \ (\text{path-expand} \ "^\sim\") \ (/ \ 1 \ 0)\)'
"/\text{Users/feeley/my-gambit2}/"
*** \text{ERROR IN} \ (\text{string}@1.3) \ -- \ \text{Divide by zero}
(/ \ 1 \ 0)
5 Debugging

5.1 Debugging model

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

a. An evaluation error has occurred, such as attempting to divide by zero.

b. The user has interrupted the evaluation (usually by typing `\C`).

c. A breakpoint has been reached or (step) was evaluated.

d. Single-stepping mode is enabled.

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

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

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

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

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

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

,? 
Give a summary of the REPL commands.

,(h subject) 
This command will show the section of the Gambit manual with the definition of the procedure or special form subject, which must be a symbol. For example, ’,(h time)’ will show the section documenting the time special form. Please see the help procedure for additional information.

,h 
This command will show the section of the Gambit manual with the definition of the procedure which raised the exception for which this REPL was started.

,q 
Terminate the process with exit status 0. This is equivalent to calling (exit 0).

,qt 
Terminate the current thread (note that terminating the primordial thread terminates the process).

,t 
Return to the outermost REPL, also known as the “top-level REPL”.

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

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

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

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

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

, N
Move to frame number N of the continuation. After changing the current frame, a one-line summary of the frame is displayed as if the ‘, y’ command was entered.

, N+
Move forward by N frames 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.

, N-
Move backward by N frames in the chain of continuation frames (i.e. towards more recent continuation frames). After changing the current frame, a one-line summary of the frame is displayed as if the ‘, y’ command was entered.

, +
Equivalent to ‘, 1+’.

, -
Equivalent to ‘, 1-’.

, ++
Equivalent to ‘, N+’ where N is the number of continuation frames displayed at the head of a backtrace.

, --
Equivalent to ‘, N-’ where N is the number of continuation frames displayed at the head of a backtrace.

, 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 or by a compiled procedure not compiled with the declaration ‘debug-location’, and the last field is missing when the frame was created by a compiled procedure not compiled with the declaration ‘debug-source’.

, 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.

,be
Like the ‘,b’ command but also display the environment.

,bed
Like the ‘,be’ command but also display the dynamic environment.

,(b expr)
Display the backtrace of expr’s value, X, which is obtained by evaluating expr in the current frame. X must be a continuation or a thread. When X is a continuation, the frames in that continuation are displayed. When X is a thread, the backtrace of the current continuation of that thread is displayed.

,(be expr)
Like the ‘,(b expr)’ command but also display the environment.

,(bed expr)
Like the ‘,(be expr)’ command but also display the dynamic environment.

,i
Pretty print the procedure that created the current frame or ‘(interaction)’ if the frame was created by an expression entered at the REPL. Compiled procedures will only be pretty printed when they are compiled with the declaration ‘debug-source’.

,e
Display the environment which is accessible from the current frame. The lexical environment is displayed, followed by the dynamic environment if the parameter object repl-display-dynamic-environment? is not false. Global lexical variables are not displayed. Moreover the frame must have been created by interpreted code or code compiled with the declaration ‘debug-environments’. Due to space safety considerations and compiler optimizations, some of the lexical variable bindings may be missing. Lexical variable bindings are displayed using the format ‘variable = expression’ (when variable is mutable) or ‘variable == expression’ (when variable is immutable, which may happen in compiled code due to compiler optimization) and dynamically-bound parameter bindings are displayed using the format ‘(parameter) = expression’. Note that expression can be a self-evaluating expression (number, string, boolean, character, ...), a quoted expression, a lambda expression or a global variable (the last two cases, which are only used when the value of the variable or parameter is a procedure, simplifies the debugging of higher-order procedures). A parameter can be a quoted expression or a global variable. Lexical bindings are displayed in inverse binding order (most deeply nested first) and shadowed variables are included in the list.

,ed
Like the ‘,e’ command but the dynamic environment is always displayed.

,(e expr)
Display the environment of expr’s value, X, which is obtained by evaluating expr in the current frame. X must be a continuation, a thread, a procedure, or a nonnegative integer. When X is a continuation, the environment at that point in the code is displayed. When X is a thread, the environment of the current continuation of that thread is displayed.
When $X$ is a procedure, the lexical environment where $X$ was created is combined with the current continuation and this combined environment is displayed. When $X$ is an integer, the environment at frame number $X$ of the continuation is displayed.

\(, (ed \ expr)\)  
Like the ‘, (e \ expr)’ command but the dynamic environment is always displayed.

\(, st\)  
Display the state of the threads in the current thread’s thread group. A thread can be: uninitialized, initialized, active, and terminated (normally or abnormally). Active threads can be running, sleeping and waiting on a synchronization object (mutex, condition variable or port) possibly with a timeout.

\(, (st \ expr)\)  
Display the state of a specific thread or thread group. The value of \(expr\) must be a thread or thread group.

\(, (v \ expr)\)  
Start a new REPL visiting \(expr\)’s value, $X$, which is obtained by evaluating \(expr\) in the current frame. $X$ must be a continuation, a thread, a procedure, or a nonnegative integer. When $X$ is a continuation, the new REPL’s continuation is $X$ and evaluations are done in the environment at that point in the code. When $X$ is a thread, the thread is interrupted and the new REPL’s continuation is the point where the thread was interrupted. When $X$ is a procedure, the lexical environment where $X$ was created is combined with the current continuation and evaluations are done in this combined environment. When $X$ is an integer, the REPL is started in frame number $X$ of the continuation.

## 5.3 Debugging example

Here is a sample interaction with gsi:

```scheme
$ gsi
Gambit v4.6.1

> (define (invsqr x) (/ 1 (expt x 2)))
> (define (mymap fn lst)
  (define (mm in)
    (if (null? in)
      '()
      (cons (fn (car in)) (mm (cdr in)))))
  (mm lst))
> (mymap invsqr '(5 2 hello 9 1))
*** ERROR IN invsqr, (console)@1.25 -- (Argument 1) NUMBER expected (expt 'hello 2)
1> ,i
#<procedure #2 invsqr> =
(lambda (x) (/ 1 (expt x 2)))
1> ,e
x = 'hello
1> ,b
0 invsqr           (console)@1:25 (expt x 2)
1 #<procedure #4>   (console)@6:17 (fn (car in))
2 #<procedure #4>   (console)@6:31 (mm (cdr in))
3 #<procedure #4>   (console)@6:31 (mm (cdr in))
```
4 (interaction) (console)@8:1 (mymap invsqr '(5 2 hel...)
1> ;
1 #<procedure #4> (console)@6.17 (fn (car in))
1\1> (pp #4)
(lambda (in) (if (null? in) '() (cons (fn (car in)) (mm (cdr in)))))
1\1> ,e
in = '(hello 9 1)
mm = (lambda (in) (if (null? in) '() (cons (fn (car in)) (mm (cdr in)))))
fn = invsqr
lst = '(5 2 hello 9 1)
1\1> ,(e mm)
mm = (lambda (in) (if (null? in) '() (cons (fn (car in)) (mm (cdr in)))))
fn = invsqr
lst = '(5 2 hello 9 1)
1\1> fn
#<procedure #2 invsqr>
1\1> (pp fn)
(lambda (x) (/ 1 (expt x 2)))
1\1> ,+
2 #<procedure #4> (console)@6.31 (mm (cdr in))
1\2> ,e
in = '(2 hello 9 1)
mm = (lambda (in) (if (null? in) '() (cons (fn (car in)) (mm (cdr in)))))
fn = invsqr
lst = '(5 2 hello 9 1)
1\2> ,(c (list 3 4 5))
(1/25 1/4 3 4 5)
> ,q

5.4 Procedures related to debugging

(help subject) procedure
(help-browser [new-value]) procedure

The help procedure displays the section of the Gambit manual with the definition
of the procedure or special form subject, which must be a procedure or symbol. For
example the call (help gensym) will show the section documenting the gensym
procedure and the call (help 'time) will show the section documenting the time
special form. The help procedure returns the void object.

The parameter object help-browser is bound to a string naming the external
program that is used by the help procedure to view the documentation. Initially it is bound to the empty string. In normal circumstances when
help-browser is bound to an empty string the help procedure runs the script
"bin/gambc-doc.bat" which searches for a suitable web browser to open the
documentation in HTML format. Unless the system was built with the command
'configure --enable-help-browser=...', the text-only browser 'lynx'
(see http://lynx.isc.org/) will be used by default if it is available. We
highly recommend that you install this browser if you are interested in viewing the
documentation within the console in which the REPL is running. You can exit
'lynx' conveniently by typing an end of file (usually ⟨CTRL-D⟩).

For example:

> (help-browser "firefox") ; use firefox instead of lynx
> (help 'gensym)
> (help gensym) ; OK because gensym is a procedure
> (help 'time)
> (help time) ; not OK because time is a special form

*** ERROR IN (console)@5.7 -- Macro name can’t be used as a variable: time

(repl-result-history-ref i) procedure
(repl-result-history-max-length-set! n) procedure

The REPL keeps a history of the last few results printed by the REPL. The call (repl-result-history-ref i) returns the ith previous result (the last for i=0, the next to last for i=1, etc). By default the REPL result history remembers up to 3 results. The maximal length of the history can be set to n between 0 and 10 by a call to (repl-result-history-max-length-set! n).

For convenience the reader defines an abbreviation for calling repl-result-history-ref. Tokens formed by a sequence of one or more hash signs, such as '#', '##', etc, are expanded by the reader into the list (repl-result-history-ref i), where i is the number of hash signs minus 1. In other words, '#' will return the last result printed by the REPL, '##' will return the next to last, etc.

For example:

> (map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
> (reverse #)
(9 4 1)
> (append # ##)
(9 4 1 1 4 9)
> 1
1
> (+ # ##)
2
> (+ # ##)
3
> (+ # ##)
5
> ####
*** ERROR IN (console)@9.1 -- (Argument 1) Out of range
(repl-result-history-ref 3)
1>

(trace proc...) procedure
(untrace proc...) procedure

The trace procedure 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).

The untrace procedure stops tracing calls to the specified procedures. When no
argument is passed to the trace procedure, the list of procedures currently being
traced is returned. The void object is returned by the trace procedure when it
is passed one or more arguments. When no argument is passed to the untrace
procedure 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 #2 fact-iter> #<procedure #3 -> #<procedure #4 fact>)
> (untrace)
> (fact 5)
120
```

(step)
The `step` procedure 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 (console)@3.15
1> ,s
   | > (fact 5)
*** STOPPED IN fact, (console)@1.22
1> ,s
   |   | > (< n 2)
   |   |   #f
*** STOPPED IN fact, (console)@1.43
1> ,s
   |   | > (- n 1)
   |   |   4
*** STOPPED IN fact, (console)@1.37
1> ,s
   |   |   | > (fact (- n 1))
*** STOPPED IN fact, (console)@1.22
1> ,s
   |   |   |   | > (< n 2)
   |   |   |   |   #f
*** STOPPED IN fact, (console)@1.43
1> ,s
   |   |   |   | > (- n 1)
   |   |   |   |   3
*** STOPPED IN fact, (console)@1.37
1> ,l
   |   |   |   | > (fact (- n 1))
*** STOPPED IN fact, (console)@1.22
```
The `break` procedure 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.

The `unbreak` procedure 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:

```lisp
> (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, (console)@1.21
1> ,b
0 double (console)@1:21 +
1 triple (console)@2:31 (double y)
2 f (console)@3:18 (triple z)
3 (interaction) (console)@6:1 (f 5)
1> ,e
x = 5
1> ,c
*** STOPPED IN double, (console)@1.21
1> ,c
*** STOPPED IN f, (console)@3.29
1> ,c
150>
(break)
(#<procedure #3 -> #<procedure #4 double>)
> (unbreak)
> (f 5)
150
```

### (generate-proper-tail-calls [new-value])

[Note: this procedure is DEPRECATED and will be removed in a future version of Gambit. Use the ‘proper-tail-calls’ declaration instead.]

The parameter object `generate-proper-tail-calls` is bound to a boolean value controlling how the interpreter handles tail calls. When it is bound to #f the inter-
preter 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 initial value of this parameter object is #t, which means that a tail call will reuse the continuation of the calling function.

This parameter object only affects code that is subsequently processed by load or eval, or entered at the REPL.

For example:

```scheme
> (generate-proper-tail-calls)
#t
> (let loop ((i 1)) (if (< i 10) (loop (* i 2)) oops))
*** ERROR IN #<procedure #2>, (console)@2.47 -- Unbound variable: oops
1>,b
0 #<procedure #2> (console)@2:47 oops
1 (interaction) (console)@2:1 ((letrec ((loop (lambda...
1>,b
> (generate-proper-tail-calls #f)
> (let loop ((i 1)) (if (< i 10) (loop (* i 2)) oops))
*** ERROR IN #<procedure #3>, (console)@6.47 -- Unbound variable: oops
1>,b
0 #<procedure #3> (console)@6:47 oops
1 #<procedure #3> (console)@6:32 (loop (* i 2))
2 #<procedure #3> (console)@6:32 (loop (* i 2))
3 #<procedure #3> (console)@6:32 (loop (* i 2))
4 #<procedure #3> (console)@6:32 (loop (* i 2))
5 (interaction) (console)@6:1 ((letrec ((loop (lambda...
```

(display-environment-set! display?)

[Note: this procedure is DEPRECATED and will be removed in a future version of Gambit. Use the parameter object repl-display-environment? instead.]

This procedure 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.

(repl-display-environment? display?)

The parameter object repl-display-environment? is bound to a boolean value 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. This is particularly useful in single-stepping mode. The default setting is not to display the environment.

(display-dynamic-environment? display?)

The parameter object display-dynamic-environment? is bound to a boolean value that controls whether the dynamic environment is displayed when the environment is displayed. The default setting is not to display the dynamic environment.

(pretty-print obj [port])

This procedure pretty-prints obj on the port port. If it is not specified, port defaults to the current output-port.

For example:
(let* ((x '(1 2 3 4)) (y (list x x x))) (list y y y))

(((1 2 3 4) (1 2 3 4) (1 2 3 4))
 ((1 2 3 4) (1 2 3 4) (1 2 3 4))
 ((1 2 3 4) (1 2 3 4) (1 2 3 4)))

(procedure pp obj [port])

This procedure pretty-prints obj on the port port. When obj is a procedure created by the interpreter or a procedure created by code compiled with the declaration ‘debug-source’, the procedure’s source code is displayed. If it is not specified, port defaults to the interaction channel (i.e. the output will appear at the REPL).

For example:

> (define (f g) (+ (time (g 100)) (time (g 1000))))
> (pp f)
(lambda (g)
 (+ (##time (lambda () (g 100)) ’(g 100))
  (##time (lambda () (g 1000)) ’(g 1000))))

(procedure gc-report-set! report?)

This procedure 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 megabytes since the program was started, the size of the heap in megabytes, the heap memory in megabytes occupied by live data, the proportion of the heap occupied by live data, and the number of bytes occupied by movable and nonmovable objects.

5.5 Console line-editing

The console implements a simple Scheme-friendly line-editing user-interface that is enabled by default. It offers parentheses balancing, a history of previous commands, symbol completion, and several emacs-compatible keyboard commands. The user’s input is displayed in a bold font and the output produced by the system is in a plain font. The history of previous commands is saved in the file ‘˜/.gambc_history’. It is restored when a REPL is started.

Symbol completion is triggered with the tab key. When the cursor is after a sequence of characters that can form a symbol, typing the tab key will search the symbol table for the first symbol (in alphabetical order) that begins with that sequence and insert that symbol. Typing the tab key in succession will cycle through all symbols with that prefix. When all possible symbols have been shown or there are no possible completions, the text reverts to the uncompleted symbol and the bell is rung.

Here are the keyboard commands available (where the ‘M-’ prefix means the escape key is typed and the ‘C-’ prefix means the control key is pressed):

C-d Generate an end-of-file when the line is empty, otherwise delete character at cursor.

delete or backspace Delete character before cursor.

M-C-d Delete word forward and keep a copy of this text on the clipboard.
M-delete Delete word backward and keep a copy of this text on the clipboard.
M-backspace Delete S-expression backward and keep a copy of this text on the clipboard.
C-a Move cursor to beginning of line.
C-e Move cursor to end of line.
C-b or left-arrow Move cursor left one character.
M-b Move cursor left one word.
M-C-b or M-left-arrow Move cursor left one S-expression.
C-f or right-arrow Move cursor right one character.
M-f Move cursor right one word.
M-C-f or M-right-arrow Move cursor right one S-expression.
C-p or M-p or up-arrow Move to previous line in history.
C-n or M-n or down-arrow Move to next line in history.
C-t Transpose character at cursor with previous character.
M-t Transpose word after cursor with previous word.
M-C-t Transpose S-expression after cursor with previous S-expression.
C-l Clear console and redraw line being edited.
C-nul Set the mark to the cursor.
C-w Delete the text between the cursor and the mark and keep a copy of this text on the clipboard.
C-k Delete the text from the cursor to the end of the line and keep a copy of this text on the clipboard.
C-y Paste the text that is on the clipboard.
F8 Same as typing ‘#||#, c;’ (REPL command to continue the computation).
F9 Same as typing ‘#||#, -;’ (REPL command to move to newer frame).
F10 Same as typing ‘#||#, +;’ (REPL command to move to older frame).
F11 Same as typing ‘#||#, s;’ (REPL command to step the computation).
F12 Same as typing ‘#||#, l;’ (REPL command to leap the computation).

On Mac OS X, depending on your configuration, you may have to press the fn key to access the function key F12 and the option key to access the other function keys.

On Microsoft Windows the clipboard is the system clipboard. This allows text to be copied and pasted between the program and other applications. On other operating systems the clipboard is internal to the program (it is not integrated with the operating system).
5.6 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 ‘@line.column’ where stream is either ‘(stdin)’ when the expression was obtained from standard input, ‘(console)’ when the expression was obtained from the console, 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:

```lisp
(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 -:d-")
```

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

```lisp
(require 'gambit)
```

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

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

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

- **F8 or C-c c**: Continue the computation (same as typing ‘#||#,c;’ to the REPL).
- **F9 or C-c ]**: Move to newer frame (same as typing ‘#||#,,-;’ to the REPL).
- **F10 or C-c [**: Move to older frame (same as typing ‘#||#+;’ to the REPL).
- **F11 or C-c s**: Step the computation (same as typing ‘#||#,s;’ to the REPL).
- **F12 or C-c l**: Leap the computation (same as typing ‘#||#,l;’ to the REPL).
- **C-c _**: Removes the last window that was opened to highlight an expression.

The two keystroke version of these commands can be shortened to ‘M-c’, ‘M-’’, ‘M-’’, ‘M-s’, ‘M-l’, and ‘M-_’ respectively by adding this line to your ‘.emacs’ file:

```lisp
(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 ; add directory containing gambit.el
  (cons "/usr/local/Gambit-C/share/emacs/site-lisp"
    load-path))
(setq scheme-program-name "/tmp/gsi -:d-" ; 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)

5.7 GUIDE

The implementation and documentation for GUIDE, the Gambit Universal IDE, are not yet complete.
6 Scheme extensions

6.1 Extensions to standard procedures

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

These procedures do nothing.

(procedure (call-with-current-continuation proc))
(procedure (call/cc proc))

The procedure call-with-current-continuation is bound to the global variables call-with-current-continuation and call/cc.

6.2 Extensions to standard special forms

(special form (lambda lambda-formals body))
(special form (define (variable define-formals) body))

lambda-formals = (formal-argument-list) | r4rs-lambda-formals
define-formals = formal-argument-list | r4rs-define-formals
formal-argument-list = dsssl-formal-argument-list | rest-at-end-formal-argument-list
dsssl-formal-argument-list = reqs opts rest keys
rest-at-end-formal-argument-list = reqs opts keys rest | reqs opts keys . rest-formal-argument
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 formal arguments, either positional or named, and support 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).

If the formal-argument-list matches dsssl-formal-argument-list or extended-formal-argument-list they are processed as follows:
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 #!key does not appear in the formal-argument-list and there is no rest-formal-argument then it shall be an error if there are any remaining actual arguments.

d. If #!key does not appear in the formal-argument-list and there is a rest-formal-argument then the rest-formal-argument is bound to a list of all remaining actual arguments.

e. If #!key appears in the formal-argument-list and there is no rest-formal-argument then 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. 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.

f. If #!key appears in the formal-argument-list and there is a rest-formal-argument before the #!key then there may be an even or odd number of remaining actual arguments and the rest-formal-argument is bound to a list of all remaining actual arguments. Then, these remaining actual arguments are scanned from left to right in pairs, stopping at the first pair whose first element is not a keyword. Each pair whose first element is a keyword matching the name of a keyword-formal-argument gives the value (i.e. the second element of the pair) of the corresponding 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.

g. If #!key appears in the formal-argument-list and there is a rest-formal-argument after the #!key then there may be an even or odd number of remaining actual arguments. The remaining actual arguments are scanned from left to right in pairs, stopping at the first pair whose first element is not a keyword. Each pair shall have as its first element a keyword matching the name of a keyword-formal-argument; the second element gives the value of the corresponding 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. Finally, the rest-formal-argument is bound to the list of the actual arguments that were not scanned (i.e. after the last keyword/value pair).

In all cases it is an error for a variable to appear more than once in a formal-argument-list.

Note that this specification is compatible with the DSSSL language standard (i.e. a correct DSSSL program will have the same semantics when run with Gambit).

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:

```scheme
> ((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 (k (* a b))) (list a b k))
> (define (h1 a #!rest r #!key k) (list a k r))
> (define (h2 a #!key k #!rest r) (list a k r))
> (f 1)
(1 #f)
> (f 1 2)
(1 2)
> (g 3)
(3 3 9)
> (g 3 4)
(3 4 12)
> (g 3 4 k: 5)
(3 4 5)
> (g 3 4 k: 5 k: 6)
(3 4 5)
> (h1 7)
(7 #f ())
> (h1 7 k: 8 9)
(7 8 (k: 8 9))
> (h1 7 k: 8 z: 9)
(7 8 (k: 8 z: 9))
> (h2 7)
(7 #f ())
> (h2 7 k: 8 9)
(7 8 (9))
> (h2 7 k: 8 z: 9)
*** ERROR IN (console)@17.1 -- Unknown keyword argument passed to procedure (h2 7 k: 8 z: 9)
```

6.3 Miscellaneous extensions

### (vector-copy vector) procedure

This procedure returns a newly allocated vector with the same content as the vector vector. Note that the elements are not recursively copied.
For example:

```scheme
> (define v1 '(1 2 3))
> (define v2 (vector-copy v1))
> v2
'(1 2 3)
> (eq? v1 v2)
#f
```

### subvector vector start end

This procedure is the vector analog of the substring procedure. It returns a newly allocated vector formed from the elements of the vector vector beginning with index start (inclusive) and ending with index end (exclusive).

For example:

```scheme
> (subvector '(a b c d e f) 3 5)
'(d e)
```

### vector-append vector...

This procedure is the vector analog of the string-append procedure. It returns a newly allocated vector whose elements form the concatenation of the given vectors.

For example:

```scheme
> (define v '#(1 2 3))
> (vector-append v v v)
'(1 2 3 1 2 3 1 2 3)
```

### append-vectors lst

This procedure returns a newly allocated vector whose elements form the concatenation of all the vectors in the list lst. It is equivalent to (apply vector-append lst).

For example:

```scheme
> (define v '#(1 2 3))
> (append-vectors (list v v v))
'(1 2 3 1 2 3 1 2 3)
```

### subvector-fill! vector start end fill

This procedure is like vector-fill!, but fills a selected part of the given vector. It sets the elements of the vector vector, beginning with index start (inclusive) and ending with index end (exclusive) to fill. The value returned is unspecified.

For example:

```scheme
> (define v (vector 'a 'b 'c 'd 'e 'f))
> (subvector-fill! v 3 5 'x)
> v
'(a b c x x f)
```

### subvector-move! src-vector src-start src-end dst-vector dst-start

This procedure replaces part of the contents of vector dst-vector with part of the contents of vector src-vector. It copies elements from src-vector, beginning with index src-start (inclusive) and ending with index src-end (exclusive) to dst-vector beginning with index dst-start (inclusive). The value returned is unspecified.

For example:
(define v1 '(1 2 3 4 5 6))
(define v2 (vector 'a 'b 'c 'd 'e 'f))
(subvector-move! v1 3 5 v2 1)
v2
  #(a 4 5 d e f)

(vector-shrink! vector k)

This procedure shortens the vector vector so that its new size is k. The value returned is unspecified.

For example:
> (define v (vector 'a 'b 'c 'd 'e 'f))
v  #(a b c d e f)
> (vector-shrink! v 3)
v  #(a b c)

(append-strings lst)

This procedure returns a newly allocated string whose elements form the concatenation of all the strings in the list lst. It is equivalent to (apply string-append lst).

For example:
> (define s "abc")
> (append-strings (list s s s))
  "abcabcabc"

(substring-fill! string start end fill)

This procedure is like string-fill!, but fills a selected part of the given string. It sets the elements of the string string, beginning with index start (inclusive) and ending with index end (exclusive) to fill. The value returned is unspecified.

For example:
> (define s (string #\a #\b #\c #\d #\e #\f))
> (substring-fill! s 3 5 #\x)
> s
  "abcxxf"

(substring-move! src-string src-start src-end dst-string dst-start)

This procedure replaces part of the contents of string dst-string with part of the contents of string src-string. It copies elements from src-string, beginning with index src-start (inclusive) and ending with index src-end (exclusive) to dst-string beginning with index dst-start (inclusive). The value returned is unspecified.

For example:
> (define s1 "123456")
> (define s2 (string #\a #\b #\c #\d #\e #\f))
> (substring-move! s1 3 5 s2 1)
> s2
  "a45def"

(string-shrink! string k)

This procedure shortens the string string so that its new size is k. The value returned is unspecified.

For example:
Chapter 6: Scheme extensions

> (define s (string #\a #\b #\c #\d #\e #\f))
> s
"abcdef"
> (string-shrink! s 3)
> s
"abc"

(box obj) procedure
(box? obj) procedure
(unbox box) procedure
(set-box! box obj) procedure

These procedures implement the box data type. A box is a cell containing a single mutable field. The lexical syntax of a box containing the object obj is #&obj (see Section 18.7 [Box syntax], page 190).

The procedure box returns a new box object whose content is initialized to obj. The procedure box? returns #t if obj is a box, and otherwise returns #f. The procedure unbox returns the content of the box box. The procedure set-box! changes the content of the box box to obj. The procedure set-box! returns an unspecified value.

For example:
> (define b (box 0))
> b
#&0
> (define (inc!) (set-box! b (+ (unbox b) 1)))
> (inc!)
> b
#&1
> (unbox b)
1

(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. The lexical syntax of keywords is specified in Section 18.6 [Keyword syntax], page 190.

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. The procedure string->keyword returns the keyword whose name is string.

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

(gensym [prefix]) procedure

This procedure returns a new uninterned symbol. Unintered 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
```

```
(make-uninterned-symbol name [hash]) procedure
(uninterned-symbol? obj) procedure
```

The procedure `make-uninterned-symbol` returns a new uninterned symbol whose name is `name` and hash is `hash`. The name must be a string and the hash must be a nonnegative fixnum.

The procedure `uninterned-symbol?` returns `#t` when `obj` is a symbol that is uninterned and `#f` otherwise.

For example:

```
> (uninterned-symbol? (gensym))
#t
> (make-uninterned-symbol "foo")
#:foo:
> (uninterned-symbol? (make-uninterned-symbol "foo"))
#t
> (uninterned-symbol? 'hello)
#f
> (uninterned-symbol? 123)
#f
```

```
(make-uninterned-keyword name [hash]) procedure
(uninterned-keyword? obj) procedure
```

The procedure `make-uninterned-keyword` returns a new uninterned keyword whose name is `name` and hash is `hash`. The name must be a string and the hash must be a nonnegative fixnum.

The procedure `uninterned-keyword?` returns `#t` when `obj` is a keyword that is uninterned and `#f` otherwise.

For example:

```
> (make-uninterned-keyword "foo")
#:foo:
> (uninterned-keyword? (make-uninterned-keyword "foo"))
#t
> (uninterned-keyword? hello:)
#f
> (uninterned-keyword? 123)
#f
```

```
(void) procedure
```

This procedure returns the void object. The read-eval-print loop prints nothing when the result is the void object.
(eval expr [env])

The first parameter is a datum representing an expression. The eval procedure evaluates this expression in the global interaction environment and returns the result. If present, the second parameter 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

(include file)

The file parameter 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 define-formals) body)

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 (unless test . body)
  '(if ,test #f (begin ,@body)))

(define-macro (push var #!optional val)
  '(set! ,var (cons ,val ,var)))

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 var #!optional val)
  '(set! ,var (cons ,val ,var)))
> (pp (lambda () (push stack 1) (push stack) (push stack 3)))
(lambda ()
  (set! stack (cons 1 stack))
  (set! stack (cons #f stack))
  (set! stack (cons 3 stack)))

(define-syntax name expander)

Define name as a macro special form whose expansion is specified by expander. This form is available only when the runtime option ‘-s’ is used. This option causes
the loading of the `lib/syntax-case` support library, which is the Hieb and Dybvig portable `syntax-case` implementation which has been ported to the Gambit interpreter and compiler. Note that this implementation of `syntax-case` does not support special forms that are specific to Gambit.

For example:

```
$ gsi -:s
Gambit v4.6.1

> (define-syntax unless
   (syntax-rules ()
     ((unless test body ...)
      (if test #f (begin body ...))))))
> (let ((test 111)) (unless (= 1 2) (list test test)))
(111 111)
> (pp (lambda () (let ((test 111)) (unless (= 1 2) (list test test)))))
(lambda () ((lambda (%%test14) (if (= 1 2) #f (%%test14 %test14)) 111)))
> (unless #f (pp xxx))
*** ERROR IN (console)@7.16 -- Unbound variable: xxx
```

(declare declaration...) special form

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

(dialect) Use the given dialect’s semantics. dialect can be: ‘ieee-scheme’, ‘r4rs-scheme’, ‘r5rs-scheme’ or ‘gambit-scheme’.

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

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

([not] inline-primitives primitive...) The given primitives should (or should not) be inlined if possible (all primitives if none specified).

(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 \texttt{(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 \texttt{(block)} declaration). Note that inlining usually causes much less code expansion than specified by the inlining limit (an expansion around 10\% is common for \(n=350\)).

\begin{itemize}
\item \texttt{[not lambda-lift]}
  Lambda-lift (or don’t lambda-lift) locally defined procedures.
\item \texttt{[not constant-fold]}
  Allow (or disallow) constant-folding of primitive procedures.
\item \texttt{[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).
\item \texttt{[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).
\item \texttt{[not run-time-bindings var... ]}
  The given global variables will be tested at run time to see if they are equal to the value defined for them in the runtime system (all variables defined in the runtime if none specified).
\item \texttt{[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 \texttt{char=?} may disregard the type of its arguments in ‘safe’ as well as ‘not safe’ mode.
\item \texttt{[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.
\item \texttt{[not debug]}
  Enable (or disable) the generation of debugging information. The kind of debugging information that is generated depends on the declarations ‘debug-location’, ‘debug-source’, and ‘debug-environments’. If any of the command line options ‘-debug’, ‘-debug-location’, ‘-debug-source’ and ‘-debug-environments’ are present, the ‘debug’ declaration is initially enabled, otherwise it is initially disabled. When all kinds of debugging information are generated there is a substantial increase in the C compilation time and the size of the generated code. When compiling a 3000 line Scheme file it was observed that the total compilation time is 500\% longer and the executable code is 150\% bigger.
\end{itemize}
(not debug-location)
Select (or deselect) source code location debugging information. When this declaration and the 'debug' declaration are in effect, run time error messages indicate the location of the error in the source code file. If any of the command line options '-debug-source' and '-debug-environments' are present and '-debug-location' is absent, the 'debug-location' declaration is initially disabled, otherwise it is initially enabled. When compiling a 3000 line Scheme file it was observed that the total compilation time is 200% longer and the executable code is 60% bigger.

(not debug-source)
Select (or deselect) source code debugging information. When this declaration and the 'debug' declaration are in effect, run time error messages indicate the source code, the backtraces are more precise, and the pp procedure will display the source code of compiled procedures. If any of the command line options '-debug-location' and '-debug-environments' are present and '-debug-source' is absent, the 'debug-source' declaration is initially disabled, otherwise it is initially enabled. When compiling a 3000 line Scheme file it was observed that the total compilation time is 90% longer and the executable code is 90% bigger.

(not debug-environments)
Select (or deselect) environment debugging information. When this declaration and the 'debug' declaration are in effect, the debugger will have access to the environments of the continuations. In other words the local variables defined in compiled procedures (and not optimized away by the compiler) will be shown by the ',e' REPL command. If any of the command line options '-debug-location' and '-debug-source' are present and '-debug-environments' is absent, the 'debug-environments' declaration is initially disabled, otherwise it is initially enabled. When compiling a 3000 line Scheme file it was observed that the total compilation time is 70% longer and the executable code is 40% bigger.

(not proper-tail-calls)
Generate (or don't generate) proper tail calls. When proper tail calls are turned off, tail calls are handled like non-tail calls, that is a continuation frame will be created for all calls regardless of their kind. This is useful for debugging because the caller of a procedure will be visible in the backtrace produced by the REPL's ',b' command even when the call is a tail call. Be advised that this does cause stack space to be consumed for tail calls which may cause the stack to overflow when performing long iterations.
with tail calls (whether they are expressed with a letrec, named let, do, or other form).

([not] optimize-dead-local-variables)
Remove (or preserve) the dead local variables in the environment. Preserving the dead local variables is useful for debugging because continuations will contain the dead variables. Thus, if the code is also compiled with the declaration ‘debug-environments’ the ‘,e’, ‘,ed’, ‘,be’, and ‘,bed’ REPL commands will display the dead variables. On the other hand, preserving the dead local variables may change the space complexity of the program (i.e. some of the data that would normally be reclaimed by the garbage collector will not be). Note that due to other compiler optimizations some dead local variables may be removed regardless of this declaration.

(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’.

(mostly-number-type primitive...)
Numeric arguments and result of the specified primitives are expected to be most often of the given type (all primitives if none specified). mostly-number-type can be: ‘mostly-generic’, ‘mostly-fixnum’, ‘mostly-fixnum-flonum’, ‘mostly-flonum’, or ‘mostly-flonum-fixnum’.

The default declarations used by the compiler are equivalent to:

(declare
 (gambit-scheme)
 (separate)
 (inline)
 (inline-primitives)
 (inlining-limit 350)
 (constant-fold)
 (lambda-lift)
 (not standard-bindings)
 (not extended-bindings)
 (run-time-bindings)
 (safe)
 (interrupts-enabled)
 (not debug) ;; depends on debugging command line options
 (debug-location) ;; depends on debugging command line options
 (debug-source) ;; depends on debugging command line options
 (debug-environments) ;; depends on debugging command line options
 (proper-tail-calls)
 (optimize-dead-local-variables)
 (generic)
 (mostly-fixnum-flonum)
)

These declarations are compatible with the semantics of R5RS Scheme and includes a few procedures from R6RS (mainly fixnum specific and flonum specific procedures). Typically used declarations that enhance performance, at the cost of violating the R5RS Scheme semantics, are: (standard-bindings), (block), (not safe) and (fixnum).

### 6.4 Undocumented extensions

The procedures in this section are not yet documented.

```
(continuation? obj)  procedure
(continuation-capture proc) procedure
(continuation-graft cont proc obj...) procedure
(continuation-return cont obj...) procedure
```

These procedures provide access to internal first-class continuations which are represented using continuation objects distinct from procedures.

The procedure `continuation?` returns `#t` when `obj` is a continuation object and `#f` otherwise.

The procedure `continuation-capture` is similar to the `call/cc` procedure but it represents the continuation with a continuation object. The `proc` parameter must be a procedure accepting a single argument. The procedure `continuation-capture` reifies its continuation and calls `proc` with the corresponding continuation object as its sole argument. Like for `call/cc`, the implicit continuation of the call to `proc` is the implicit continuation of the call to `continuation-capture`.

The procedure `continuation-graft` performs a procedure call to the procedure `proc` with arguments `obj...` and the implicit continuation corresponding to the continuation object `cont`. The current continuation of the call to procedure `continuation-graft` is ignored.

The procedure `continuation-return` invokes the implicit continuation corresponding to the continuation object `cont` with the result(s) `obj...`. This procedure can be easily defined in terms of `continuation-graft`:

```
(define (continuation-return cont . objs)
  (continuation-graft (lambda () (apply values objs))))
```

For example:

```
> (define x #f)
> (define p (make-parameter 11))
> (pp (parameterize ((p 22))
   (cons 33 (continuation-capture
     (lambda (c) (set! x c) 44))))))
(33 . 44)
> x
#<continuation #2>
> (continuation-return x 55)
(33 . 55)
> (continuation-graft x (lambda () (expt 2 10))))
(33 . 1024)
> (continuation-graft x expt 2 10)
(33 . 1024)
> (continuation-graft x (lambda () (p)))
(33 . 22)
```
> (define (map-sqrt1 lst)
  (call/cc
    (lambda (k)
      (map (lambda (x)
            (if (< x 0)
                (k 'error)
                (sqrt x)))
        lst)))))
> (map-sqrt1 '(1 4 9))
(1 2 3)
> (map-sqrt1 '(1 -1 9))
error
> (define (map-sqrt2 lst)
  (continuation-capture
    (lambda (c)
      (map (lambda (x)
            (if (< x 0)
                (continuation-return c 'error)
                (sqrt x)))
        lst)))))
> (map-sqrt2 '(1 4 9))
(1 2 3)
> (map-sqrt2 '(1 -1 9))
error
(procedure display-exception exc [port])
(procedure display-exception-in-context exc cont [port])
(procedure display-procedure-environment proc [port])
(procedure display-continuation-environment cont [port])
(procedure display-continuation-dynamic-environment cont [port])
(procedure display-continuation-backtrace cont [port] [all-frames? display-env? [max-head [max-tail [depth]]]])

The procedure display-continuation-backtrace displays the frames of the continuation corresponding to the continuation object cont on the port port. If it is not specified, port defaults to the current output-port. The frames are displayed in the same format as the REPL’s ‘b’ command.

The parameter all-frames?, which defaults to #f, controls which frames are displayed. Some frames of ancillary importance, such as internal frames created by the interpreter, are not displayed when all-frames? is #f. Otherwise all frames are displayed.

The parameter display-env?, which defaults to #f, controls if the frames are displayed with its environment (the variables accessible and their bindings).

The parameters max-head and max-tail, which default to 10 and 4 respectively, control how many frames are displayed at the head and tail of the continuation.

The parameter depth, which defaults to 0, causes the frame numbers to be offset by that value.

For example:
> (define x #f)
> (define (fib n)
  (if (< n 2)
    (continuation-capture
      (lambda (c) (set! x c) 1)))
(+ (fib (- n 1))
   (fib (- n 2))))

> (fib 10)
89

(display-continuation-backtrace x)
0 fib (console)@7:12 (fib (- n 2))
1 fib (console)@7:12 (fib (- n 2))
2 fib (console)@7:12 (fib (- n 2))
3 fib (console)@7:12 (fib (- n 2))
4 fib (console)@7:12 (fib (- n 2))
5 (interaction) (console)@8:1 (fib 10)
#f

(display-continuation-backtrace x (current-output-port) #t)
0 fib (console)@7:12 (fib (- n 2))
1 fib (console)@6:9 (+ (fib (- n 1)) (fib - ...)
2 fib (console)@7:12 (fib (- n 2))
3 fib (console)@6:9 (+ (fib (- n 1)) (fib - ...)
4 fib (console)@7:12 (fib (- n 2))
5 fib (console)@6:9 (+ (fib (- n 1)) (fib - ...)
6 fib (console)@7:12 (fib (- n 2))
7 fib (console)@6:9 (+ (fib (- n 1)) (fib - ...)
8 fib (console)@7:12 (fib (- n 2))
9 fib (console)@6:9 (+ (fib (- n 1)) (fib - ...
...
13 ##with-no-result-expected-toplevel
14 ##repl-debug
15 ##repl-debug-main
16 ##kernel-handlers
#f

(display-continuation-backtrace x (current-output-port) #f #t)
0 fib (console)@7:12 (fib (- n 2))
n = 2
1 fib (console)@7:12 (fib (- n 2))
n = 4
2 fib (console)@7:12 (fib (- n 2))
n = 6
3 fib (console)@7:12 (fib (- n 2))
n = 8
4 fib (console)@7:12 (fib (- n 2))
n = 10
5 (interaction) (console)@8:1 (fib 10)
#f

(display-continuation-backtrace x (current-output-port) #f #f 2 1 100)
100 fib (console)@7:12 (fib (- n 2))
101 fib (console)@7:12 (fib (- n 2))
...
105 (interaction) (console)@8:1 (fib 10)
#f

(make-thread-group [name [thread-group]]) procedure
(thread-group? obj) procedure
(thread-group-name thread-group) procedure
(thread-group-parent thread-group) procedure
(thread-group-resume! thread-group) procedure
(thread-group-suspend! thread-group) procedure
(thread-group-terminate! thread-group) procedure
(thread-group->thread-group-list thread-group) procedure
(thread-group->thread-group-vector thread-group)  procedure
(thread-group->thread-list thread-group)   procedure
(thread-group->thread-vector thread-group)  procedure

(thread-state thread)                     procedure
(thread-state-uninitialized? thread-state) procedure
(thread-state-initialized? thread-state)   procedure
(thread-state-active? thread-state)       procedure
(thread-state-active-waiting-for thread-state) procedure
(thread-state-normally-terminated? thread-state) procedure
(thread-state-normally-terminated-result thread-state) procedure
(thread-state-abnormally-terminated? thread-state) procedure
(thread-state-abnormally-terminated-reason thread-state) procedure

top [thread-group [port]])               procedure
(thread-interrupt! thread [thunk])        procedure
(thread-suspend! thread)                   procedure
(thread-resume! thread)                    procedure

(thread-thread-group thread)              procedure
(define-type-of-thread name field...)    special form

(thread-init! thread thunk [name [thread-group]]) procedure
(initialized-thread-exception? obj)        procedure
(initialized-thread-exception-procedure exc) procedure
(initialized-thread-exception-arguments exc) procedure

(uninitialized-thread-exception? obj)      procedure
(uninitialized-thread-exception-procedure exc) procedure
(uninitialized-thread-exception-arguments exc) procedure

(inactive-thread-exception? obj)          procedure
(inactive-thread-exception-procedure exc)  procedure
(inactive-thread-exception-arguments exc)  procedure

(rpc-remote-error-exception? obj)         procedure
(rpc-remote-error-exception-procedure exc) procedure
(rpc-remote-error-exception-arguments exc) procedure
(rpc-remote-error-exception-message exc)  procedure

timeout->time timeout)                    procedure

(open-dummy)                              procedure
(port-settings-set! port settings)        procedure
(input-port-bytes-buffered port)          procedure
(input-port-characters-buffered port)      procedure

(nonempty-input-port-character-buffer-exception? obj) procedure
(nonempty-input-port-character-buffer-exception-arguments exc)
(procedure
  (nonempty-input-port-character-buffer-exception-procedure exc)
(procedure
  (repl-input-port)
(procedure
  (repl-output-port)
(procedure
  (console-port)
(procedure
  (current-user-interrupt-handler [handler])
(procedure
  (defer-user-interrupts)
(procedure
  (primordial-exception-handler exc)
(procedure
  (err-code->string code)
(procedure
  (foreign? obj)
(procedure
  (foreign-tags foreign)
(procedure
  (foreign-address foreign)
(procedure
  (foreign-release! foreign)
(procedure
  (foreign-released? foreign)
(procedure
  (invalid-hash-number-exception? obj)
(procedure
  (invalid-hash-number-exception-procedure exc)
(procedure
  (invalid-hash-number-exception-arguments exc)
(procedure
  (tcp-client-peer-socket-info tcp-client-port)
(procedure
  (tcp-client-self-socket-info tcp-client-port)
(procedure
  (tcp-server-socket-info tcp-server-port)
(procedure
  (socket-info? obj)
(procedure
  (socket-info-address socket-info)
(procedure
  (socket-info-family socket-info)
(procedure
  (socket-info-port-number socket-info)
(procedure
  (system-version)
(procedure
  (system-version-string)
(procedure
  (system-type)
(procedure
  (system-type-string)
(procedure
  (configure-command-string)
(procedure
  (system-stamp)
(procedure
  (future expr)
(special form
  (touch obj)
(procedure
  (tty? obj)
(procedure
  (tty-history tty)
(procedure
  (tty-history-set! tty history)
(procedure
  (tty-history-max-length-set! tty n)
(procedure
  (tty-paren-balance-duration-set! tty duration)
(procedure
  (tty-text-attributes-set! tty attributes)
(procedure
  (tty-mode-set! tty mode)
(tty-type-set! tty type) procedure
(with-input-from-port port thunk) procedure
(with-output-to-port port thunk) procedure
(input-port-char-position port) procedure
(output-port-char-position port) procedure
(open-event-queue n) procedure
(main ...) procedure
(define-record-type ...) special form
(define-type ...) special form
(namespace ...) special form
(this-source-file) special form
(receive ...) special form
(cond-expand ...) special form
(define-cond-expand-feature ident) special form
(finite? x) procedure
(infinite? x) procedure
(nan? x) procedure
(six.! x) undefined
(six.!x x) special form
(six.&x x) special form
(six.*x x) special form
(six.+x x) special form
(six.+x x) special form
(six.--x x) special form
(six.-x x) special form
(six.arrow expr ident) special form
(six.break) undefined
(six.call func arg...) special form
(six.case) undefined
(six.clause) undefined
(six.compound statement...) special form
(six.cons x y) special form
(six.continue) undefined
(six.define-procedure ident proc) special form
(six.define-variable ident type dims init) special form
(six.do-while stat expr) special form
(six.dot expr ident) special form
(six.for stat1 expr2 expr3 stat2) special form
(six.goto) undefined
(six.identifier ident) special form
(six.if expr stat1 [stat2]) special form
(six.index expr1 expr2) special form
(six.label)
(six.list x y)
(six.literal value)
(six.make-array init dim...)
(six.new ident arg...)
(six.null)
(six.prefix datum)
(six.procedure type params stat)
(six.procedure-body stat...)
(six.return)
(six.switch)
(six.while expr stat...)
(six.x!=y x y)
(six.x%=y x y)
(six.x%y x y)
(six.x&y x y)
(six.x|=y x y)
(six.x&y x y)
(six.x*=y x y)
(six.x*y x y)
(six.x+=y x y)
(six.x+y x y)
(six.x-- x)
(six.x-=y x y)
(six.x-y x y)
(six.x/=y x y)
(six.x/y x y)
(six.x:-y x y)
(six.x:=y x y)
(six.x:=y x y)
(six.x<<=y x y)
(six.x<<y x y)
(six.x>y x y)
(six.x?y:z x y z)
(six.x^=y x y)
(six.x^y x y)
(|six.x\|=y| x y)
(\textit{six}.\textit{x}\textbar\|y\textbar x y)
special form

(\textit{six}.\textit{x}\textbar \|\|y\textbar x y)
special form

(\textit{six}.\textit{\~{}}x\textit{x})
special form
7 Namespaces

TO DO!
8 Characters and strings

Gambit supports the Unicode character encoding standard. Scheme characters can be any of the characters whose Unicode encoding is in the range 0 to #x10ffff (inclusive) but not in the range #xd800 to #xdbff. Source code can also contain any Unicode character, 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. ASCII, ISO-8859-1, UTF-8, etc). This can be done by specifying the runtime option ‘-f’ when gsi and gsc are started.

8.1 Extensions to character procedures

\begin{itemize}
\item \texttt{(char->integer char)} \hspace{2cm} procedure
\item \texttt{(integer->char n)} \hspace{2cm} procedure
\end{itemize}

The procedure \texttt{char->integer} returns the Unicode encoding of the character \texttt{char}.

The procedure \texttt{integer->char} returns the character whose Unicode encoding is the exact integer \texttt{n}.

For example:
\begin{verbatim}
> (char->integer #\!)
33
> (integer->char 65)
#\A
> (integer->char (char->integer #\u1234))
#\u1234
> (integer->char #xd800)
*** ERROR IN (console)@4.1 -- (Argument 1) Out of range (integer->char 55296)
\end{verbatim}

\begin{itemize}
\item \texttt{(char=? char1...)} \hspace{2cm} procedure
\item \texttt{(char<? char1...)} \hspace{2cm} procedure
\item \texttt{(char>? char1...)} \hspace{2cm} procedure
\item \texttt{(char<=? char1...)} \hspace{2cm} procedure
\item \texttt{(char>=? char1...)} \hspace{2cm} procedure
\item \texttt{(char-ci=? char1...)} \hspace{2cm} procedure
\item \texttt{(char-ci<? char1...)} \hspace{2cm} procedure
\item \texttt{(char-ci>? char1...)} \hspace{2cm} procedure
\item \texttt{(char-ci<=? char1...)} \hspace{2cm} procedure
\item \texttt{(char-ci>=? char1...)} \hspace{2cm} procedure
\end{itemize}

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 characters \texttt{lst} is sorted in nondecreasing order can be done with the call \texttt{(apply char<? lst)}.

8.2 Extensions to string procedures

\begin{itemize}
\item \texttt{(string=? string1...)} \hspace{2cm} procedure
\item \texttt{(string<? string1...)} \hspace{2cm} procedure
\item \texttt{(string>? string1...)} \hspace{2cm} procedure
\item \texttt{(string<=? string1...)} \hspace{2cm} procedure
\end{itemize}
(string>=? string1...) procedure
(string-ci=? string1...) procedure
(string-ci<? string1...) procedure
(string-ci>? string1...) procedure
(string-ci<=? string1...) procedure
(string-ci>=? string1...) procedure

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

9.1 Extensions to numeric procedures

\[
(= z1 \ldots) \quad \text{procedure}
\]

\[
(< x1 \ldots) \quad \text{procedure}
\]

\[
(> x1 \ldots) \quad \text{procedure}
\]

\[
(\leq x1 \ldots) \quad \text{procedure}
\]

\[
(\geq x1 \ldots) \quad \text{procedure}
\]

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 \( \text{lst} \) is sorted in nondecreasing order can be done with the call \((\text{apply} < \text{lst})\).

9.2 IEEE floating point arithmetic

To better conform to IEEE floating point arithmetic the standard numeric tower is extended with these special inexact reals:

\[+\text{inf.0}\] positive infinity

\[-\text{inf.0}\] negative infinity

\[+\text{nan.0}\] “not a number”

\[-0.\] negative zero (‘0.’ is the positive zero)

The infinities and “not a number” are reals (i.e. \((\text{real?} +\text{inf.0})\) is \#t) but are not rational (i.e. \((\text{rational?} +\text{inf.0})\) is \#f).

Both zeros are numerically equal (i.e. \((= -0.\ 0.)\) is \#t) but are not equivalent (i.e. \((\text{eqv?} -0.\ 0.)\) and \((\text{equal?} -0.\ 0.)\) are \#f). All numerical comparisons with “not a number”, including \((= +\text{nan.0} +\text{nan.0})\), are \#f.

9.3 Integer square root and nth root

\[(\text{integer-sqrt } n)\] procedure

This procedure returns the integer part of the square root of the nonnegative exact integer \(n\).

For example:

\[
> \ (\text{integer-sqrt } 123)
\]

\[11\]

\[(\text{integer-nth-root } n1 n2)\] procedure

This procedure returns the integer part of \(n1\) raised to the power \(1/n2\), where \(n1\) is a nonnegative exact integer and \(n2\) is a positive exact integer.

For example:

\[
> \ (\text{integer-nth-root } 100 3)
\]

\[4\]
9.4 Bitwise-operations on exact integers

The procedures defined in this section are compatible with the withdrawn “Integer Bitwise-operation Library SRFI” (SRFI 33). Note that some of the procedures specified in SRFI 33 are not provided.

Most procedures in this section are specified in terms of the binary representation of exact integers. The two’s complement representation is assumed where an integer is composed of an infinite number of bits. The upper section of an integer (the most significant bits) are either an infinite sequence of ones when the integer is negative, or they are an infinite sequence of zeros when the integer is nonnegative.

**(arithmetic-shift n1 n2)**  
This procedure returns \( n1 \) shifted to the left by \( n2 \) bits, that is \( \left\lfloor \frac{n1 \times 2^{n2}}{1} \right\rfloor \). Both \( n1 \) and \( n2 \) must be exact integers.

For example:

\[
\begin{align*}
&> (\text{arithmetic-shift} 1000 7) \quad ; n1=...0000001111101000 \\
&\quad 128000 \\
&> (\text{arithmetic-shift} 1000 -6) \quad ; n1=...0000001111101000 \\
&\quad 15 \\
&> (\text{arithmetic-shift} -23 -3) \quad ; n1=...1111111111101001 \\
&\quad -3
\end{align*}
\]

**(bitwise-merge n1 n2 n3)**  
This procedure returns an exact integer whose bits combine the bits from \( n2 \) and \( n3 \) depending on \( n1 \). The bit at index \( i \) of the result depends only on the bits at index \( i \) in \( n1, n2 \) and \( n3 \): it is equal to the bit in \( n2 \) when the bit in \( n1 \) is 0 and it is equal to the bit in \( n3 \) when the bit in \( n1 \) is 1. All arguments must be exact integers.

For example:

\[
\begin{align*}
&> (\text{bitwise-merge} -4 -11 10) \quad ; ...11111100 ...11110101 ...00001010 \\
&\quad 9 \\
&> (\text{bitwise-merge} 12 -11 10) \quad ; ...00001100 ...11110101 ...00001010 \\
&\quad -7
\end{align*}
\]

**(bitwise-and n...)**  
This procedure returns the bitwise “and” of the exact integers \( n... \). The value -1 is returned when there are no arguments.

For example:

\[
\begin{align*}
&> (\text{bitwise-and} 6 12) \quad ; ...00000110 ...00001100 \\
&\quad 4 \\
&> (\text{bitwise-and} 6 -4) \quad ; ...00000110 ...11111100 \\
&\quad 4 \\
&> (\text{bitwise-and} -6 -4) \quad ; ...11111010 ...11111100 \\
&\quad -8 \\
&> (\text{bitwise-and}) \\
&\quad -1
\end{align*}
\]

**(bitwise-ior n...)**  
This procedure returns the bitwise “inclusive-or” of the exact integers \( n... \). The value 0 is returned when there are no arguments.

For example:
> (bitwise-ior 6 12) ; ...00000110 ...00001100
14
> (bitwise-ior 6 -4) ; ...00000110 ...11111100
-2
> (bitwise-ior -6 -4) ; ...11110110 ...11111100
-2
> (bitwise-ior)
0

(bitwise-xor n...) procedure
This procedure returns the bitwise “exclusive-or” of the exact integers n.... The value 0 is returned when there are no arguments.

For example:
> (bitwise-xor 6 12) ; ...00000110 ...00001100
10
> (bitwise-xor 6 -4) ; ...00000110 ...11111100
-6
> (bitwise-xor -6 -4) ; ...11110110 ...11111100
6
> (bitwise-xor)
0

(bitwise-not n) procedure
This procedure returns the bitwise complement of the exact integer n.

For example:
> (bitwise-not 3) ; ...00000011
-4
> (bitwise-not -1) ; ...11111111
0

(bit-count n) procedure
This procedure returns the bit count of the exact integer n. If n is nonnegative, the bit count is the number of 1 bits in the two’s complement representation of n. If n is negative, the bit count is the number of 0 bits in the two’s complement representation of n.

For example:
> (bit-count 0) ; ...00000000
0
> (bit-count 1) ; ...00000001
1
> (bit-count 2) ; ...00000010
1
> (bit-count 3) ; ...00000011
2
> (bit-count 4) ; ...00000100
1
> (bit-count -23) ; ...11101011
3

(integer-length n) procedure
This procedure returns the bit length of the exact integer n. If n is a positive integer the bit length is one more than the index of the highest 1 bit (the least significant bit is at index 0). If n is a negative integer the bit length is one more than the index of the highest 0 bit. If n is zero, the bit length is 0.
For example:

```scheme
> (integer-length 0) ; ...00000000
0
> (integer-length 1) ; ...00000001
1
> (integer-length 2) ; ...00000010
2
> (integer-length 3) ; ...00000011
2
> (integer-length 4) ; ...00000100
3
> (integer-length -23) ; ...11101001
5
```

(bit-set? n1 n2)

This procedure returns a boolean indicating if the bit at index n1 of n2 is set (i.e. equal to 1) or not. Both n1 and n2 must be exact integers, and n1 must be nonnegative.

For example:

```scheme
> (map (lambda (i) (bit-set? i -23)) '(7 6 5 4 3 2 1 0))
'(t t t f t f f t)
```

(any-bits-set? n1 n2)

This procedure returns a boolean indicating if the bitwise and of n1 and n2 is different from zero or not. This procedure is implemented more efficiently than the naive definition:

```scheme
(define (any-bits-set? n1 n2) (not (zero? (bitwise-and n1 n2))))
```

For example:

```scheme
> (any-bits-set? 5 10) ; ...00000101 ...00001010
#f
> (any-bits-set? -23 32) ; ...11101001 ...00100000
#t
```

(all-bits-set? n1 n2)

This procedure returns a boolean indicating if the bitwise and of n1 and n2 is equal to n1 or not. This procedure is implemented more efficiently than the naive definition:

```scheme
(define (all-bits-set? n1 n2) (= n1 (bitwise-and n1 n2)))
```

For example:

```scheme
> (all-bits-set? 1 3) ; ...00000001 ...00000011
#t
> (all-bits-set? 7 3) ; ...00000111 ...00000011
#f
```

(first-bit-set n)

This procedure returns the bit index of the least significant bit of n equal to 1 (which is also the number of 0 bits that are below the least significant 1 bit). This procedure returns -1 when n is zero.

For example:

```scheme
> (first-bit-set 24) ; ...00011000
3
> (first-bit-set 0) ; ...00000000
-1
```
These procedures operate on a bit-field which is \( n1 \) bits wide starting at bit index \( n2 \). All arguments must be exact integers and \( n1 \) and \( n2 \) must be nonnegative.

The procedure `extract-bit-field` returns the bit-field of \( n3 \) shifted to the right so that the least significant bit of the bit-field is the least significant bit of the result.

The procedure `test-bit-field?` returns \#t if any bit in the bit-field of \( n3 \) is equal to 1, otherwise \#f is returned.

The procedure `clear-bit-field` returns \( n3 \) with all bits in the bit-field replaced with 0.

The procedure `replace-bit-field` returns \( n4 \) with the bit-field replaced with the least-significant \( n1 \) bits of \( n3 \).

The procedure `copy-bit-field` returns \( n4 \) with the bit-field replaced with the (same index and size) bit-field in \( n3 \).

For example:

\[
\begin{align*}
> (extract-bit-field 5 2 -37) & \quad ; \ldots11011011 \\
& \quad 22 \\
> (test-bit-field? 5 2 -37) & \quad ; \ldots11011011 \\
& \quad \#t \\
> (test-bit-field? 1 2 -37) & \quad ; \ldots11011011 \\
& \quad \#f \\
> (clear-bit-field 5 2 -37) & \quad ; \ldots11011011 \\
& \quad -125 \\
> (replace-bit-field 5 2 -6 -37) & \quad ; \ldots1111010 \ldots11011011 \\
& \quad -21 \\
> (copy-bit-field 5 2 -6 -37) & \quad ; \ldots1111010 \ldots11011011 \\
& \quad -5
\end{align*}
\]

### 9.5 Fixnum specific operations

\[
\begin{align*}
(fixnum? \ obj) & \quad \text{procedure} \\
(fx* \ n1\ldots) & \quad \text{procedure} \\
(fx+ \ n1\ldots) & \quad \text{procedure} \\
(fx- \ n1\ldots) & \quad \text{procedure} \\
(fx< \ n1\ldots) & \quad \text{procedure} \\
(fx<= \ n1\ldots) & \quad \text{procedure} \\
(fx= \ n1\ldots) & \quad \text{procedure} \\
(fx> \ n1\ldots) & \quad \text{procedure} \\
(fx>\ = \ n1\ldots) & \quad \text{procedure} \\
(fxabs \ n) & \quad \text{procedure} \\
(fxand \ n1\ldots) & \quad \text{procedure}
\end{align*}
\]
(fxarithmetic-shift n1 n2) procedure
(fxarithmetic-shift-left n1 n2) procedure
(fxarithmetic-shift-right n1 n2) procedure
(fxbit-count n) procedure
(fxbit-set? n1 n2) procedure
(fxeven? n) procedure
(fxfirst-bit-set n) procedure
(fxif n1 n2 n3) procedure
(fxior n1...) procedure
(fxlength n) procedure
(fxmax n1 n2...) procedure
(fxmin n1 n2...) procedure
(fxmodulo n1 n2) procedure
(fxnegative? n) procedure
(fxnot n) procedure
(fxodd? n) procedure
(fxpositive? n) procedure
(fxquotient n1 n2) procedure
(fxremainder n1 n2) procedure
(fxwrap* n1...) procedure
(fxwrap+ n1...) procedure
(fxwrap- n1 n2...) procedure
(fxwrapabs n) procedure
(fxwraparithmetic-shift n1 n2) procedure
(fxwraparithmetic-shift-left n1 n2) procedure
(fxwraparithmetic-shift-right n1 n2) procedure
(fxwrapquotient n1 n2) procedure
(fxxor n1...) procedure
(fxzero? n) procedure
(fixnum-overflow-exception? obj) procedure
(fixnum-overflow-exception-procedure exc) procedure
(fixnum-overflow-exception-arguments exc) procedure

Fixnum-overflow-exception objects are raised by some of the fixnum specific procedures when the result is larger than can fit in a fixnum. The parameter exc must be a fixnum-overflow-exception object.
The procedure `fixnum-overflow-exception?` returns `#t` when `obj` is a fixnum-overflow-exception object and `#f` otherwise.

The procedure `fixnum-overflow-exception-procedure` returns the procedure that raised `exc`.

The procedure `fixnum-overflow-exception-arguments` returns the list of arguments of the procedure that raised `exc`.

For example:

```scheme
> (define (handler exc)
    (if (fixnum-overflow-exception? exc)
        (list (fixnum-overflow-exception-procedure exc)
              (fixnum-overflow-exception-arguments exc))
        'not-fixnum-overflow-exception))
> (with-exception-catcher
    handler
    (lambda () (fx* 100000 100000)))
(#<procedure #2 fx*> (100000 100000))
```

### 9.6 Flonum specific operations

```scheme
(flonum? obj) procedure
(fixnum->flonum n) procedure
(fl* x1...) procedure
(fl+ x1...) procedure
(fl- x1 x2...) procedure
(fl/ x1 x2) procedure
(fl< x1...) procedure
(fl<= x1...) procedure
(fl= x1...) procedure
(fl> x1...) procedure
(fl>= x1...) procedure
(flabs x) procedure
(flacos x) procedure
(flasin x) procedure
(flatan x) procedure
```
(flatan y x) procedure
(flceiling x) procedure
(flcos x) procedure
(fldenominator x) procedure
(fleven? x) procedure
(flexp x) procedure
(flexpt x y) procedure
(flfinite? x) procedure
(flfloor x) procedure
(flinfinite? x) procedure
(flinteger? x) procedure
(fllog x) procedure
(flmax x1 x2...) procedure
(flmin x1 x2...) procedure
(flnan? x) procedure
(flnegative? x) procedure
(flnumerator x) procedure
(flodd? x) procedure
(flpositive? x) procedure
(flround x) procedure
(flsin x) procedure
(flsqrt x) procedure
(fltan x) procedure
(fltruncate x) procedure
(flzero? x) procedure

9.7 Pseudo random numbers

The procedures and variables defined in this section are compatible with the “Sources of Random Bits SRFI” (SRFI 27). The implementation is based on Pierre L’Ecuyer’s MRG32k3a pseudo random number generator. At the heart of SRFI 27’s interface is the random source type which encapsulates the state of a pseudo random number generator. The state of a random source object changes every time a pseudo random number is generated from this random source object.

default-random-source variable

The global variable default-random-source is bound to the random source object which is used by the random-integer, random-real, random-u8vector and random-f64vector procedures.
This procedure returns a pseudo random exact integer in the range 0 to n-1. The random source object in the global variable default-random-source is used to generate this number. The parameter n must be a positive exact integer.

For example:

```
> (random-integer 100)
24
> (random-integer 100)
2
> (random-integer 10000000000000000000000000000000000000000)
61436302709022284438072426748425263488507
```

This procedure returns a pseudo random inexact real between, but not including, 0 and 1. The random source object in the global variable default-random-source is used to generate this number.

For example:

```
> (random-real)
.24230672079133753
> (random-real)
.02317001922506932
```

This procedure returns a u8vector of length n containing pseudo random exact integers in the range 0 to 255. The random source object in the global variable default-random-source is used to generate these numbers. The parameter n must be a nonnegative exact integer.

For example:

```
> (random-u8vector 10)
#u8(200 53 29 202 3 85 208 187 73 219)
```

This procedure returns a f64vector of length n containing pseudo random inexact reals between, but not including, 0 and 1. The random source object in the global variable default-random-source is used to generate these numbers. The parameter n must be a nonnegative exact integer.

For example:

```
> (random-f64vector 3)
#f64(.7145854494613069 .47089632669147946 .5400124875182746)
```

This procedure returns a new random source object initialized to a predetermined state (to initialize to a pseudo random state the procedure random-source-randomize! should be called).

For example:

```
> (define rs (make-random-source))
> ((random-source-make-integers rs) 10000000)
8583952
```
(random-source? obj)      procedure
This procedure returns #t when obj is a random source object and #f otherwise.

For example:
> (random-source? default-random-source)
#t
> (random-source? 123)
#f

(random-source-state-ref random-source) procedure
(random-source-state-set! random-source state) procedure
The procedure random-source-state-ref extracts the state of the random source object random-source and returns a vector containing the state.

The procedure random-source-state-set! restores the state of the random source object random-source to state which must be a vector returned from a call to the procedure random-source-state-ref.

For example:
> (define s (random-source-state-ref default-random-source))
> (random-integer 10000000000000000000000000000000000000000)
758388018890342336152609321
> (random-source-state-set! default-random-source s)
> (random-integer 10000000000000000000000000000000000000000)
758388018890342336152609321

(random-source-randomize! random-source) procedure
(random-source-pseudo-randomize! random-source i j) procedure
These procedures change the state of the random source object random-source. The procedure random-source-randomize! sets the random source object to a state that depends on the current time (which for typical uses can be considered to randomly initialize the state). The procedure random-source-pseudo-randomize! sets the random source object to a state that is determined only by the current state and the nonnegative exact integers i and j. For both procedures the value returned is unspecified.

For example:
> (define s (random-source-state-ref default-random-source))
> (random-source-pseudo-randomize! default-random-source 5 99)
9816755163910623041601722050112674079767
> (random-source-state-set! default-random-source s)
> (random-source-randomize! default-random-source 5 99)
9816755163910623041601722050112674079767
> (random-source-pseudo-randomize! default-random-source 5 99)
9816755163910623041601722050112674079767
> (random-source-state-set! default-random-source s)
> (random-source-randomize! default-random-source)
2271441220851943333844931436877678110622
> (random-source-state-set! default-random-source s)
> (random-source-randomize! default-random-source)
> (random-integer 10000000000000000000000000000000000000000)
6247966138948323029033944059178072366895
Chapter 9: Numbers

(random-source-make-integers random-source)  procedure
This procedure returns a procedure for generating pseudo random exact integers using
the random source object random-source. The returned procedure accepts a single
parameter n, a positive exact integer, and returns a pseudo random exact integer in
the range 0 to n-1.
For example:
> (define rs (make-random-source))
> (define ri (random-source-make-integers rs))
> (ri 10000000)
8583952
> (ri 10000000)
2879793

(random-source-make-reals random-source [precision])  procedure
This procedure returns a procedure for generating pseudo random inexact reals using
the random source object random-source. The returned procedure accepts no param-
eters and returns a pseudo random inexact real between, but not including, 0 and 1.
The optional parameter precision specifies an upper bound on the minimum amount
by which two generated pseudo-random numbers can be separated.
For example:
> (define rs (make-random-source))
> (define rr (random-source-make-reals rs))
> (rr)
.857402537562821
> (rr)
.2876463473845367

(random-source-make-u8vectors random-source)  procedure
This procedure returns a procedure for generating pseudo random u8vectors using the
random source object random-source. The returned procedure accepts a single pa-
rameter n, a nonnegative exact integer, and returns a u8vector of length n containing
pseudo random exact integers in the range 0 to 255.
For example:
> (define rs (make-random-source))
> (define rv (random-source-make-u8vectors rs))
> (rv 10)
#u8(200 53 29 202 3 85 208 187 73 219)
> (rv 10)
#u8(113 8 182 120 138 103 53 192 40 176)

(random-source-make-f64vectors random-source [precision])  procedure
This procedure returns a procedure for generating pseudo random f64vectors using
the random source object random-source. The returned procedure accepts a sin-
gle parameter n, a nonnegative exact integer, and returns an f64vector of length n containing
pseudo random inexact reals between, but not including, 0 and 1. The
optional parameter precision specifies an upper bound on the minimum amount by
which two generated pseudo-random numbers can be separated.
For example:
> (define rs (make-random-source))
> (define rv (random-source-make-f64vectors rs))
\begin{verbatim}
> (rv 3)
f64(.7342236104231586 .2876463473845367 .8574025375628211)
> (rv 3)
f64(.013863292728449427 .33449296573515447 .8162050798467028)
\end{verbatim}
10 Homogeneous vectors

Homogeneous vectors are vectors containing raw numbers of the same type (signed or unsigned exact integers or inexact reals). There are 10 types of homogeneous vectors: ‘s8vector’ (vector of exact integers in the range \(-2^{37} \text{ to } 2^{37}-1\)), ‘u8vector’ (vector of exact integers in the range \(0 \text{ to } 2^{27}-1\)), ‘s16vector’ (vector of exact integers in the range \(-2^{15} \text{ to } 2^{15}-1\)), ‘u16vector’ (vector of exact integers in the range \(0 \text{ to } 2^{16}-1\)), ‘s32vector’ (vector of exact integers in the range \(-2^{31} \text{ to } 2^{31}-1\)), ‘u32vector’ (vector of exact integers in the range \(0 \text{ to } 2^{32}-1\)), ‘s64vector’ (vector of exact integers in the range \(-2^{63} \text{ to } 2^{63}-1\)), ‘u64vector’ (vector of exact integers in the range \(0 \text{ to } 2^{64}-1\)), ‘f32vector’ (vector of 32 bit floating point numbers), and ‘f64vector’ (vector of 64 bit floating point numbers).

The lexical syntax of homogeneous vectors is specified in Section 18.9 [Homogeneous vector syntax], page 191.

The procedures available for homogeneous vectors, listed below, are the analog of the normal vector/string procedures for each of the homogeneous vector types.

```
(s8vector? obj)  procedure
(make-s8vector k [fill])  procedure
(s8vector exact-int8...)  procedure
(s8vector-length s8vector)  procedure
(s8vector-ref s8vector k)  procedure
(s8vector-set! s8vector k exact-int8)  procedure
(s8vector->list s8vector)  procedure
(list->s8vector list-of-exact-int8)  procedure
(s8vector-fill! s8vector fill)  procedure
(subs8vector-fill! vector start end fill)  procedure
(append-s8vectors lst)  procedure
(s8vector-copy s8vector)  procedure
(s8vector-append s8vector...)  procedure
(subs8vector s8vector start end)  procedure
(subs8vector-move! src-s8vector src-start src-end dst-s8vector dst-start)  procedure
(s8vector-shrink! s8vector k)  procedure

(u8vector? obj)  procedure
(make-u8vector k [fill])  procedure
(u8vector exact-int8...)  procedure
(u8vector-length u8vector)  procedure
(u8vector-ref u8vector k)  procedure
(u8vector-set! u8vector k exact-int8)  procedure
(u8vector->list u8vector)  procedure
(list->u8vector list-of-exact-int8)  procedure
(u8vector-fill! u8vector fill)  procedure
(subu8vector-fill! vector start end fill)  procedure
(append-u8vectors lst)  procedure
(u8vector-copy u8vector)  procedure
```
Chapter 10: Homogeneous vectors

(u8vector-append u8vector...)
(procedure)
(subu8vector u8vector start end)
(procedure)
(subu8vector-move! src-u8vector src-start src-end dst-u8vector
   dst-start)
(u8vector-shrink! u8vector k)
(procedure)

(s16vector? obj)
(procedure)
(make-s16vector k [fill])
(procedure)
(s16vector exact-int16...)
(procedure)
(s16vector-length s16vector)
(procedure)
(s16vector-ref s16vector k)
(procedure)
(s16vector-set! s16vector k exact-int16)
(procedure)
(s16vector->list s16vector)
(procedure)
(list->s16vector list-of-exact-int16)
(procedure)
(s16vector-fill! s16vector fill)
(procedure)
(subs16vector-fill! vector start end fill)
(procedure)
(append-s16vectors lst)
(procedure)
(s16vector-copy s16vector)
(procedure)
(s16vector-append s16vector...)
(procedure)
(subs16vector s16vector start end)
(procedure)
(subs16vector-move! src-s16vector src-start src-end dst-s16vector
   dst-start)
(s16vector-shrink! s16vector k)
(procedure)

(u16vector? obj)
(procedure)
(make-u16vector k [fill])
(procedure)
(u16vector exact-int16...)
(procedure)
(u16vector-length u16vector)
(procedure)
(u16vector-ref u16vector k)
(procedure)
(u16vector-set! u16vector k exact-int16)
(procedure)
(u16vector->list u16vector)
(procedure)
(list->u16vector list-of-exact-int16)
(procedure)
(u16vector-fill! u16vector fill)
(procedure)
(subu16vector-fill! vector start end fill)
(procedure)
(append-u16vectors lst)
(procedure)
(u16vector-copy u16vector)
(procedure)
(u16vector-append u16vector...)
(procedure)
(subu16vector u16vector start end)
(procedure)
(subu16vector-move! src-u16vector src-start src-end dst-u16vector
   dst-start)
(u16vector-shrink! u16vector k)
(procedure)

(s32vector? obj)
(procedure)
(make-s32vector k [fill])
(procedure)
(s32vector exact-int32...)
(procedure)
(s32vector-length s32vector)
(procedure)
(s32vector-ref s32vector k)
(procedure)
(s32vector-set! s32vector k exact-int32)
(procedure)
(s32vector->list s32vector)
(procedure)
(list->s32vector list-of-exact-int32)  procedure
(s32vector-fill! s32vector fill)  procedure
(subs32vector-fill! vector start end fill)  procedure
(append-s32vectors lst)  procedure
(s32vector-copy s32vector)  procedure
(s32vector-append s32vector...)  procedure
(subs32vector s32vector start end)  procedure
(subs32vector-move! src-s32vector src-start src-end dst-s32vector dst-start)  procedure
(s32vector-shrink! s32vector k)  procedure
(u32vector? obj)  procedure
(make-u32vector k [fill])  procedure
(u32vector exact-int32...)  procedure
(u32vector-length u32vector)  procedure
(u32vector-ref u32vector k)  procedure
(u32vector-set! u32vector k exact-int32)  procedure
(u32vector->list u32vector)  procedure
(list->u32vector list-of-exact-int32)  procedure
(u32vector-fill! u32vector fill)  procedure
(subu32vector-fill! vector start end fill)  procedure
(append-u32vectors lst)  procedure
(u32vector-copy u32vector)  procedure
(u32vector-append u32vector...)  procedure
(subu32vector u32vector start end)  procedure
(subu32vector-move! src-u32vector src-start src-end dst-u32vector dst-start)  procedure
(u32vector-shrink! u32vector k)  procedure
(s64vector? obj)  procedure
(make-s64vector k [fill])  procedure
(s64vector exact-int64...)  procedure
(s64vector-length s64vector)  procedure
(s64vector-ref s64vector k)  procedure
(s64vector-set! s64vector k exact-int64)  procedure
(s64vector->list s64vector)  procedure
(list->s64vector list-of-exact-int64)  procedure
(s64vector-fill! s64vector fill)  procedure
(subs64vector-fill! vector start end fill)  procedure
(append-s64vectors lst)  procedure
(s64vector-copy s64vector)  procedure
(s64vector-append s64vector...)  procedure
(subs64vector s64vector start end)  procedure
(subs64vector-move! src-s64vector src-start src-end dst-s64vector dst-start)  procedure
(s64vector-shrink! s64vector k)  procedure
(u64vector? obj)  procedure
(make-u64vector k [fill])  procedure
(u64vector exact-int64...) procedure
(u64vector-length u64vector) procedure
(u64vector-ref u64vector k) procedure
(u64vector-set! u64vector k exact-int64) procedure
(u64vector->list u64vector) procedure
(list->u64vector list-of-exact-int64) procedure
(u64vector-fill! u64vector fill) procedure
(subu64vector-fill! vector start end fill) procedure
(append-u64vectors lst) procedure
(u64vector-copy u64vector) procedure
(u64vector-append u64vector...) procedure
(subu64vector u64vector start end) procedure
(subu64vector-move! src-u64vector src-start src-end dst-u64vector dst-start) procedure
(u64vector-shrink! u64vector k) procedure

(f32vector? obj) procedure
(make-f32vector k [fill]) procedure
(f32vector inexact-real...) procedure
(f32vector-length f32vector) procedure
(f32vector-ref f32vector k) procedure
(f32vector-set! f32vector k inexact-real) procedure
(f32vector->list f32vector) procedure
(list->f32vector list-of-inexact-real) procedure
(f32vector-fill! f32vector fill) procedure
(subf32vector-fill! vector start end fill) procedure
(append-f32vectors lst) procedure
(f32vector-copy f32vector) procedure
(f32vector-append f32vector...) procedure
(subf32vector f32vector start end) procedure
(subf32vector-move! src-f32vector src-start src-end dst-f32vector dst-start) procedure
(f32vector-shrink! f32vector k) procedure

(f64vector? obj) procedure
(make-f64vector k [fill]) procedure
(f64vector inexact-real...) procedure
(f64vector-length f64vector) procedure
(f64vector-ref f64vector k) procedure
(f64vector-set! f64vector k inexact-real) procedure
(f64vector->list f64vector) procedure
(list->f64vector list-of-inexact-real) procedure
(f64vector-fill! f64vector fill) procedure
(subf64vector-fill! vector start end fill) procedure
(append-f64vectors lst) procedure
(f64vector-copy f64vector) procedure
(f64vector-append f64vector...) procedure
(subf64vector f64vector start end) procedure
(subf64vector-move! src-f64vector src-start src-end dst-f64vector dst-start)

(f64vector-shrink! f64vector k)

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)
> (u8vector-shrink! v 2)
> v
#u8(10 255)

(object->u8vector obj [encoder])

(u8vector->object u8vector [decoder])

The procedure object->u8vector returns a u8vector that contains the sequence of bytes that encodes the object obj. The procedure u8vector->object decodes the sequence of bytes contained in the u8vector u8vector, which was produced by the procedure object->u8vector, and reconstructs an object structurally equal to the original object. In other words the procedures object->u8vector and u8vector->object respectively perform serialization and deserialization of Scheme objects. Note that some objects are non-serializable (e.g. threads, wills, some types of ports, and any object containing a non-serializable object).

The optional encoder and decoder parameters are single parameter procedures which default to the identity function. The encoder procedure is called during serialization. As the serializer walks through obj, it calls the encoder procedure on each sub-object X that is encountered. The encoder transforms the object X into an object Y that will be serialized instead of X. Similarly the decoder procedure is called during deserialization. When an object Y is encountered, the decoder procedure is called to transform it into the object X that is the result of deserialization.

The encoder and decoder procedures are useful to customize the serialized representation of objects. In particular, it can be used to define the semantics of serializing objects, such as threads and ports, that would otherwise not be serializable. The decoder procedure is typically the inverse of the encoder procedure, i.e. (decoder (encoder X)) = X.

For example:
> (define (make-adder x) (lambda (y) (+ x y)))
> (define f (make-adder 10))
> (define a (object->u8vector f))
> (define b (u8vector->object a))
> (u8vector-length a)
1639
> (f 5)
15
> (b 5)
15
> (pp b)
(lambda (y) (+ x y))
11 Hashing and weak references

11.1 Hashing

\begin{align*}
\text{(object-\textgreater serial-number } & \text{ obj)} \quad \text{procedure} \\
\text{(serial-number-\textgreater object } & \text{ n [default])} \quad \text{procedure}
\end{align*}

All Scheme objects are uniquely identified with a serial number which is a nonnegative exact integer. The \text{object-\textgreater serial-number} procedure returns the serial number of object \text{obj}. This serial number is only allocated the first time the \text{object-\textgreater serial-number} procedure is called on that object. Objects which do not have an external textual representation that can be read by the \text{read} procedure, use an external textual representation that includes a serial number of the form \#n. Consequently, the procedures \text{write}, \text{pretty-print}, etc will call the \text{object-\textgreater serial-number} procedure to get the serial number, and this may cause the serial number to be allocated.

The \text{serial-number-\textgreater object} procedure takes an exact integer parameter \text{n} and returns the object whose serial number is \text{n}. If no object currently exists with that serial number, \text{default} is returned if it is specified, otherwise an unbound-serial-number-exception object is raised. The reader defines the following abbreviation for calling \text{serial-number-\textgreater object}: the syntax \#n, where \text{n} is a sequence of decimal digits and it is not followed by ‘=’ or ‘#’, is equivalent to the list (\text{serial-number-\textgreater object } \text{n}).

For example:
\begin{verbatim}
> (define z (list (lambda (x) (* x x)) (lambda (y) (/ 1 y))))
> z
(#<procedure #2> (#<procedure #3>)
> (#3 10)
1/10
> ‘(#3 10)
((serial-number->object 3) 10)
> car
#<procedure #4 car>
> (#4 z)
#<procedure #2>
\end{verbatim}

\begin{align*}
\text{(unbound-serial-number-exception? } & \text{ obj)} \quad \text{procedure} \\
\text{(unbound-serial-number-exception-procedure } & \text{ exc}) \quad \text{procedure} \\
\text{(unbound-serial-number-exception-arguments } & \text{ exc}) \quad \text{procedure}
\end{align*}

Unbound-serial-number-exception objects are raised by the procedure \text{serial-number-\textgreater object} when no object currently exists with that serial number. The parameter \text{exc} must be an unbound-serial-number-exception object.

The procedure \text{unbound-serial-number-exception?} returns \#t when \text{obj} is a unbound-serial-number-exception object and \#f otherwise.

The procedure \text{unbound-serial-number-exception-procedure} returns the procedure that raised \text{exc}.

The procedure \text{unbound-serial-number-exception-arguments} returns the list of arguments of the procedure that raised \text{exc}.

For example:
(symbol-hash symbol)
The symbol-hash procedure returns the hash number of the symbol symbol. The hash number is a small exact integer (fixnum). When symbol is an interned symbol the value returned is the same as (string=?-hash (symbol->string symbol)).
For example:
> (symbol-hash 'car)
444471047

(keyword-hash keyword)
The keyword-hash procedure returns the hash number of the keyword keyword. The hash number is a small exact integer (fixnum). When keyword is an interned keyword the value returned is the same as (string=?-hash (keyword->string keyword)).
For example:
> (keyword-hash car:)
444471047

(string=?-hash string)
The string=?-hash procedure returns the hash number of the string string. The hash number is a small exact integer (fixnum). For any two strings s1 and s2, (string=? s1 s2) implies (= (string=?-hash s1) (string=?-hash s2)).
For example:
> (string=?-hash "car")
444471047

(string-ci=?-hash string)
The string-ci=?-hash procedure returns the hash number of the string string. The hash number is a small exact integer (fixnum). For any two strings s1 and s2, (string-ci=? s1 s2) implies (= (string-ci=?-hash s1) (string-ci=?-hash s2)).
For example:
> (string-ci=?-hash "CaR")
444471047

(eq?-hash obj)
The eq?-hash procedure returns the hash number of the object obj. The hash number is a small exact integer (fixnum). For any two objects o1 and o2, (eq? o1 o2) implies (= (eq?-hash o1) (eq?-hash o2)).
For example:
> (eq?-hash #t)
536870910
The `eqv?-hash` procedure returns the hash number of the object `obj`. The hash number is a small exact integer (fixnum). For any two objects `o1` and `o2`, \((eqv? \; o1 \; o2)\) implies \(= (eqv?-hash \; o1) \; (eqv?-hash \; o2)\).

For example:

\[
> (eqv?-hash \; 1.5) \\
496387656
\]

The `equal?-hash` procedure returns the hash number of the object `obj`. The hash number is a small exact integer (fixnum). For any two objects `o1` and `o2`, \((equal? \; o1 \; o2)\) implies \(= (equal?-hash \; o1) \; (equal?-hash \; o2)\).

For example:

\[
> (equal?-hash \; (list \; 1 \; 2 \; 3)) \\
442438567
\]

### 11.2 Weak references

The garbage collector is responsible for reclaiming objects that are no longer needed by the program. This is done by analyzing the reachability graph of all objects from the roots (i.e. the global variables, the runnable threads, permanently allocated objects such as procedures defined in a compiled file, nonexecutable wills, etc). If a root or a reachable object `X` contains a reference to an object `Y` then `Y` is reachable. As a general rule, unreachable objects are reclaimed by the garbage collector.

There are two types of references: strong references and weak references. Most objects, including pairs, vectors, records and closures, contain strong references. An object `X` is **strongly reachable** if there is a path from the roots to `X` that traverses only strong references. Weak references only occur in wills and tables. There are two types of weak references: will-weak references and table-weak references. If all paths from the roots to an object `Y` traverse at least one table-weak reference, then `Y` will be reclaimed by the garbage collector. The will-weak references are used for finalization and are explained in the next section.

#### 11.2.1 Wills

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

\[
(make-will \; testator \; action) \quad \text{procedure}
\]
\[
(will? \; obj) \quad \text{procedure}
\]
\[
(will-testator \; will) \quad \text{procedure}
\]
\[
(will-execute! \; will) \quad \text{procedure}
\]

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

A will becomes **executable** when its `testator` object is not strongly reachable (i.e. the `testator` object is either unreachable or only reachable using paths from the roots that
traverse at least one weak reference). Some objects, including symbols, small exact integers (fixnums), booleans and characters, are considered to be always strongly reachable.

When the runtime system detects that a will has become executable the current computation is interrupted, the will’s testator is set to \#f and the will’s action procedure is called with the will’s testator as the sole argument. Currently only the garbage collector detects when wills become executable but this may change in future versions of Gambit (for example the compiler could perform an analysis to infer will executability at compile time). The garbage collector builds a list of all executable wills. 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 determined executability of the will. It is only when an object is not reachable from the roots that it is reclaimed by the garbage collector.

A remarkable feature of wills is that an action procedure can “resurrect” an object. An action procedure could for example assign the testator object to a global variable or create a new will with the same testator object.

For example:

```scheme
> (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
```

### 11.2.2 Tables

The following procedures implement the table data type. Tables are heterogenous structures whose elements are indexed by keys which are arbitrary objects. Tables are similar to association lists but are abstract and the access time for large tables is typically smaller. Each key contained in the table is bound to a value. The length of the table is the number of key/value bindings it contains. New key/value bindings can be added to a table, the value bound to a key can be changed, and existing key/value bindings can be removed.
The references to the keys can either be all strong or all table-weak and the references to the values can either be all strong or all table-weak. The garbage collector removes key/value bindings from a table when 1) the key is a table-weak reference and the key is unreachable or only reachable using paths from the roots that traverse at least one table-weak reference, or 2) the value is a table-weak reference and the value is unreachable or only reachable using paths from the roots that traverse at least one table-weak reference. Key/value bindings that are removed by the garbage collector are reclaimed immediately.

Although there are several possible ways of implementing tables, the current implementation uses hashing with open-addressing. This is space efficient and provides constant-time access. Hash tables are automatically resized to maintain the load within specified bounds. The load is the number of active entries (the length of the table) divided by the total number of entries in the hash table.

Tables are parameterized with a key comparison procedure. By default the equal? procedure is used, but eq?, eqv?, string=?, string-ci=?, or a user defined procedure can also be used. To support arbitrary key comparison procedures, tables are also parameterized with a hashing procedure accepting a key as its single parameter and returning a fixnum result. The hashing procedure hash must be consistent with the key comparison procedure test, that is, for any two keys k1 and k2 in the table, (test k1 k2) implies (= (hash k1) (hash k2)). A default hashing procedure consistent with the key comparison procedure is provided by the system. The default hashing procedure generally gives good performance when the key comparison procedure is eq?, eqv?, equal?, string=?, and string-ci=?.

Tables can be compared for equality using the equal? procedure. Two tables X and Y are considered equal by equal? when they have the same weakness attributes, the same key comparison procedure, the same hashing procedure, the same length, and for all the keys k in X, (equal? (table-ref X k) (table-ref Y k)).

(make-table [size: size] [init: init] [weak-keys: weak-keys] [weak-values: weak-values] [test: test] [hash: hash] [min-load: min-load] [max-load: max-load])

The procedure make-table returns a new table. The optional keyword parameters specify various parameters of the table.

The size parameter is a nonnegative exact integer indicating the expected length of the table. The system uses size to choose an appropriate initial size of the hash table so that it does not need to be resized too often.

The init parameter indicates a value that is associated to keys that are not in the table. When init is not specified, no value is associated to keys that are not in the table.

The weak-keys and weak-values parameters are extended booleans indicating respectively whether the keys and values are table-weak references (true) or strong references (false). By default the keys and values are strong references.

The test parameter indicates the key comparison procedure. The default key comparison procedure is equal?. The key comparison procedures eq?, eqv?, equal?, string=?, and string-ci=? are special because the system will use a reasonably good hash procedure when none is specified.
The `hash` parameter indicates the hash procedure. This procedure must accept a single key parameter, return a fixnum, and be consistent with the key comparison procedure. When `hash` is not specified, a default hash procedure is used. The default hash procedure is reasonably good when the key comparison procedure is `eq?`, `eqv?`, `equal?`, `string=?`, or `string-ci=?`.

The `min-load` and `max-load` parameters are real numbers that indicate the minimum and maximum load of the table respectively. The table is resized when adding or deleting a key/value binding would bring the table’s load outside of this range. The `min-load` parameter must be no less than 0.05 and the `max-load` parameter must be no greater than 0.95. Moreover the difference between `min-load` and `max-load` must be at least 0.20. When `min-load` is not specified, the value 0.45 is used. When `max-load` is not specified, the value 0.90 is used.

For example:

```scheme
> (define t (make-table))
> (table? t) #t
> (table-length t) 0
> (table-set! t (list 1 2) 3)
> (table-set! t (list 4 5) 6)
> (table-ref t (list 1 2)) 3
> (table-length t) 2
```

### Procedure `table? obj`

The procedure `table?` returns `#t` when `obj` is a table and `#f` otherwise.

For example:

```scheme
> (table? (make-table)) #t
> (table? 123) #f
```

### Procedure `table-length table`

The procedure `table-length` returns the number of key/value bindings contained in the table `table`.

For example:

```scheme
> (define t (make-table weak-keys: #t))
> (define x (list 1 2))
> (define y (list 3 4))
> (table-set! t x 111)
> (table-set! t y 222)
> (table-length t) 2
> (table-set! t x)
> (table-length t) 1
> (#gc)
> (table-length t) 1
> (set! y #f)
> (#gc)
```
> (table-length t)
0

(table-ref table key [default]) procedure
The procedure `table-ref` returns the value bound to the object `key` in the table `table`. When `key` is not bound and `default` is specified, `default` is returned. When `default` is not specified but an `init` parameter was specified when `table` was created, `init` is returned. Otherwise an unbound-table-key-exception object is raised.

For example:
> (define t1 (make-table init: 999))
> (table-set! t1 (list 1 2) 3)
> (table-ref t1 (list 1 2))
3
> (table-ref t1 (list 4 5))
999
> (table-ref t1 (list 4 5) #f)
#f
> (define t2 (make-table))
> (table-ref t2 (list 4 5))
*** ERROR IN (console)@7.1 -- Unbound table key

(table-set! table key [value]) procedure
The procedure `table-set!` binds the object `key` to `value` in the table `table`. When `value` is not specified, if `table` contains a binding for `key` then the binding is removed from `table`. The procedure `table-set!` returns an unspecified value.

For example:
> (define t (make-table))
> (table-set! t (list 1 2) 3)
> (table-set! t (list 4 5) 6)
> (table-set! t (list 4 5))
> (table-ref t (list 7 8))
> (table-set! t (list 1 2))
3
> (table-ref t (list 4 5))
*** ERROR IN (console)@7.1 -- Unbound table key

(table-search proc table) procedure
The procedure `table-search` searches the table `table` for a key/value binding for which the two parameter procedure `proc` returns a non false result. For each key/value binding visited by `table-search` the procedure `proc` is called with the key as the first parameter and the value as the second parameter. The procedure `table-search` returns the first non false value returned by `proc`, or `#f` if `proc` returned `#f` for all key/value bindings in `table`.

The order in which the key/value bindings are visited is unspecified and may vary from one call of `table-search` to the next. While a call to `table-search` is being performed on `table`, it is an error to call any of the following procedures on `table`: `table-ref`, `table-set!`, `table-search`, `table-for-each`, `table-copy`, `table-merge`, `table-merge!`, and `table->list`. It is also an error to compare with `equal?` (directly or indirectly with `member`, `assoc`, `table-ref`, etc.)
an object that contains table. All these procedures may cause table to be reordered and resized. This restriction allows a more efficient iteration over the key/value bindings.

For example:

```scheme
(define square (make-table))
(table-set! square 2 4)
(table-set! square 3 9)
(table-search (lambda (k v) (and (odd? k) v)) square)
```

procedure

(table-for-each proc table)

The procedure table-for-each calls the two parameter procedure proc for each key/value binding in the table table. The procedure proc is called with the key as the first parameter and the value as the second parameter. The procedure table-for-each returns an unspecified value.

The order in which the key/value bindings are visited is unspecified and may vary from one call of table-for-each to the next. While a call to table-for-each is being performed on table, it is an error to call any of the following procedures on table: table-ref, table-set!, table-search, table-for-each, and table->list. It is also an error to compare with equal? (directly or indirectly with member, assoc, table-ref, etc.) an object that contains table. All these procedures may cause table to be reordered and resized. This restriction allows a more efficient iteration over the key/value bindings.

For example:

```scheme
(define square (make-table))
(table-set! square 2 4)
(table-set! square 3 9)
(table-for-each (lambda (k v) (write (list k v)) (newline)) square)
```

procedure

(table->list table)

The procedure table->list returns an association list containing the key/value bindings in the table table. Each key/value binding yields a pair whose car field is the key and whose cdr field is the value bound to that key. The order of the bindings in the list is unspecified.

For example:

```scheme
(define square (make-table))
(table-set! square 2 4)
(table-set! square 3 9)
(table->list square)
((3 . 9) (2 . 4))
```

procedure

(list->table list [size: size] [init: init] [weak-keys: weak-keys] [weak-values: weak-values] [test: test] [hash: hash] [min-load: min-load] [max-load: max-load])

The procedure list->table returns a new table containing the key/value bindings in the association list list. The optional keyword parameters specify various parameters of the table and have the same meaning as for the make-table procedure.
Each element of list is a pair whose car field is a key and whose cdr field is the value bound to that key. If a key appears more than once in list (tested using the table’s key comparison procedure) it is the first key/value binding in list that has precedence.

For example:

> (define t (list->table '((b . 2) (a . 1) (c . 3) (a . 4))))
> (table->list t)
  ((a . 1) (b . 2) (c . 3))

(procedure (unbound-table-key-exception? obj))
(procedure (unbound-table-key-exception-procedure exc))
(procedure (unbound-table-key-exception-arguments exc))

Unbound-table-key-exception objects are raised by the procedure table-ref when the key does not have a binding in the table. The parameter exc must be an unbound-table-key-exception object.

The procedure unbound-table-key-exception? returns #t when obj is a unbound-table-key-exception object and #f otherwise.

The procedure unbound-table-key-exception-procedure returns the procedure that raised exc.

The procedure unbound-table-key-exception-arguments returns the list of arguments of the procedure that raised exc.

For example:

> (define t (make-table))
> (define (handler exc)
>   (if (unbound-table-key-exception? exc)
>       (list (unbound-table-key-exception-procedure exc)
>             (unbound-table-key-exception-arguments exc))
>       'not-unbound-table-key-exception))
> (with-exception-catcher handler
>   (lambda () (table-ref t '(1 2))))
  (#<procedure #2 table-ref> (#<table #3> (1 2)))

(procedure (table-copy table))

The procedure table-copy returns a new table containing the same key/value bindings as table and the same table parameters (i.e. hash procedure, key comparison procedure, key and value weakness, etc).

For example:

> (define t (list->table '((b . 2) (a . 1) (c . 3))))
> (define x (table-copy t))
> (table-set! t 'b 99)
> (table->list t)
  ((a . 1) (b . 99) (c . 3))
> (table->list x)
  ((a . 1) (b . 2) (c . 3))

(procedure (table-merge! table1 table2 [table2-takes-precedence?]))

The procedure table-merge! returns table1 after the key/value bindings contained in table2 have been added to it. When a key exists both in table1 and table2, then the parameter table2-takes-precedence? indicates which binding will be kept (the one
in \textit{table1} if \textit{table2-takes-precedence?} is false, and the one in \textit{table2} otherwise). If \textit{table2-takes-precedence?} is not specified the binding in \textit{table1} is kept.

For example:
\begin{verbatim}
> (define t1 (list->table '((a . 1) (b . 2) (c . 3))))
> (define t2 (list->table '((a . 4) (b . 5) (z . 6))))
> (table->list (table-merge! t1 t2))
  ((a . 1) (b . 2) (c . 3) (z . 6))
> (define t1 (list->table '((a . 1) (b . 2) (c . 3))))
> (define t2 (list->table '((a . 4) (b . 5) (z . 6))))
> (table->list (table-merge! t1 t2 #t))
  ((a . 4) (b . 5) (c . 3) (z . 6))
\end{verbatim}

\textbf{(table-merge \textit{table1} \textit{table2} \textit{table2-takes-precedence}? \textit{)}} procedure

The procedure \textit{table-merge} returns a copy of \textit{table1} (created with \textit{table-copy}) to which the key/value bindings contained in \textit{table2} have been added using \textit{table-merge!}. When a key exists both in \textit{table1} and \textit{table2}, then the parameter \textit{table2-takes-precedence?} indicates which binding will be kept (the one in \textit{table1} if \textit{table2-takes-precedence?} is false, and the one in \textit{table2} otherwise). If \textit{table2-takes-precedence?} is not specified the binding in \textit{table1} is kept.

For example:
\begin{verbatim}
> (define t1 (list->table '((a . 1) (b . 2) (c . 3))))
> (define t2 (list->table '((a . 4) (b . 5) (z . 6))))
> (table->list (table-merge! t1 t2))
  ((a . 1) (b . 2) (c . 3) (z . 6))
> (table->list (table-merge! t1 t2 #t))
  ((a . 4) (b . 5) (c . 3) (z . 6))
\end{verbatim}
12 Records

(define-structure name field...)

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 printed representation that 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 #2 x: 3 y: 5 color: red
> (point-x p)
3
> (point-color p)
red
> (point-color-set! p 'black)
> p
#:point #2 x: 3 y: 5 color: black
13 Threads

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

13.1 Introduction

Multithreading is a paradigm that is well suited for building complex systems such as: servers, GUIs, and high-level operating systems. Gambit’s thread system offers mechanisms for creating threads of execution and for synchronizing them. The thread system also supports features which are useful in a real-time context, such as priorities, priority inheritance and timeouts.

The thread system provides the following data types:

- Thread (a virtual processor which shares object space with all other threads)
- Mutex (a mutual exclusion device, also known as a lock and binary semaphore)
- Condition variable (a set of blocked threads)

13.2 Thread objects

A running thread is a thread that is currently executing. A runnable thread is a thread that is ready to execute or running. A thread is blocked if it is waiting for a mutex to become unlocked, an I/O operation to become possible, the end of a “sleep” period, etc. A new thread is a thread that has been allocated but has not yet been initialized. An initialized thread is a thread that can be made runnable. A new thread becomes runnable when it is started by calling thread-start!. A terminated thread is a thread that can no longer become runnable (but deadlocked threads are not considered terminated). The only valid transitions between the thread states are from new to initialized, from initialized to runnable, between runnable and blocked, and from any state except new to terminated as indicated in the following diagram:

```
start          unblock
NEW           INITIALIZED     RUNNABLE      BLOCKED
             \                     \                  /
             \                     V                  /
             +---------> TERMINATED <-----+
```

Each thread has a base priority, which is a real number (where a higher numerical value means a higher priority), a priority boost, which is a nonnegative real number representing the priority increase applied to a thread when it blocks, and a quantum, which is a nonnegative real number representing a duration in seconds.

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

Each thread has a mailbox which is used for inter-thread communication.
### 13.3 Mutex objects

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

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

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

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

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

### 13.4 Condition variable objects

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

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

### 13.5 Fairness

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

The definition of fairness requires the notion of time ordering, i.e. “event $A$ occurred before event $B$”. For the purpose of establishing time ordering, the scheduler uses a clock with a discrete, usually variable, resolution (a “tick”). Events occurring in a given tick can be considered to be simultaneous (i.e. if event $A$ occurred before event $B$ in real time, then the scheduler will claim that event $A$ occurred before event $B$ unless both events fall within the same tick, in which case the scheduler arbitrarily chooses a time ordering).

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

- The *base priority* is the value contained in $T$’s *base priority* field (which is set with the `thread-base-priority-set!` primitive).
• *T*'s *boosted flag* field contains a boolean that affects *T*'s *boosted priority*. When the boosted flag field is false, the boosted priority is equal to the base priority, otherwise the boosted priority is equal to the base priority plus the value contained in *T*'s *priority boost* field (which is set with the `thread-priority-boost-set!` primitive). The boosted flag field is set to false when a thread is created, when its quantum expires, and when `thread-yield!` is called. The boosted flag field is set to true when a thread blocks. By carefully choosing the base priority and priority boost, relatively to the other threads, it is possible to set up an interactive thread so that it has good I/O response time without being a CPU hog when it performs long computations.

• The *effective priority* is equal to the maximum of *T*'s boosted priority and the effective priority of all the threads that are blocked on a mutex owned by *T*. This *priority inheritance* avoids priority inversion problems that would prevent a high priority thread blocked at the entry of a critical section to progress because a low priority thread inside the critical section is preempted for an arbitrary long time by a medium priority thread.

Let \( P(T) \) be the effective priority of thread *T* and let \( R(T) \) be the most recent time when one of the following events occurred for thread *T*, thus making it runnable: *T* was started by calling `thread-start!`, *T* called `thread-yield!`, *T* expired its quantum, or *T* became unblocked. Let the relation \( NL(T_1, T_2) \), “\( T_1 \) no later than \( T_2 \)”, be true if \( P(T_1) < P(T_2) \) or \( P(T_1) = P(T_2) \) and \( R(T_1) > R(T_2) \), and false otherwise. The scheduler will schedule the execution of threads in such a way that whenever there is at least one runnable thread, 1) within a finite time at least one thread will be running, and 2) there is never a pair of runnable threads \( T_1 \) and \( T_2 \) for which \( NL(T_1, T_2) \) is true and \( T_1 \) is not running and \( T_2 \) is running.

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

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

### 13.6 Memory coherency

Read and write operations on the store (such as reading and writing a variable, an element of a vector or a string) are not atomic. It is an error for a thread to write a location in the store while some other thread reads or writes that same location. It is the responsibility of the application to avoid write/read and write/write races through appropriate uses of the synchronization primitives.

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

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

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

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

13.8 Primordial thread

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

13.9 Procedures

**procedure**

```
(current-thread)
```
This procedure returns the current thread. For example:

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

**procedure**

```
(thread? obj)
```
This procedure returns #t when obj is a thread object and #f otherwise. For example:

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

**procedure**

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

**procedure**

```
(make-root-thread thunk [name [thread-group [input-port [output-port]]]])
```

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

The make-root-thread procedure behaves like the make-thread procedure except the created thread does not inherit the dynamic environment from the current thread and the base priority is set to 0, the priority boost is set to 1.0e-6, and the quantum is set to 0.02. The dynamic environment of the thread has the global bindings of the parameter objects, except current-input-port which is bound to input-port, current-output-port which is bound to output-port, and current-directory which is bound to the initial current working directory of the current process. If input-port is not specified it defaults to a port corresponding to the standard input (‘stdin’). If output-port is not specified it defaults to a port corresponding to the standard output (‘stdout’).

For example:

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

(thread-name thread) procedure
This procedure returns the name of the thread. For example:

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

(thread-specific thread) procedure
(thread-specific-set! thread obj) procedure
The thread-specific procedure returns the content of the thread’s specific field. The thread-specific-set! procedure stores obj into the thread’s specific field and returns an unspecified value.

For example:

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

(thread-base-priority thread) procedure
(thread-base-priority-set! thread priority) procedure
The procedure thread-base-priority returns a real number which corresponds to the base priority of the thread.
The procedure `thread-base-priority-set!` changes the base priority of the thread to `priority` and returns an unspecified value. The `priority` must be a real number.

For example:

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

```
12.3
```

```
(thread-priority-boost thread)
```

```
(thread-priority-boost-set! thread priority-boost)
```

The procedure `thread-priority-boost` returns a real number which corresponds to the priority boost of the thread.

The procedure `thread-priority-boost-set!` changes the priority boost of the thread to `priority-boost` and returns an unspecified value. The `priority-boost` must be a nonnegative real.

For example:

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

```
2.5
```

```
(thread-quantum thread)
```

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

The procedure `thread-quantum` returns a real number which corresponds to the quantum of the thread.

The procedure `thread-quantum-set!` changes the quantum of the thread to `quantum` and returns an unspecified value. The `quantum` must be a nonnegative real. A value of zero selects the smallest quantum supported by the implementation.

For example:

```
> (thread-quantum-set! (current-thread) 1.5)
> (thread-quantum (current-thread))
```

```
1.5
```

```
> (thread-quantum-set! (current-thread) 0)
> (thread-quantum (current-thread))
```

```
0.
```

```
(thread-start! thread)
```

This procedure makes `thread` runnable and returns the `thread`. The `thread` must be an initialized thread.

For example:

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

```
ab> or ba> 
```

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

This procedure causes the current thread to exit the running state as if its quantum
had expired and returns an unspecified value.

For example:

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

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

(thread-sleep! timeout)

This procedure causes the current thread to wait until the timeout is reached and
returns an unspecified value. This blocks the thread only if timeout represents a
point in the future. It is an error for timeout to be ‘#f’.

For example:

; a clock with a gradual drift:

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

; a clock with no drift:

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

(thread-terminate! thread)

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

For example:

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

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

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

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

For example:

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

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

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

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

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

(let ((t1 (spawn increment-forever!))
  (t2 (spawn increment-forever!)))
  (thread-sleep! 1)
  (stop! t1))
(thread-join! thread [timeout [timeout-val]])

This procedure causes the current thread to wait until the thread terminates (normally or not) or until the timeout is reached if timeout is supplied. If the timeout is reached, thread-join! returns timeout-val if it is supplied, otherwise a join-timeout-exception object is raised. If the thread terminated normally, the content of the end-result field is returned, otherwise the content of the end-exception field is raised.

For example:

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

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

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

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

(thread-send thread msg)

Each thread has a mailbox which stores messages delivered to the thread in the order delivered.

The procedure thread-send adds the message msg at the end of the mailbox of thread thread and returns an unspecified value.

For example:
Chapter 13: Threads

> (thread-send (current-thread) 111)
> (thread-send (current-thread) 222)
> (thread-receive)
111
> (thread-receive)
222

(thread-receive [timeout [default]]) procedure
(thread-mailbox-next [timeout [default]]) procedure
(thread-mailbox-rewind) procedure
(thread-mailbox-extract-and-rewind) procedure

To allow a thread to examine the messages in its mailbox without removing them from the mailbox, each thread has a mailbox cursor which normally points to the last message accessed in the mailbox. When a mailbox cursor is rewound using the procedure thread-mailbox-rewind or thread-mailbox-extract-and-rewind or thread-receive, the cursor does not point to a message, but the next call to thread-receive and thread-mailbox-next will set the cursor to the oldest message in the mailbox.

The procedure thread-receive advances the mailbox cursor of the current thread to the next message, removes that message from the mailbox, rewinds the mailbox cursor, and returns the message. When timeout is not specified, the current thread will wait until a message is available in the mailbox. When timeout is specified and default is not specified, a mailbox-receive-timeout-exception object is raised if the timeout is reached before a message is available. When timeout is specified and default is specified, default is returned if the timeout is reached before a message is available.

The procedure thread-mailbox-next behaves like thread-receive except that the message remains in the mailbox and the mailbox cursor is not rewound.

The procedures thread-mailbox-rewind or thread-mailbox-extract-and-rewind rewind the mailbox cursor of the current thread so that the next call to thread-mailbox-next and thread-receive will access the oldest message in the mailbox. Additionally the procedure thread-mailbox-extract-and-rewind will remove from the mailbox the message most recently accessed by a call to thread-mailbox-next. When thread-mailbox-next has not been called since the last call to thread-receive or thread-mailbox-rewind or thread-mailbox-extract-and-rewind, a call to thread-mailbox-extract-and-rewind only resets the mailbox cursor (no message is removed).

For example:

> (thread-send (current-thread) 111)
> (thread-receive 1 999)
111
> (thread-send (current-thread) 222)
> (thread-send (current-thread) 333)
> (thread-mailbox-next 1 999)
222
> (thread-mailbox-next 1 999)
333
> (thread-mailbox-next 1 999)
Mailbox-receive-timeout-exception objects are raised by the procedures thread-
receive and thread-mailbox-next when a timeout expires before a message
is available and no default value is specified. The parameter exc must be a mailbox-
receive-timeout-exception object.

The procedure mailbox-receive-timeout-exception? returns #t when obj
is a mailbox-receive-timeout-exception object and #f otherwise.

The procedure mailbox-receive-timeout-exception-procedure returns
the procedure that raised exc.

The procedure mailbox-receive-timeout-exception-arguments returns
the list of arguments of the procedure that raised exc.

For example:

> (define (handler exc)
    (if (mailbox-receive-timeout-exception? exc)
        (list (mailbox-receive-timeout-exception-procedure exc)
             (mailbox-receive-timeout-exception-arguments exc))
             'not-mailbox-receive-timeout-exception))

(mutex? obj) procedure
This procedure returns #t when obj is a mutex object and #f otherwise.

For example:

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

(make-mutex [name]) procedure
This procedure returns a new mutex in the unlocked/not-abandoned state. The op-
tional name is an arbitrary Scheme object which identifies the mutex (useful for
debugging); it defaults to an unspecified value. The mutex’s specific field is set to an
unspecified value.

For example:

> (make-mutex)
#<mutex #2>
> (make-mutex 'foo)
#<mutex #3 foo>
(mutex-name mutex)  
procedure  
Returns the name of the mutex. For example:
  > (mutex-name (make-mutex 'foo))  
  foo

(mutex-specific mutex)  
(mutex-specific-set! mutex obj)  
procedure  
procedure  
The mutex-specific procedure returns the content of the mutex’s specific field.
The mutex-specific-set! procedure stores obj into the mutex’s specific field and returns an unspecified value.

For example:
  > (define m (make-mutex))  
  > (mutex-specific-set! m "hello")  
  > (mutex-specific m)  
  "hello"  
  > (define (mutex-lock-recursively! mutex)  
    (if (eq? (mutex-state mutex) (current-thread))  
      (let ((n (mutex-specific mutex)))  
        (mutex-specific-set! mutex (+ n 1)))  
      (begin  
        (mutex-lock! mutex)  
        (mutex-specific-set! mutex 0)))))  
  > (define (mutex-unlock-recursively! mutex)  
    (let ((n (mutex-specific mutex)))  
      (if (= n 0)  
        (mutex-unlock! mutex)  
        (mutex-specific-set! mutex (- n 1)))))  
  > (mutex-lock-recursively! m)  
  > (mutex-lock-recursively! m)  
  > (mutex-lock-recursively! m)  
  > (mutex-specific m)  
  2

(mutex-state mutex)  
procedure  
This procedure returns information about the state of the mutex. The possible results are:
- thread $T$: the mutex is in the locked/owned state and thread $T$ is the owner of the mutex
- symbol not-owned: the mutex is in the locked/not-owned state
- symbol abandoned: the mutex is in the unlocked/abandoned state
- symbol not-abandoned: the mutex is in the unlocked/not-abandoned state

For example:
  (mutex-state (make-mutex)) ==> not-abandoned
  
(define (thread-alive? thread)  
  (let ((mutex (make-mutex)))  
    (mutex-lock! mutex #:f thread)  
    (let ((state (mutex-state mutex)))  
      (mutex-unlock! mutex) ; avoid space leak  
      (eq? state thread))))
Chapter 13: Threads

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

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

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

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

For example:

; an implementation of a mailbox object of depth one; this
; implementation does not behave well in the presence of forced
; thread terminations using thread-terminate! (deadlock can occur
; if a thread is terminated in the middle of a put! or get! operation)
(define (make-empty-mailbox)
  (let ((put-mutex (make-mutex)) ; allow put! operation
         (get-mutex (make-mutex))
         (cell #f))
    (define (put! obj)
      (mutex-lock! put-mutex #f #f) ; prevent put! operation
      (set! cell obj)
      (mutex-unlock! get-mutex)) ; allow get! operation
    (define (get!)
      (mutex-lock! get-mutex #f #f) ; wait until object in mailbox
      (let ((result cell))
        (set! cell #f) ; prevent space leaks
        (mutex-unlock! put-mutex) ; allow put! operation
        result))
    (mutex-lock! get-mutex #f #f) ; prevent get! operation
    (lambda (msg)
      (case msg
        ((put!) put!)
        ((get!) get!)
        (else (error "unknown message")))))

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

; an alternate implementation of thread-sleep!
(define (sleep! timeout)
  (let ((m (make-mutex)))
    (mutex-lock! m #f #f)
    (mutex-lock! m timeout #f)))
(define (lock-one-of! mutex1 mutex2)
    ; this procedure assumes that neither mutex1 or mutex2 are owned by the current thread
    (let ((ct (current-thread))
          (done-mutex (make-mutex)))
      (mutex-lock! done-mutex #f #f)
      (let ((t1 (thread-start!
                (make-thread
                 (lambda ()
                  (mutex-lock! mutex1 #f ct)
                  (mutex-unlock! done-mutex)))))))
      (let ((t2 (thread-start!
                 (make-thread
                  (lambda ()
                  (mutex-lock! mutex2 #f ct)
                  (mutex-unlock! done-mutex)))))))
      (mutex-lock! done-mutex #f #f)
      (thread-terminate! t1)
      (thread-terminate! t2)
      (if (eq? (mutex-state mutex1) ct)
          (begin
           (if (eq? (mutex-state mutex2) ct)
               (mutex-unlock! mutex2) ; don’t lock both mutex1
               mutex2))))

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

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

NOTE: The reason the thread can unblock at any time (when condition-variable is supplied) is that the scheduler, when it detects a serious problem such as a deadlock, must interrupt one of the blocked threads (such as the primordial thread) so that it can perform some appropriate action. After a thread blocked on a condition-variable has handled such an interrupt it would be wrong for the scheduler to return the thread to the blocked state, because any calls to condition-variable-broadcast! during the interrupt will have gone unnoticed. It is necessary for the thread to remain runnable and return from the call to mutex-unlock! with a result of ‘#t’.

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

For example:

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

(condition-variable? obj)
This procedure returns #t when obj is a condition-variable object and #f otherwise.

For example:
> (condition-variable? (make-condition-variable))
#t
> (condition-variable? 'foo)
#f

(make-condition-variable [name])
This procedure returns a new empty condition variable. The optional name is an 
arbitrary Scheme object which identifies the condition variable (useful for debugging); 
it defaults to an unspecified value. The condition variable's specific field is set to an 
unspecified value.

For example:
> (make-condition-variable)
#<condition-variable #2>

(condition-variable-name condition-variable)
This procedure returns the name of the condition-variable. For example:
> (condition-variable-name (make-condition-variable 'foo))
foo

(condition-variable-specific condition-variable)
(condition-variable-specific-set! condition-variable obj)
The condition-variable-specific procedure returns the content of the 
condition-variable's specific field.

The condition-variable-specific-set! procedure stores obj into the 
condition-variable's specific field and returns an unspecified value.

For example:
> (define cv (make-condition-variable))
> (condition-variable-specific-set! cv "hello")
> (condition-variable-specific cv)
"hello"

(condition-variable-signal! condition-variable)
This procedure unblocks a thread blocked on the condition-variable (if there is at 
least one) and returns an unspecified value.
For example:

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

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

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

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

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

(condition-variable-broadcast! condition-variable) procedure
This procedure unblocks all the thread blocked on the condition-variable and returns
an unspecified value.

For example:

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

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

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

The dynamic environment is the structure which allows the system to find the value returned by the standard procedures `current-input-port` and `current-output-port`. The standard procedures `with-input-from-file` and `with-output-to-file` extend the dynamic environment to produce a new dynamic environment which is in effect for the dynamic extent of the call to the thunk passed as their last argument. These procedures are essentially special purpose dynamic binding operations on hidden dynamic variables (one for `current-input-port` and one for `current-output-port`). Gambit generalizes this dynamic binding mechanism to allow the user to introduce new dynamic variables, called parameter objects, and dynamically bind them. The parameter objects implemented by Gambit are compatible with the specification of the “Parameter objects SRFI” (SRFI 39).

One important issue is the relationship between the dynamic environments of the parent and child threads when a thread is created. Each thread has its own dynamic environment that is accessed when looking up the value bound to a parameter object by that thread. When a thread’s dynamic environment is extended it does not affect the dynamic environment of other threads. When a thread is created it is given a dynamic environment whose bindings are inherited from the parent thread. In this inherited dynamic environment the parameter objects are bound to the same cells as the parent’s dynamic environment (in other words an assignment of a new value to a parameter object is visible in the other thread).

Another important issue is the interaction between the `dynamic-wind` procedure and dynamic environments. When a thread creates a continuation, the thread’s dynamic environment and the ‘dynamic-wind’ stack are saved within the continuation (an alternate but equivalent point of view is that the ‘dynamic-wind’ stack is part of the dynamic environment). When this continuation is invoked the required ‘dynamic-wind’ before and after thunks are called and the saved dynamic environment is reinstated as the dynamic environment of the current thread. During the call to each required ‘dynamic-wind’ before and after thunk, the dynamic environment and the ‘dynamic-wind’ stack in effect when the corresponding ‘dynamic-wind’ was executed are reinstated. Note that this specification precisely defines the semantics of calling ‘call-with-current-continuation’ or invoking a continuation within a before or after thunk. The semantics are well defined even when a continuation created by another thread is invoked. Below is an example exercising the subtleties of this semantics.

```scheme
(with-output-to-file "foo"
  (lambda ()
    (let ((k (call-with-current-continuation
              (lambda (exit)
                (with-output-to-file "bar"
                  (lambda ()
                    (dynamic-wind
                      (lambda ()
                        (write '(bl))
                        (force-output)))
                      (lambda ()
                        (let ((x (call-with-current-continuation
```
The following actions will occur when this code is executed: (b1)(a1) is written to "bar", (b2) is then written to "foo", (t2) is then written to "baz", (a2) is then written to "foo", and finally (b1)(t1)(a1) is written to "bar".

(make-parameter obj [filter])

A parameter object is a procedure which accepts zero or one argument. The cell bound to a particular parameter object in the dynamic environment is accessed by calling the parameter object. When no argument is passed, the content of the cell is returned. When one argument is passed the content of the cell is updated with the result of applying the parameter object’s conversion procedure to the argument. Note that the conversion procedure can be used for guaranteeing the type of the parameter object’s binding and/or to perform some conversion of the value.

For example:

```
> (define radix (make-parameter 10))
> (radix)
10
> (radix 2)
> (radix)
```

The dynamic environment is composed of two parts: the local dynamic environment and the global dynamic environment. There is a single global dynamic environment, and it is used to lookup parameter objects that can’t be found in the local dynamic environment.

The make-parameter procedure returns a new parameter object. The filter argument is a one argument conversion procedure. If it is not specified, filter defaults to the identity function.

The global dynamic environment is updated to associate the parameter object to a new cell. The initial content of the cell is the result of applying the conversion procedure to obj.

The dynamic environment is composed of two parts: the local dynamic environment and the global dynamic environment. There is a single global dynamic environment, and it is used to lookup parameter objects that can’t be found in the local dynamic environment.

The make-parameter procedure returns a new parameter object. The filter argument is a one argument conversion procedure. If it is not specified, filter defaults to the identity function.

The global dynamic environment is updated to associate the parameter object to a new cell. The initial content of the cell is the result of applying the conversion procedure to obj.

A parameter object is a procedure which accepts zero or one argument. The cell bound to a particular parameter object in the dynamic environment is accessed by calling the parameter object. When no argument is passed, the content of the cell is returned. When one argument is passed the content of the cell is updated with the result of applying the parameter object’s conversion procedure to the argument. Note that the conversion procedure can be used for guaranteeing the type of the parameter object’s binding and/or to perform some conversion of the value.

For example:

```
> (define radix (make-parameter 10))
> (radix)
10
> (radix 2)
> (radix)
```
(define prompt
  (make-parameter
   123
   (lambda (x)
     (if (string? x)
         x
         (object->string x)))))

(prompt)
"123"
(prompt "$")

(define write-shared
  (make-parameter
    #f
    (lambda (x)
      (if (boolean? x)
          x
          (error "only booleans are accepted by write-shared")))))

(write-shared 123)
*** ERROR IN ##make-parameter -- only booleans are accepted by write-shared

(parameterize ((procedure value)...) body) special form

The parameterize form, evaluates all procedure and value expressions in an unspecified order. All the procedure expressions must evaluate to procedures, either parameter objects or procedures accepting zero and one argument. Then, for each procedure p and in an unspecified order:

- If p is a parameter object a new cell is created, initialized, and bound to the parameter object in the local dynamic environment. The value contained in the cell is the result of applying the parameter object’s conversion procedure to value. The resulting dynamic environment is then used for processing the remaining bindings (or the evaluation of body if there are no other bindings).

- Otherwise p will be used according to the following protocol: we say that the call (p) “gets p’s value” and that the call (p x) “sets p’s value to x”. First, the parameterize form gets p’s value and saves it in a local variable. It then sets p’s value to value. It then processes the remaining bindings (or evaluates body if there are no other bindings). Then it sets p’s value to the saved value. These steps are performed in a dynamic-wind so that it is possible to use continuations to jump into and out of the body (i.e. the dynamic-wind’s before thunk sets p’s value to value and the after thunk sets p’s value to the saved value).

The result(s) of the parameterize form are the result(s) of the body.

Note that using procedures instead of parameter objects may lead to unexpected results in multithreaded programs because the before and after thunks of the dynamic-wind are not called when control switches between threads.

For example:

  (define radix (make-parameter 2))
  (define prompt
    (make-parameter
     123)
(lambda (x)
  (if (string? x)
      x
      (object->string x))))

> (radix)
2
> (parameterize ((radix 16)) (radix))
16
> (radix)
2
> (define (f n) (number->string n (radix)))
> (f 10)
"1010"
> (parameterize ((radix 8)) (f 10))
"12"
> (parameterize ((radix 8) (prompt (f 10))) (prompt))
"1010"
> (define p (let ((x 1))
     (lambda args
       (if (null? args) x (set! x (car args))))))
> (let* ((a (p))
        (b (parameterize ((p 2)) (list (p))))
        (c (p)))
        (list a b c))
(1 (2) 1)
15 Exceptions

15.1 Exception-handling

Gambit’s exception-handling model is inspired from the withdrawn “Exception Handling SRFI” (SRFI 12), the “Multithreading support SRFI” (SRFI 18), and the “Exception Handling for Programs SRFI” (SRFI 34). The two fundamental operations are the dynamic binding of an exception handler (i.e. the procedure \texttt{with-exception-handler}) and the invocation of the exception handler (i.e. the procedure \texttt{raise}).

All predefined procedures which check for errors (including type errors, memory allocation errors, host operating-system errors, etc) report these errors using the exception-handling system (i.e. they “raise” an exception that can be handled in a user-defined exception handler). When an exception is raised and the exception is not handled by a user-defined exception handler, the predefined exception handler will display an error message (if the primordial thread raised the exception) or the thread will silently terminate with no error message (if it is not the primordial thread that raised the exception). This default behavior can be changed through the ‘- :d’ runtime option (see \textbf{Chapter 4 [Runtime options], page 23}).

Predefined procedures normally raise exceptions by performing a tail-call to the exception handler (the exceptions are “complex” procedures such as \texttt{eval}, \texttt{compile-file}, \texttt{read}, \texttt{write}, etc). This means that the continuation of the exception handler and of the REPL that may be started due to this is normally the continuation of the predefined procedure that raised the exception. By exiting the REPL with the \texttt{,(c expression)} command it is thus possible to resume the program as though the call to the predefined procedure returned the value of \texttt{expression}. For example:

\begin{verbatim}
> (define (f x) (+ (car x) 1))
> (f 2) ; typo... we meant to say (f '(2))
*** ERROR IN f, (console)@1.18 -- (Argument 1) PAIR expected
(car 2)
1> , (c 2)
3
\end{verbatim}

\texttt{(current-exception-handler [new-exception-handler])} procedure

The parameter object \texttt{current-exception-handler} is bound to the current exception-handler. Calling this procedure with no argument returns the current exception-handler and calling this procedure with one argument sets the current exception-handler to \texttt{new-exception-handler}.

For example:

\begin{verbatim}
> (current-exception-handler)
#<procedure #2 primordial-exception-handler>
> (current-exception-handler (lambda (exc) (pp exc) 999))
> (/ 1 0)
#<divide-by-zero-exception #3>
999
\end{verbatim}

\texttt{(with-exception-handler handler thunk)} procedure

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

For example:

```lisp
> (with-exception-handler
   (lambda (e) (write e) 5)
   (lambda () (+ 1 (* 2 3) 4)))
11
```

```lisp
> (with-exception-handler
   (lambda (e) (write e) 5)
   (lambda () (+ 1 (* 'foo 3) 4)))
#<type-exception #2>10
```

```lisp
> (with-exception-handler
   (lambda (e) (write e 9))
   (lambda () (+ 1 (* 'foo 3) 4)))
infinite loop
```

### (with-exception-catcher handler thunk)

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

For example:

```lisp
> (with-exception-catcher
   (lambda (e) (write e) 5)
   (lambda () (+ 1 (* 2 3) 4)))
11
```

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

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

```lisp
(write '#<type-exception #3> 9)
```

### (raise obj)

This procedure tail-calls the current exception-handler with obj as the sole argument. If the exception-handler returns, the continuation of the call to raise is invoked.

For example:

```lisp
> (with-exception-handler
   (lambda (exc)
     (pp exc)
     100)
   (lambda ()
     (+ 1 (raise "hello")))))
"hello"
```

101
The procedure `abort` calls the current exception-handler with `obj` as the sole argument. If the exception-handler returns, the procedure `abort` will be tail-called with a noncontinuable-exception object, whose reason field is `obj`, as sole argument.

Noncontinuable-exception objects are raised by the `abort` procedure when the exception-handler returns. The parameter `exc` must be a noncontinuable-exception object.

The procedure `noncontinuable-exception?` returns `#t` when `obj` is a noncontinuable-exception object and `#f` otherwise.

The procedure `noncontinuable-exception-reason` returns the argument of the call to `abort` that raised `exc`.

For example:

```scheme
> (call-with-current-continuation
 (lambda (k)
 (with-exception-handler
 (lambda (exc)
 (pp exc)
 (if (noncontinuable-exception? exc)
 (k (list (noncontinuable-exception-reason exc)))
 100))
 (lambda ()
 (+ 1 (abort "hello")))))
"hello"
#<noncontinuable-exception #2> ("hello")
```

### 15.2 Exception objects related to memory management

(heap-overflow-exception? `obj`) procedure

Heap-overflow-exception objects are raised when the allocation of an object would cause the heap to use more memory space than is available.

The procedure `heap-overflow-exception?` returns `#t` when `obj` is a heap-overflow-exception object and `#f` otherwise.

For example:

```scheme
> (define (handler exc)
 (if (heap-overflow-exception? exc)
 exc
 'not-heap-overflow-exception))
> (with-exception-catcher handler
 (lambda ()
 (define (f x) (f (cons 1 x)))
 (f '())))
#<heap-overflow-exception #2>
```

(stack-overflow-exception? `obj`) procedure

Stack-overflow-exception objects are raised when the allocation of a continuation frame would cause the heap to use more memory space than is available.
The procedure `stack-overflow-exception?` returns `#t` when `obj` is a stack-overflow-exception object and `#f` otherwise.

For example:

```scheme
(define (handler exc)
  (if (stack-overflow-exception? exc)
      exc
      'not-stack-overflow-exception))

(with-exception-catcher
  handler
  (lambda ()
    (define (f) (+ 1 (f)))
    (f)))
#<stack-overflow-exception #2>
```

### 15.3 Exception objects related to the host environment

- `os-exception? obj` procedure
- `os-exception-procedure exc` procedure
- `os-exception-arguments exc` procedure
- `os-exception-code exc` procedure
- `os-exception-message exc` procedure

Os-exception objects are raised by procedures which access the host operating-system’s services when the requested operation fails. The parameter `exc` must be a `os-exception` object.

The procedure `os-exception?` returns `#t` when `obj` is a `os-exception` object and `#f` otherwise.

The procedure `os-exception-procedure` returns the procedure that raised `exc`.

The procedure `os-exception-arguments` returns the list of arguments of the procedure that raised `exc`.

The procedure `os-exception-code` returns an exact integer error code that can be converted to a string by the `err-code->string` procedure. Note that the error code is operating-system dependent.

The procedure `os-exception-message` returns `#f` or a string giving details of the exception in a human-readable form.

For example:

```scheme
(define (handler exc)
  (if (os-exception? exc)
      (list (os-exception-procedure exc)
            (os-exception-arguments exc)
            (err-code->string (os-exception-code exc))
            (os-exception-message exc))
      'not-os-exception))

(with-exception-catcher
  handler
  (lambda () (host-info "x.y.z"))
  (#<procedure #2 host-info> ("x.y.z") "Unknown host" #f)

(no-such-file-or-directory-exception? obj) procedure
(no-such-file-or-directory-exception-procedure exc) procedure
```
(no-such-file-or-directory-exception-arguments exc)  procedure
No-such-file-or-directory-exception objects are raised by procedures which access the filesystem (such as open-input-file and directory-files) when the path specified can’t be found on the filesystem. The parameter exc must be a no-such-file-or-directory-exception object.

The procedure no-such-file-or-directory-exception? returns #t when obj is a no-such-file-or-directory-exception object and #f otherwise.

The procedure no-such-file-or-directory-exception-procedure returns the procedure that raised exc.

The procedure no-such-file-or-directory-exception-arguments returns the list of arguments of the procedure that raised exc.

For example:

> (define (handler exc)
  (if (no-such-file-or-directory-exception? exc)
      (list (no-such-file-or-directory-exception-procedure exc)
            (no-such-file-or-directory-exception-arguments exc))
      'not-no-such-file-or-directory-exception))
> (with-exception-catcher
   handler
   (lambda () (with-input-from-file "nofile" read)))
(#<procedure #2 with-input-from-file> ("nofile" #<procedure #3 read>))

(unbound-os-environment-variable-exception? obj)  procedure
(unbound-os-environment-variable-exception-procedure procedure exc)
(unbound-os-environment-variable-exception-arguments procedure exc)

Unbound-os-environment-variable-exception objects are raised when an unbound operating-system environment variable is accessed by the procedures getenv and setenv. The parameter exc must be an unbound-os-environment-variable-exception object.

The procedure unbound-os-environment-variable-exception? returns #t when obj is an unbound-os-environment-variable-exception object and #f otherwise.

The procedure unbound-os-environment-variable-exception-procedure returns the procedure that raised exc.

The procedure unbound-os-environment-variable-exception-arguments returns the list of arguments of the procedure that raised exc.

For example:

> (define (handler exc)
  (if (unbound-os-environment-variable-exception? exc)
      (list (unbound-os-environment-variable-exception-procedure exc)
            (unbound-os-environment-variable-exception-arguments exc))
      'not-unbound-os-environment-variable-exception))
> (with-exception-catcher
   handler
   (lambda () (getenv "DOES_NOT_EXIST")))
(#<procedure #2 getenv> ("DOES_NOT_EXIST"))
15.4 Exception objects related to threads

(scheduler-exception? obj) procedure
(scheduler-exception-reason exc) procedure

Scheduler-exception objects are raised by the scheduler when some operation requested from the host operating system failed (e.g., checking the status of the devices in order to wake up threads waiting to perform I/O on these devices). The parameter exc must be a scheduler-exception object.

The procedure scheduler-exception? returns #t when obj is a scheduler-exception object and #f otherwise.

The procedure scheduler-exception-reason returns the os-exception object that describes the failure detected by the scheduler.

(deadlock-exception? obj) procedure

Deadlock-exception objects are raised when the scheduler discovers that all threads are blocked and can make no further progress. In that case the scheduler unblocks the primordial-thread and forces it to raise a deadlock-exception object.

The procedure deadlock-exception? returns #t when obj is a deadlock-exception object and #f otherwise.

For example:

> (define (handler exc)
>   (if (deadlock-exception? exc)
>       exc
>       ’not-deadlock-exception))
> (with-exception-catcher
>   handler
>   (lambda () (read (open-vector))))
> #<deadlock-exception #2>

(abandoned-mutex-exception? obj) procedure

Abandoned-mutex-exception objects are raised when the current thread locks a mutex that was owned by a thread which terminated (see mutex-lock!).

The procedure abandoned-mutex-exception? returns #t when obj is a abandoned-mutex-exception object and #f otherwise.

For example:

> (define (handler exc)
>   (if (abandoned-mutex-exception? exc)
>       exc
>       ’not-abandoned-mutex-exception))
> (with-exception-catcher
>   handler
>   (lambda ()
>     (let ((m (make-mutex)))
>       (thread-join!
>         (thread-start!
>           (make-thread
>             (lambda () (mutex-lock! m))))
>           (mutex-lock! m))))
> #<abandoned-mutex-exception #2>
Chapter 15: Exceptions

(procedure (join-timeout-exception? obj))
(procedure (join-timeout-exception-procedure exc))
(procedure (join-timeout-exception-arguments exc))

Join-timeout-exception objects are raised when a call to the thread-join! procedure reaches its timeout before the target thread terminates and a timeout-value parameter is not specified. The parameter exc must be a join-timeout-exception object.

The procedure join-timeout-exception? returns #t when obj is a join-timeout-exception object and #f otherwise.

The procedure join-timeout-exception-procedure returns the procedure that raised exc.

The procedure join-timeout-exception-arguments returns the list of arguments of the procedure that raised exc.

For example:
> (define (handler exc)
  (if (join-timeout-exception? exc)
      (list (join-timeout-exception-procedure exc)
            (join-timeout-exception-arguments exc))
      'not-join-timeout-exception))
> (with-exception-catcher handler
  (lambda ()
    (thread-join!
      (thread-start!
        (make-thread
          (lambda () (thread-sleep! 10))
          5))))
  (#<procedure #2 thread-join!> (#<thread #3> 5))

(procedure (started-thread-exception? obj))
(procedure (started-thread-exception-procedure exc))
(procedure (started-thread-exception-arguments exc))

Started-thread-exception objects are raised when the target thread of a call to the procedure thread-start! is already started. The parameter exc must be a started-thread-exception object.

The procedure started-thread-exception? returns #t when obj is a started-thread-exception object and #f otherwise.

The procedure started-thread-exception-procedure returns the procedure that raised exc.

The procedure started-thread-exception-arguments returns the list of arguments of the procedure that raised exc.

For example:
> (define (handler exc)
  (if (started-thread-exception? exc)
      (list (started-thread-exception-procedure exc)
            (started-thread-exception-arguments exc))
      'not-started-thread-exception))
> (with-exception-catcher handler
(lambda ()
   (let ((t (make-thread (lambda () (expt 2 1000)))))
      (thread-start! t)
      (thread-start! t)))

(lambda ()
   (let ((t (make-thread (lambda () (expt 2 1000)))))
      (thread-start! t)
      (thread-start! t)))

(terminated-thread-exception? obj)  procedure
(terminated-thread-exception-procedure exc)  procedure
(terminated-thread-exception-arguments exc)  procedure

Terminated-thread-exception objects are raised when the thread-join! procedure is called and the target thread has terminated as a result of a call to the thread-terminate! procedure. The parameter exc must be a terminated-thread-exception object.

The procedure terminated-thread-exception? returns #t when obj is a terminated-thread-exception object and #f otherwise.

The procedure terminated-thread-exception-procedure returns the procedure that raised exc.

The procedure terminated-thread-exception-arguments returns the list of arguments of the procedure that raised exc.

For example:

> (define (handler exc)
    (if (terminated-thread-exception? exc)
        (list (terminated-thread-exception-procedure exc)
                (terminated-thread-exception-arguments exc))
        'not-terminated-thread-exception))

> (with-exception-catcher
    handler
    (lambda ()
       (thread-join!
            (thread-start!
                (make-thread
                    (lambda () (thread-terminate! (current-thread)))))))

(uncaught-exception? obj)  procedure
(uncaught-exception-procedure exc)  procedure
(uncaught-exception-arguments exc)  procedure
(uncaught-exception-reason exc)  procedure

Uncaught-exception objects are raised when an object is raised in a thread and that thread does not handle it (i.e. the thread terminated because it did not catch an exception it raised). The parameter exc must be an uncaught-exception object.

The procedure uncaught-exception? returns #t when obj is an uncaught-exception object and #f otherwise.

The procedure uncaught-exception-procedure returns the procedure that raised exc.

The procedure uncaught-exception-arguments returns the list of arguments of the procedure that raised exc.

The procedure uncaught-exception-reason returns the object that was raised by the thread and not handled by that thread.
For example:

```scheme
> (define (handler exc)
    (if (uncaught-exception? exc)
        (list (uncaught-exception-procedure exc)
              (uncaught-exception-arguments exc)
              (uncaught-exception-reason exc))
        'not-uncaught-exception))
> (with-exception-catcher
  handler
  (lambda ()
    (thread-join!
      (thread-start!
        (make-thread
          (lambda () (open-input-file "data" 99)))))))

#<procedure #2 thread-join!>
(#<thread #3>)
#<wrong-number-of-arguments-exception #4>)
```

### 15.5 Exception objects related to C-interface

- `(cfun-conversion-exception? obj)` procedure
- `(cfun-conversion-exception-procedure exc)` procedure
- `(cfun-conversion-exception-arguments exc)` procedure
- `(cfun-conversion-exception-code exc)` procedure
- `(cfun-conversion-exception-message exc)` procedure

Cfun-conversion-exception objects are raised by the C-interface when converting between the Scheme representation and the C representation of a value during a call from Scheme to C. The parameter `exc` must be a cfun-conversion-exception object.

The procedure `cfun-conversion-exception?` returns `#t` when `obj` is a cfun-conversion-exception object and `#f` otherwise.

The procedure `cfun-conversion-exception-procedure` returns the procedure that raised `exc`.

The procedure `cfun-conversion-exception-arguments` returns the list of arguments of the procedure that raised `exc`.

The procedure `cfun-conversion-exception-code` returns an exact integer error code that can be converted to a string by the `err-code->string` procedure.

The procedure `cfun-conversion-exception-message` returns `#f` or a string giving details of the exception in a human-readable form.

For example:

```
$ cat test1.scm
(define weird
  (c-lambda (char-string) nonnull-char-string
    "___result = ___arg1;"))
$ gsc test1.scm
$ gsi
Gambit v4.6.1

> (load "test1")
"/Users/feeley/gambit/doc/test1.o1"
> (weird "hello")
"hello"
```
Chapter 15: Exceptions

> (define (handler exc)
>   (if (cfun-conversion-exception? exc)
>       (list (cfun-conversion-exception-procedure exc)
>             (cfun-conversion-exception-arguments exc)
>             (err-code->string (cfun-conversion-exception-code exc))
>             (cfun-conversion-exception-message exc))
>       'not-cfun-conversion-exception)

> (with-exception-catcher
  handler
  (lambda () (weird 'not-a-string)))
(#<procedure #2 weird>
  (not-a-string)
  "(Argument 1) Can’t convert to C char-string"
  #f)

> (with-exception-catcher
  handler
  (lambda () (weird #f)))
(#<procedure #2 weird>
  (#f)
  "Can’t convert result from C nonnull-char-string"
  #f)

(sfun-conversion-exception? obj)  procedure
(sfun-conversion-exception-procedure exc)  procedure
(sfun-conversion-exception-arguments exc)  procedure
(sfun-conversion-exception-code exc)  procedure
(sfun-conversion-exception-message exc)  procedure

Sfun-conversion-exception objects are raised by the C-interface when converting between the Scheme representation and the C representation of a value during a call from C to Scheme. The parameter exc must be a sfun-conversion-exception object.

The procedure sfun-conversion-exception? returns #t when obj is a sfun-conversion-exception object and #f otherwise.

The procedure sfun-conversion-exception-procedure returns the procedure that raised exc.

The procedure sfun-conversion-exception-arguments returns the list of arguments of the procedure that raised exc.

The procedure sfun-conversion-exception-code returns an exact integer error code that can be converted to a string by the err-code->string procedure.

The procedure sfun-conversion-exception-message returns #f or a string giving details of the exception in a human-readable form.

For example:

$ cat test2.scm
(c-define (f str) (nonnull-char-string) int "f" ""
  (string->number str))
(define t1 (c-lambda () int "___result = f ("123");"))
(define t2 (c-lambda () int "___result = f (0);")
(define t3 (c-lambda () int "___result = f ("1.5");")
$ gsc test2.scm
$ gsi
Gambit v4.6.1

> (load "test2")
"/u/feeley/test2.o1"

> (t1) 123

> (define (handler exc)
>   (if (sfun-conversion-exception? exc)
>     (list (sfun-conversion-exception-procedure exc)
>           (sfun-conversion-exception-arguments exc)
>           (err-code->string (sfun-conversion-exception-code exc))
>           (sfun-conversion-exception-message exc))
>     'not-sfun-conversion-exception))

> (with-exception-catcher handler t2)
(#<procedure #2 f> ()
 ("(Argument 1) Can’t convert from C nonnull-char-string"
  #f))

> (with-exception-catcher handler t3)
(#<procedure #2 f> () "Can’t convert result to C int" #f)

(multiple-c-return-exception? obj)

procedure

Multiple-c-return-exception objects are raised by the C-interface when a C to Scheme
procedure call returns and that call’s stack frame is no longer on the C stack because
the call has already returned, or has been removed from the C stack by a longjump.

The procedure multiple-c-return-exception? returns #t when obj is a
multiple-c-return-exception object and #f otherwise.

For example:

$ cat test3.scm
(c-define (f str) (char-string) scheme-object "f" ""
 (pp (list 'entry 'str= str))
 (let ((k (call-with-current-continuation (lambda (k) k))))
   (pp (list 'exit 'k= k)))
 (define scheme-to-c-to-scheme-and-back
   (c-lambda (char-string) scheme-object
     "___result = f (___arg1);"))

$ gsc test3.scm
$ gsi
Gambit v4.6.1

> (load "test3")
"/Users/feeley/gambit/doc/test3.o1"

> (define (handler exc)
>   (if (multiple-c-return-exception? exc)
>     exc
>     'not-multiple-c-return-exception))

> (with-exception-catcher handler
>   (lambda ()
>     (let ((c (scheme-to-c-to-scheme-and-back "hello"))
>           (pp c)
>           (c 999)))
>     (entry str= "hello")
>     (exit k= #<procedure #2>))
>   (#<procedure #2>)
>   (exit k= 999)
>   (#<multiple-c-return-exception #3>
15.6 Exception objects related to the reader

Datum-parsing-exception objects are raised by the reader (i.e. the read procedure) when the input does not conform to the grammar for datum. The parameter exc must be a datum-parsing-exception object.

The procedure datum-parsing-exception? returns #t when obj is a datum-parsing-exception object and #f otherwise.

The procedure datum-parsing-exception-kind returns a symbol denoting the kind of parsing error that was encountered by the reader when it raised exc. Here is a table of the possible return values:

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datum-or eof-expected</td>
<td>Datum or EOF expected</td>
</tr>
<tr>
<td>datum-expected</td>
<td>Datum expected</td>
</tr>
<tr>
<td>improperly placed dot</td>
<td>Improperly placed dot</td>
</tr>
<tr>
<td>incomplete form eof-reached</td>
<td>Incomplete form, EOF reached</td>
</tr>
<tr>
<td>incomplete form</td>
<td>Incomplete form</td>
</tr>
<tr>
<td>character out of range</td>
<td>Character out of range</td>
</tr>
<tr>
<td>invalid character name</td>
<td>Invalid ' #' name</td>
</tr>
<tr>
<td>illegal character</td>
<td>Illegal character</td>
</tr>
<tr>
<td>s8-expected</td>
<td>Signed 8 bit exact integer expected</td>
</tr>
<tr>
<td>u8-expected</td>
<td>Unsigned 8 bit exact integer expected</td>
</tr>
<tr>
<td>s16-expected</td>
<td>Signed 16 bit exact integer expected</td>
</tr>
<tr>
<td>u16-expected</td>
<td>Unsigned 16 bit exact integer expected</td>
</tr>
<tr>
<td>s32-expected</td>
<td>Signed 32 bit exact integer expected</td>
</tr>
<tr>
<td>u32-expected</td>
<td>Unsigned 32 bit exact integer expected</td>
</tr>
<tr>
<td>s64-expected</td>
<td>Signed 64 bit exact integer expected</td>
</tr>
<tr>
<td>u64-expected</td>
<td>Unsigned 64 bit exact integer expected</td>
</tr>
<tr>
<td>inexact real-expected</td>
<td>Inexact real expected</td>
</tr>
<tr>
<td>invalid hex escape</td>
<td>Invalid hexadecimal escape</td>
</tr>
<tr>
<td>invalid escaped character</td>
<td>Invalid escaped character</td>
</tr>
<tr>
<td>open paren expected</td>
<td>'(' expected</td>
</tr>
<tr>
<td>invalid token</td>
<td>Invalid token</td>
</tr>
<tr>
<td>invalid sharp bang name</td>
<td>Invalid ' #!' name</td>
</tr>
<tr>
<td>duplicate label definition</td>
<td>Duplicate definition for label</td>
</tr>
<tr>
<td>missing label definition</td>
<td>Missing definition for label</td>
</tr>
<tr>
<td>illegal label definition</td>
<td>Illegal definition of label</td>
</tr>
<tr>
<td>invalid infix syntax character</td>
<td>Invalid infix syntax character</td>
</tr>
<tr>
<td>invalid infix syntax number</td>
<td>Invalid infix syntax number</td>
</tr>
<tr>
<td>invalid infix syntax</td>
<td>Invalid infix syntax</td>
</tr>
</tbody>
</table>
The procedure `datum-parsing-exception-parameters` returns a list of the parameters associated with the parsing error that was encountered by the reader when it raised `exc`.

For example:

```
> (define (handler exc)
    (if (datum-parsing-exception? exc)
        (list (datum-parsing-exception-kind exc)
            (datum-parsing-exception-parameters exc))
        'not-datum-parsing-exception))
> (with-exception-catcher
     handler
     (lambda ()
         (with-input-from-string "(s #\pace)" read)))
(invalid-character-name ("pace"))
```

**15.7 Exception objects related to evaluation and compilation**

- `expression-parsing-exception?` — procedure
- `expression-parsing-exception-kind` — procedure
- `expression-parsing-exception-parameters` — procedure
- `expression-parsing-exception-source` — procedure

Expression-parsing-exception objects are raised by the evaluator and compiler (i.e. the procedures `eval`, `compile-file`, etc) when the input does not conform to the grammar for expression. The parameter `exc` must be a `expression-parsing-exception` object.

The procedure `expression-parsing-exception?` returns `#t` when `obj` is a `expression-parsing-exception` object and `#f` otherwise.

The procedure `expression-parsing-exception-kind` returns a symbol denoting the kind of parsing error that was encountered by the evaluator or compiler when it raised `exc`. Here is a table of the possible return values:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id-expected</td>
<td>Identifier expected</td>
</tr>
<tr>
<td>ill-formed-namespace</td>
<td>Ill-formed namespace</td>
</tr>
<tr>
<td>ill-formed-namespace-prefix</td>
<td>Ill-formed namespace prefix</td>
</tr>
<tr>
<td>namespace-prefix-must-be-string</td>
<td>Namespace prefix must be a string</td>
</tr>
<tr>
<td>macro-used-as-variable</td>
<td>Macro name can't be used as a variable</td>
</tr>
<tr>
<td>variable-is-immutable</td>
<td>Variable is immutable</td>
</tr>
<tr>
<td>ill-formed-macro-transformer</td>
<td>Macro transformer must be a lambda expression</td>
</tr>
<tr>
<td>reserved-used-as-variable</td>
<td>Reserved identifier can't be used as a variable</td>
</tr>
<tr>
<td>ill-formed-special-form</td>
<td>Ill-formed special form</td>
</tr>
<tr>
<td>cannot-open-file</td>
<td>Can't open file</td>
</tr>
<tr>
<td>filename-expected</td>
<td>Filename expected</td>
</tr>
<tr>
<td>ill-placed-define</td>
<td>Ill-placed 'define'</td>
</tr>
<tr>
<td>ill-placed-**include</td>
<td>Ill-placed '##include'</td>
</tr>
<tr>
<td>ill-placed-**define-macro</td>
<td>Ill-placed '##define-macro'</td>
</tr>
</tbody>
</table>
The procedure expression-parsing-exception-parameters returns a list of the parameters associated with the parsing error that was encountered by the evaluator or compiler when it raised exc.

For example:

```scheme
> (define (handler exc)
    (if (expression-parsing-exception? exc)
        (list (expression-parsing-exception-kind exc)
              (expression-parsing-exception-parameters exc)
              'not-expression-parsing-exception))
> (with-exception-catcher
    handler
    (lambda ()
      (eval '(+ do 1))))
(reserved-used-as-variable (do))
```

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ill-placed-**declare</td>
<td>Ill-placed '##declare'</td>
</tr>
<tr>
<td>ill-placed-**namespace</td>
<td>Ill-placed '##namespace'</td>
</tr>
<tr>
<td>ill-formed-expression</td>
<td>Ill-formed expression</td>
</tr>
<tr>
<td>unsupported-special-form</td>
<td>Interpreter does not support</td>
</tr>
<tr>
<td>ill-placed-unquote</td>
<td>Ill-placed 'unquote'</td>
</tr>
<tr>
<td>ill-placed-unquote-splicing</td>
<td>Ill-placed 'unquote-splicing'</td>
</tr>
<tr>
<td>parameter-must-be-id</td>
<td>Parameter must be an identifier</td>
</tr>
<tr>
<td>parameter-must-be-id-or-default</td>
<td>Parameter must be an identifier or default binding</td>
</tr>
<tr>
<td>duplicate-parameter</td>
<td>Duplicate parameter in parameter list</td>
</tr>
<tr>
<td>ill-placed-dotted-rest-parameter</td>
<td>Ill-placed dotted rest parameter</td>
</tr>
<tr>
<td>parameter-expected-after-rest</td>
<td>#!rest must be followed by a parameter</td>
</tr>
<tr>
<td>ill-formed-default</td>
<td>Ill-formed default binding</td>
</tr>
<tr>
<td>ill-placed-optional</td>
<td>Ill-placed #!optional</td>
</tr>
<tr>
<td>ill-placed-rest</td>
<td>Ill-placed #!rest</td>
</tr>
<tr>
<td>ill-placed-key</td>
<td>Ill-placed #!key</td>
</tr>
<tr>
<td>key-expected-after-rest</td>
<td>#!key expected after rest parameter</td>
</tr>
<tr>
<td>ill-placed-default</td>
<td>Ill-placed default binding</td>
</tr>
<tr>
<td>duplicate-variable-definition</td>
<td>Duplicate definition of a variable</td>
</tr>
<tr>
<td>empty-body</td>
<td>Body must contain at least one expression</td>
</tr>
<tr>
<td>variable-must-be-id</td>
<td>Defined variable must be an identifier</td>
</tr>
<tr>
<td>else-clause-not-last</td>
<td>Else clause must be last</td>
</tr>
<tr>
<td>ill-formed-selector-list</td>
<td>Ill-formed selector list</td>
</tr>
<tr>
<td>duplicate-variable-binding</td>
<td>Duplicate variable in bindings</td>
</tr>
<tr>
<td>ill-formed-binding-list</td>
<td>Ill-formed binding list</td>
</tr>
<tr>
<td>ill-formed-call</td>
<td>Ill-formed procedure call</td>
</tr>
<tr>
<td>ill-formed-cond-expand</td>
<td>Ill-formed 'cond-expand'</td>
</tr>
<tr>
<td>unfulfilled-cond-expand</td>
<td>Unfulfilled 'cond-expand'</td>
</tr>
</tbody>
</table>

(procedure (unbound-global-exception? obj))
(procedure (unbound-global-exception-variable exc))
(procedure (unbound-global-exception-code exc))
(unbound-global-exception-rte exc)                  procedure

Unbound-global-exception objects are raised when an unbound global variable is ac-
cessed. The parameter exc must be an unbound-global-exception object.

The procedure unbound-global-exception? returns #t when obj is an unbound-global-exception object and #f otherwise.

The procedure unbound-global-exception-variable returns a symbol identifying the unbound global variable.

For example:

> (define (handler exc)
   (if (unbound-global-exception? exc)
       (list 'variable= (unbound-global-exception-variable exc))
       'not-unbound-global-exception))

> (with-exception-catcher
   handler
   (lambda () foo))

(variable= foo)

15.8 Exception objects related to type checking

(type-exception? obj)                             procedure

(type-exception-procedure exc)                   procedure

(type-exception-arguments exc)                  procedure

(type-exception-arg-num exc)                    procedure

(type-exception-type-id exc)                    procedure

Type-exception objects are raised when a primitive procedure is called with an argu-
ment of incorrect type (i.e. when a runtime type-check fails). The parameter exc
must be a type-exception object.

The procedure type-exception? returns #t when obj is a type-exception object and #f otherwise.

The procedure type-exception-procedure returns the procedure that raised exc.

The procedure type-exception-arguments returns the list of arguments of the
procedure that raised exc.

The procedure type-exception-arg-num returns the position of the argument
whose type is incorrect. Position 1 is the first argument.

The procedure type-exception-type-id returns an identifier of the type
expected. The type-id can be a symbol, such as number and string-or-
nonnegative-fixnum, or a record type descriptor.

For example:

> (define (handler exc)
   (if (type-exception? exc)
       (list (type-exception-procedure exc)
             (type-exception-arguments exc)
             (type-exception-arg-num exc)
             (type-exception-type-id exc))
       'not-type-exception))

> (with-exception-catcher
   handler
(lambda () (vector-ref '#(a b c) 'foo)))
(#<procedure #2 vector-ref> (#(a b c) foo) 2 exact-integer)
> (with-exception-catcher
  handler
  (lambda () (time->seconds 'foo)))
(#<procedure #3 time->seconds> (foo) 1 #<type #4 time>)

(range-exception? obj)
(range-exception-procedure exc)
(range-exception-arguments exc)
(range-exception-arg-num exc)

Range-exception objects are raised when a numeric parameter is not in the allowed range. The parameter exc must be a range-exception object.

The procedure range-exception? returns #t when obj is a range-exception object and #f otherwise.

The procedure range-exception-procedure returns the procedure that raised exc.

The procedure range-exception-arguments returns the list of arguments of the procedure that raised exc.

The procedure range-exception-arg-num returns the position of the argument which is not in the allowed range. Position 1 is the first argument.

For example:
> (define (handler exc)
      (if (range-exception? exc)
          (list (range-exception-procedure exc)
                (range-exception-arguments exc)
                (range-exception-arg-num exc))
          'not-range-exception))
> (with-exception-catcher
  handler
  (lambda () (string-ref "abcde" 10)))
(#<procedure #2 string-ref> ("abcde" 10) 2)

(divide-by-zero-exception? obj)
(divide-by-zero-exception-procedure exc)
(divide-by-zero-exception-arguments exc)

Divide-by-zero-exception objects are raised when a division by zero is attempted. The parameter exc must be a divide-by-zero-exception object.

The procedure divide-by-zero-exception? returns #t when obj is a divide-by-zero-exception object and #f otherwise.

The procedure divide-by-zero-exception-procedure returns the procedure that raised exc.

The procedure divide-by-zero-exception-arguments returns the list of arguments of the procedure that raised exc.

For example:
> (define (handler exc)
      (if (divide-by-zero-exception? exc)
          (list (divide-by-zero-exception-procedure exc)
                (divide-by-zero-exception-arguments exc))
Improper-length-list-exception objects are raised by the map and for-each proce-
dures when they are called with two or more list arguments and the lists are not of the
same length. The parameter exc must be an improper-length-list-exception object.
The procedure improper-length-list-exception? returns #t when obj is an
improper-length-list-exception object and #f otherwise.
The procedure improper-length-list-exception-procedure returns the
procedure that raised exc.
The procedure improper-length-list-exception-arguments returns the
list of arguments of the procedure that raised exc.
The procedure improper-length-list-exception-arg-num returns the posi-
tion of the argument whose length is the shortest. Position 1 is the first argument.
For example:

```scheme
> (define (handler exc)
    (if (improper-length-list-exception? exc)
        (list (improper-length-list-exception-procedure exc)
               (improper-length-list-exception-arguments exc)
               (improper-length-list-exception-arg-num exc))
        'not-improper-length-list-exception))
> (with-exception-catcher
    handler
    (lambda () (map + '(1 2) '(3) '(4 5)))))
(#<procedure #2 map> (#<procedure #3 +> (1 2) (3) (4 5)) 3)
```

15.9 Exception objects related to procedure call

Wrong-number-of-arguments-exception objects are raised when a procedure is called
with the wrong number of arguments. The parameter exc must be a wrong-number-
of-arguments-exception object.
The procedure wrong-number-of-arguments-exception? returns #t when
obj is a wrong-number-of-arguments-exception object and #f otherwise.
The procedure wrong-number-of-arguments-exception-procedure returns the
procedure that raised exc.
The procedure wrong-number-of-arguments-exception-arguments returns the
list of arguments of the procedure that raised exc.
For example:
> (define (handler exc)
  (if (wrong-number-of-arguments-exception? exc)
      (list (wrong-number-of-arguments-exception-procedure exc)
            (wrong-number-of-arguments-exception-arguments exc))
      ’not-wrong-number-of-arguments-exception))

> (with-exception-catcher
   handler
   (lambda () (open-input-file "data" 99)))
(#<procedure #2 open-input-file> ("data" 99))

---

(define (iota n) (if (= n 0) ’() (cons n (iota (- n 1)))))

> (define (handler exc)
  (if (number-of-arguments-limit-exception? exc)
      (list (number-of-arguments-limit-exception-procedure exc)
            (length (number-of-arguments-limit-exception-arguments exc)))
      ’not-number-of-arguments-limit-exception))

> (with-exception-catcher
   handler
   (lambda () (apply + 1 2 3 (iota 8190))))
(#<procedure #2 +> 8193)

---

(define (iota n) (if (= n 0) ’() (cons n (iota (- n 1)))))

> (define (handler exc)
  (if (number-of-arguments-limit-exception? exc)
      (list (number-of-arguments-limit-exception-procedure exc)
            (length (number-of-arguments-limit-exception-arguments exc)))
      ’not-number-of-arguments-limit-exception))

> (with-exception-catcher
   handler
   (lambda () (apply + 1 2 3 (iota 8190))))
(#<procedure #2 +> 8193)

---

(define (number-of-arguments-limit-exception? obj)
  procedure
(define (number-of-arguments-limit-exception-procedure exc)
  procedure
(define (number-of-arguments-limit-exception-arguments exc)
  procedure

Number-of-arguments-limit-exception objects are raised by the apply procedure when the procedure being called is passed more than 8192 arguments. The parameter exc must be a number-of-arguments-limit-exception object.

The procedure number-of-arguments-limit-exception? returns #t when obj is a number-of-arguments-limit-exception object and #f otherwise.

The procedure number-of-arguments-limit-exception-procedure returns the target procedure of the call to apply that raised exc.

The procedure number-of-arguments-limit-exception-arguments returns the list of arguments of the target procedure of the call to apply that raised exc.

For example:

> (define (iota n) (if (= n 0) ’() (cons n (iota (- n 1)))))
> (define (handler exc)
  (if (number-of-arguments-limit-exception? exc)
      (list (number-of-arguments-limit-exception-procedure exc)
            (length (number-of-arguments-limit-exception-arguments exc)))
      ’not-number-of-arguments-limit-exception))
> (with-exception-catcher
   handler
   (lambda () (apply + 1 2 3 (iota 8190))))
(#<procedure #2 +> 8193)

---

(define (nonprocedure-operator-exception? obj)
  procedure
(define (nonprocedure-operator-exception-operator exc)
  procedure
(define (nonprocedure-operator-exception-arguments exc)
  procedure
(define (nonprocedure-operator-exception-code exc)
  procedure
(define (nonprocedure-operator-exception-rte exc)
  procedure

Nonprocedure-operator-exception objects are raised when a procedure call is executed and the operator position is not a procedure. The parameter exc must be an nonprocedure-operator-exception object.

The procedure nonprocedure-operator-exception? returns #t when obj is an nonprocedure-operator-exception object and #f otherwise.

The procedure nonprocedure-operator-exception-operator returns the value in operator position of the procedure call that raised exc.

The procedure nonprocedure-operator-exception-arguments returns the list of arguments of the procedure call that raised exc.

For example:
> (define (handler exc)
>   (if (nonprocedure-operator-exception? exc)
>       (list (nonprocedure-operator-exception-operator exc)
>             (nonprocedure-operator-exception-arguments exc))
>       'not-nonprocedure-operator-exception))
> (with-exception-catcher
>   handler
>   (lambda () (11 22 33)))
(11 (22 33))

Unknown-keyword-argument-exception objects are raised when a procedure accepting keyword arguments is called and one of the keywords supplied is not among those that are expected. The parameter exc must be an unknown-keyword-argument-exception object.

The procedure unknown-keyword-argument-exception? returns #t when obj is an unknown-keyword-argument-exception object and #f otherwise.

The procedure unknown-keyword-argument-exception-procedure returns the procedure that raised exc.

The procedure unknown-keyword-argument-exception-arguments returns the list of arguments of the procedure that raised exc.

For example:

> (define (handler exc)
>   (if (unknown-keyword-argument-exception? exc)
>       (list (unknown-keyword-argument-exception-procedure exc)
>             (unknown-keyword-argument-exception-arguments exc))
>       'not-unknown-keyword-argument-exception))
> (with-exception-catcher
>   handler
>   (lambda () ((lambda (#!key (foo 5)) foo) bar: 11)))
(#<procedure #2> (bar: 11))

Keyword-expected-exception objects are raised when a procedure accepting keyword arguments is called and a nonkeyword was supplied where a keyword was expected. The parameter exc must be a keyword-expected-exception object.

The procedure keyword-expected-exception? returns #t when obj is a keyword-expected-exception object and #f otherwise.

The procedure keyword-expected-exception-procedure returns the procedure that raised exc.

The procedure keyword-expected-exception-arguments returns the list of arguments of the procedure that raised exc.

For example:

> (define (handler exc)
>   (if (keyword-expected-exception? exc)
(list (keyword-expected-exception-procedure exc)
  (keyword-expected-exception-arguments exc)
  'not-keyword-expected-exception))
> (with-exception-catcher
  handler
  (lambda () ((lambda (#
      key (foo 5)) foo) 11 22)))
  (#<procedure #2> (11 22))

15.10 Other exception objects

(error-exception? obj) procedure
(error-exception-message exc) procedure
(error-exception-parameters exc) procedure
(error message obj...) procedure

Error-exception objects are raised when the error procedure is called. The parameter exc must be an error-exception object.

The procedure error-exception? returns #t when obj is an error-exception object and #f otherwise.

The procedure error-exception-message returns the first argument of the call to error that raised exc.

The procedure error-exception-parameters returns the list of arguments, starting with the second argument, of the call to error that raised exc.

The error procedure raises an error-exception object whose message field is message and parameters field is the list of values obj... .

For example:
  > (define (handler exc)
        (if (error-exception? exc)
            (list (error-exception-message exc)
                  (error-exception-parameters exc))
                  'not-error-exception))
  > (with-exception-catcher
         handler
         (lambda () (error "unexpected object:" 123)))
         ("unexpected object:" (123))
Chapter 16: Host environment

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

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

16.1 Handling of file names

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

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

A path which starts with the characters '˜' denotes a file in a home directory. If nothing follows the '˜' then the directory denoted is the central installation directory. Otherwise what follows the '˜' is the name of the installation directory, for example '˜lib' denotes the 'lib' installation directory. Note that the location of the installation directories may be overridden by using the '-:=DIRECTORY' and '-:=DIR=DIRECTORY' runtime options or by defining the 'GAMBCOPT' environment variable.

A path which starts with the character '˜' not followed by '˜' denotes a file in the user's home directory. The user's home directory is contained in the 'HOME' environment variable under UNIX, Mac OS X, MSDOS and Microsoft Windows. Under MSDOS and Microsoft Windows, if the 'HOME' environment variable is not defined, the environment variables 'HOMEDRIVE' and 'HOMEPATH' are concatenated if they are defined. If this fails to yield a home directory, the central installation directory is used instead.

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

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

The parameter object current-directory is bound to the current working directory. Calling this procedure with no argument returns the absolute normalized path of the directory and calling this procedure with one argument sets the directory to new-current-directory. The initial binding of this parameter object is the current working directory of the current process. The path returned by current-directory always contains a trailing directory separator. Modifications of the parameter object do not change the current working directory of the current process (i.e. that is accessible with the UNIX getcwd() function and the Microsoft Windows GetCurrentDirectory function). It is an error to mutate the string returned by current-directory.
For example under UNIX:

```lisp
> (current-directory)
"/Users/feeley/gambit/doc/
> (current-directory ".")
> (current-directory)
"/Users/feeley/gambit/
> (path-expand "foo" "~")
"/usr/local/Gambit-C/foo"
> (parameterize ((current-directory "~")) (path-expand "foo"))
"/usr/local/Gambit-C/foo"
```

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

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

For example under UNIX:

```lisp
> (path-expand "foo")
"/Users/feeley/gambit/doc/foo"
> (path-expand "/foo")
"/Users/feeley/foo"
> (path-expand "lib/foo")
"/usr/local/Gambit-C/lib/foo"
> (path-expand "/./foo")
"/Users/feeley/gambit/doc/./foo"
> (path-expand "foo" ")
"foo"
> (path-expand "foo" "/tmp")
"/tmp/foo"
> (path-expand "this/file/does/not/exist")
"/Users/feeley/gambit/doc/this/file/does/not/exist"
> (path-expand ")
"/Users/feeley/gambit/doc/
```

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

The procedure path-normalize takes a path of a file or directory and returns its normalized path. The optional origin-directory parameter, which defaults to the current working directory, is the directory used to resolve relative paths. All components of the paths path and origin-directory must exist, except possibly the last component of path. A normalized path is a path containing no redundant parts and which is consistent with the current structure of the filesystem. A normalized path of a directory will always end with a path separator (i.e. ‘/’, ‘\’, or ‘:’ depending on the operating system). The optional allow-relative? parameter, which defaults to #f, indicates if the path returned can be expressed relatively to origin-directory: a #f requests an absolute path, the symbol shortest requests the shortest of the absolute and relative paths, and any other value requests the relative path. The shortest path is useful for interaction with the user because short relative paths are typically easier to read than long absolute paths.

For example under UNIX:

```lisp
> (path-expand "/./foo")
"/Users/feeley/gambit/doc/./foo"
```
> (path-normalize "../foo")
"/Users/feeley/gambit/foo"
> (path-normalize "this/file/does/not/exist")
*** ERROR IN (console)@3.1 -- No such file or directory
(path-normalize "this/file/does/not/exist")

(path-extension path) procedure
(path-strip-extension path) procedure
(path-directory path) procedure
(path-strip-directory path) procedure
(path-strip-trailing-directory-separator path) procedure
(path-volume path) procedure
(path-strip-volume path) procedure

These procedures extract various parts of a path, which need not be a normalized path.

The procedure path-extension returns the file extension (including the period) or
the empty string if there is no extension. The procedure path-strip-extension returns the path with the extension stripped off. The procedure path-directory returns the file’s directory (including the last path separator) or the empty string if no
directory is specified in the path. The procedure path-strip-directory returns the path with the directory stripped off. The procedure path-strip-trailing-directory-separator returns the path with the directory separator stripped off if
one is at the end of the path. The procedure path-volume returns the file’s volume (including the last path separator) or the empty string if no volume is specified in
the path. The procedure path-strip-volume returns the path with the volume stripped off.

For example under UNIX:

> (path-extension "/tmp/foo")
""
> (path-extension "/tmp/foo.txt")
".txt"
> (path-strip-extension "/tmp/foo.txt")
"/tmp/foo"
> (path-directory "/tmp/foo.txt")
"/tmp/
> (path-strip-directory "/tmp/foo.txt")
"foo.txt"
> (path-strip-trailing-directory-separator "/usr/local/bin/"
"/usr/local/bin"
> (path-strip-trailing-directory-separator "/usr/local/bin")
"/usr/local/bin"
> (path-volume "/tmp/foo.txt")
""
> (path-volume "C:/tmp/foo.txt")
"" ; result is "C:" under Microsoft Windows
> (path-strip-volume "C:/tmp/foo.txt")
"C:/tmp/foo.txt" ; result is "/tmp/foo.txt" under Microsoft Windows
16.2 Filesystem operations

\texttt{(create-directory \textit{path-or-settings})}

This procedure creates a directory. The argument \textit{path-or-settings} is either a string denoting a filesystem path or a list of port settings which must contain a \texttt{path}: setting. Here are the settings allowed:

- \texttt{path: string}
  This setting indicates the location of the directory to create in the filesystem. There is no default value for this setting.

- \texttt{permissions: 12-bit-exact-integer}
  This setting controls the UNIX permissions that will be attached to the file if it is created. The default value of this setting is \#o777.

For example:

\begin{verbatim}
> (create-directory "newdir")
> (create-directory "newdir")
*** ERROR IN (console)@2.1 -- File exists
(create-directory "newdir")
\end{verbatim}

\texttt{(create-fifo \textit{path-or-settings})}

This procedure creates a FIFO. The argument \textit{path-or-settings} is either a string denoting a filesystem path or a list of port settings which must contain a \texttt{path}: setting. Here are the settings allowed:

- \texttt{path: string}
  This setting indicates the location of the FIFO to create in the filesystem. There is no default value for this setting.

- \texttt{permissions: 12-bit-exact-integer}
  This setting controls the UNIX permissions that will be attached to the file if it is created. The default value of this setting is \#o666.

For example:

\begin{verbatim}
> (create-fifo "fifo")
> (define a (open-input-file "fifo"))
> (define b (open-output-file "fifo"))
> (display "1 22 333" b)
> (force-output b)
> (read a)
1
> (read a)
22
\end{verbatim}

\texttt{(create-link \textit{source-path} \textit{destination-path})}

This procedure creates a hard link between \textit{source-path} and \textit{destination-path}. The argument \textit{source-path} must be a string denoting the path of an existing file. The argument \textit{destination-path} must be a string denoting the path of the link to create.

\texttt{(create-symbolic-link \textit{source-path} \textit{destination-path})}

This procedure creates a symbolic link between \textit{source-path} and \textit{destination-path}. The argument \textit{source-path} must be a string denoting the path of an existing file. The argument \textit{destination-path} must be a string denoting the path of the symbolic link to create.
(rename-file source-path destination-path)  
 procedure  
 This procedure renames the file source-path to destination-path. The argument source-path must be a string denoting the path of an existing file. The argument destination-path must be a string denoting the new path of the file.

(copy-file source-path destination-path)  
 procedure  
 This procedure copies the file source-path to destination-path. The argument source-path must be a string denoting the path of an existing file. The argument destination-path must be a string denoting the path of the file to create.

(delete-file path)  
 procedure  
 This procedure deletes the file path. The argument path must be a string denoting the path of an existing file.

(delete-directory path)  
 procedure  
 This procedure deletes the directory path. The argument path must be a string denoting the path of an existing directory.

(directory-files [path-or-settings])  
 procedure  
 This procedure returns the list of the files in a directory. The argument path-or-settings is either a string denoting a filesystem path to a directory or a list of settings which must contain a path: setting. If it is not specified, path-or-settings defaults to the current directory (the value bound to the current-directory parameter object). Here are the settings allowed:

  • path: string  
    This setting indicates the location of the directory in the filesystem. There is no default value for this setting.

  • ignore-hidden: (#f | #t | dot-and-dot-dot)  
    This setting controls whether hidden-files will be returned. Under UNIX and Mac OS X hidden-files are those that start with a period (such as '.', '..', and '.profile'). Under Microsoft Windows hidden files are the '.', and '..' entries and the files whose “hidden file” attribute is set. A setting of #f will enumerate all the files. A setting of #t will only enumerate the files that are not hidden. A setting of dot-and-dot-dot will enumerate all the files except for the '.', and '..' hidden files. The default value of this setting is #t.

For example:

> (directory-files)  
("complex" "README" "simple")
> (directory-files "../include")
("config.h" "config.h.in" "gambit.h" "makefile" "makefile.in")
> (directory-files (list path: "../include" ignore-hidden: #f))
(."." ".." "config.h" "config.h.in" "gambit.h" "makefile" "makefile.in")

16.3 Shell command execution

(shell-command command)  
 procedure  
 The procedure shell-command calls up the shell to execute command which must be a string. This procedure returns the exit status of the shell in the form that the C library's system routine returns.
For example under UNIX:

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

### 16.4 Process termination

```
(exit [status])
```

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

For example under UNIX:

```
$ gsi
Gambit v4.6.1

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

### 16.5 Command line arguments

```
(command-line)
```

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

For example under UNIX:

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

### 16.6 Environment variables

```
(getenv name [default])
(setenv name [new-value])
```

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

The procedure `setenv` changes the binding of the environment variable `name` to `new-value` which must be a string. If `new-value` is not specified the binding is removed.

For example under UNIX:

```
> (getenv "HOME")
"/Users/feeley"
> (getenv "DOES_NOT_EXIST" #f)
```
Chapter 16: Host environment

16.7 Measuring time

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

```
(setenv "DOES_NOT_EXIST" "it does now")
(getenv "DOES_NOT_EXIST" #f)
"it does now"
(setenv "DOES_NOT_EXIST")
(getenv "DOES_NOT_EXIST" #f)
(setenv "DOES_NOT_EXIST"
(getenv "DOES_NOT_EXIST")
*** ERROR IN (console)@7.1 -- Unbound OS environment variable
```

procedure (current-time)
(procedure (time? obj))
(procedure (time->seconds time))
(procedure (seconds->time x))

The procedure current-time returns a time object representing the current point in real time.

The procedure time? returns #t when obj is a time object and #f otherwise.

The procedure time->seconds converts the time object time into an inexact real number representing the number of seconds elapsed since the “epoch” (which is 00:00:00 Coordinated Universal Time 01-01-1970).

The procedure seconds->time converts the real number x representing the number of seconds elapsed since the “epoch” into a time object.

For example:

```
> (current-time)
#<time #2>
> (time? (current-time))
#t
> (time? 123)
#f
> (time->seconds (current-time))
1083118758.63973
> (time->seconds (current-time))
1083118759.909163
> (seconds->time (+ 10 (time->seconds (current-time))))
#<time #3> ; a time object representing 10 seconds in the future
```

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

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

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

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

For example:

```scheme
> (process-times)
#f64(.02794 .021754 .159926176071167)
> (cpu-time)
.051223
> (real-time)
.40660619735717773
```

(time expr)

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

For example:

```scheme
> (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))
  683 ms real time
  558 ms cpu time (535 user, 23 system)
  8 collections accounting for 102 ms real time (70 user, 5 system)
  6400160 bytes allocated
  no minor faults
  no major faults
100000
```

16.8 File information

(file-exists? path [chase?])

The `path` argument must be a string. This procedure returns `#t` when a file by that name exists, and returns `#f` otherwise.

When `chase?` is present and `#f`, symbolic links will not be chased, in other words if `path` refers to a symbolic link, `file-exists?` will return `#t` whether or not it points to an existing file.
For example:

```scheme
> (file-exists? "nofile")
#f
```

(file-info path [chase?])

This procedure accesses the filesystem to get information about the file whose location is given by the string path. A file-information record is returned that contains the file's type, the device number, the inode number, the mode (permission bits), the number of links, the file's user id, the file's group id, the file's size in bytes, the times of last-access, last-modification and last-change, the attributes, and the creation time.

When chase? is present and #f, symbolic links will not be chased, in other words if path refers to a symbolic link the file-info procedure will return information about the link rather than the file it links to.

For example:

```scheme
> (file-info "/dev/tty")
#<file-info #2
type: character-special
device: 19513156
inode: 20728196
mode: 438
number-of-links: 1
owner: 0
group: 0
size: 0
last-access-time: #<time #3>
last-modification-time: #<time #4>
last-change-time: #<time #5>
attributes: 128
creation-time: #<time #6>>
```

(file-info? obj)

This procedure returns #t when obj is a file-information record and #f otherwise.

For example:

```scheme
> (file-info? (file-info "/dev/tty"))
#t
> (file-info? 123)
#f
```

(file-info-type file-info)

Returns the type field of the file-information record file-info. The type is denoted by a symbol. The following types are possible:

- regular Regular file
- directory Directory
- character-special Character special device
- block-special Block special device
- fifo FIFO
- symbolic-link Symbolic link
socket
unknown

File is of an unknown type

For example:
> (file-info-type (file-info "/dev/tty"))
character-special
> (file-info-type (file-info "/dev"))
directory

(file-info-device file-info)
procedure
Returns the device field of the file-information record file-info.
For example:
> (file-info-device (file-info "/dev/tty"))
19513156

(file-info-inode file-info)
procedure
Returns the inode field of the file-information record file-info.
For example:
> (file-info-inode (file-info "/dev/tty"))
20728196

(file-info-mode file-info)
procedure
Returns the mode field of the file-information record file-info.
For example:
> (file-info-mode (file-info "/dev/tty"))
438

(file-info-number-of-links file-info)
procedure
Returns the number-of-links field of the file-information record file-info.
For example:
> (file-info-number-of-links (file-info "/dev/tty"))
1

(file-info-owner file-info)
procedure
Returns the owner field of the file-information record file-info.
For example:
> (file-info-owner (file-info "/dev/tty"))
0

(file-info-group file-info)
procedure
Returns the group field of the file-information record file-info.
For example:
> (file-info-group (file-info "/dev/tty"))
0

(file-info-size file-info)
procedure
Returns the size field of the file-information record file-info.
For example:
> (file-info-size (file-info "/dev/tty"))
0
(file-info-last-access-time file-info) 
Returns the last-access-time field of the file-information record file-info.

For example:
> (file-info-last-access-time (file-info "~/dev/tty"))
#<time #2>

(file-info-last-modification-time file-info) 
Returns the last-modification-time field of the file-information record file-info.

For example:
> (file-info-last-modification-time (file-info "~/dev/tty"))
#<time #2>

(file-info-last-change-time file-info) 
Returns the last-change-time field of the file-information record file-info.

For example:
> (file-info-last-change-time (file-info "~/dev/tty"))
#<time #2>

(file-info-attributes file-info) 
Returns the attributes field of the file-information record file-info.

For example:
> (file-info-attributes (file-info "~/dev/tty"))
128

(file-info-creation-time file-info) 
Returns the creation-time field of the file-information record file-info.

For example:
> (file-info-creation-time (file-info "~/dev/tty"))
#<time #2>

(file-type path) 
(file-device path) 
(file-inode path) 
(file-mode path) 
(file-number-of-links path) 
(file-owner path) 
(file-group path) 
(file-size path) 
(file-last-access-time path) 
(file-last-modification-time path) 
(file-last-change-time path) 
(file-attributes path) 
(file-creation-time path)

These procedures combine a call to the file-info procedure and a call to a file-information record field accessor. For instance (file-type path) is equivalent to (file-info-type (file-info path)).
16.9 Group information

\begin{verbatim}
(group-info group-name-or-id) procedure
This procedure accesses the group database to get information about the group identified by group-name-or-id, which is the group’s symbolic name (string) or the group’s GID (exact integer). A group-information record is returned that contains the group’s symbolic name, the group’s id (GID), and the group’s members (list of symbolic user names).

For example:

> (group-info "staff")
#<group-info #2 name: "staff" gid: 20 members: ("root")>
> (group-info 29)
#<group-info #3
  name: "certusers"
  gid: 29
  members: ("root" "jabber" "postfix" "cyrusimap")>
> (group-info 5000)
*** ERROR IN (console)@3.1 -- Resource temporarily unavailable

(group-info? obj) procedure
This procedure returns #t when obj is a group-information record and #f otherwise.

For example:

> (group-info? (group-info "daemon"))
#t
> (group-info? 123)
#f

(group-info-name group-info) procedure
Returns the symbolic name field of the group-information record group-info.

For example:

> (group-info-name (group-info 29))
"certusers"

(group-info-gid group-info) procedure
Returns the group id field of the group-information record group-info.

For example:

> (group-info-gid (group-info "staff"))
20

(group-info-members group-info) procedure
Returns the members field of the group-information record group-info.

For example:

> (group-info-members (group-info "staff"))
("root")
\end{verbatim}
16.10 User information

(procedure (user-name))

This procedure returns the user’s name as a string.
For example:

> (user-name)
"feeley"

(procedure (user-info user-name-or-id))

This procedure accesses the user database to get information about the user identified by user-name-or-id, which is the user’s symbolic name (string) or the user’s UID (exact integer). A user-information record is returned that contains the user’s symbolic name, the user’s id (UID), the user’s group id (GID), the path to the user’s home directory, and the user’s login shell.
For example:

> (user-info "feeley")
#<user-info #2
  name: "feeley"
  uid: 506
  gid: 506
  home: "/Users/feeley"
  shell: "/bin/bash">
> (user-info 0)
#<user-info #3 name: "root" uid: 0 gid: 0 home: "/var/root" shell: "/bin/sh">
> (user-info 5000)
*** ERROR IN (console)@3.1 -- Resource temporarily unavailable

(procedure (user-info? obj))

This procedure returns #t when obj is a user-information record and #f otherwise.
For example:

> (user-info? (user-info "feeley"))
#t
> (user-info? 123)
#f

(procedure (user-info-name user-info))

Returns the symbolic name field of the user-information record user-info.
For example:

> (user-info-name (user-info 0))
"root"

(procedure (user-info-uid user-info))

Returns the user id field of the user-information record user-info.
For example:

> (user-info-uid (user-info "feeley"))
506

(procedure (user-info-gid user-info))

Returns the group id field of the user-information record user-info.
For example:
> (user-info-gid (user-info "feeley")) 506

(\texttt{user-info-home user-info}) \texttt{procedure}
Returns the home directory field of the user-information record \texttt{user-info}.
For example:
> (user-info-home (user-info 0)) 
"/var/root"

(\texttt{user-info-shell user-info}) \texttt{procedure}
Returns the shell field of the user-information record \texttt{user-info}.
For example:
> (user-info-shell (user-info 0)) 
"/bin/sh"

\textbf{16.11 Host information}

(\texttt{host-name}) \texttt{procedure}
This procedure returns the machine's host name as a string.
For example:
> (host-name) 
"mega.iro.umontreal.ca"

(\texttt{host-info host-name}) \texttt{procedure}
This procedure accesses the internet host database to get information about the machine whose name is denoted by the string \texttt{host-name}. A host-information record is returned that contains the official name of the machine, a list of aliases (alternative names), and a non-empty list of IP addresses for this machine. An exception is raised when \texttt{host-name} does not appear in the database.
For example:
> (host-info "www.google.com") 
<host-info #2 name: "www.1.google.com" aliases: ("www.google.com") addresses: (#u8(66 249 85 99) #u8(66 249 85 104))>
> (host-info "unknown.domain")
*** ERROR IN (console)@2.1 -- Unknown host
(host-info "unknown.domain")

(\texttt{host-info? obj}) \texttt{procedure}
This procedure returns \texttt{#t} when \texttt{obj} is a host-information record and \texttt{#f} otherwise.
For example:
> (host-info? (host-info "www.google.com"))
#t
> (host-info? 123)
#f

(\texttt{host-info-name host-info}) \texttt{procedure}
Returns the official name field of the host-information record \texttt{host-info}.
For example:
> (host-info-name (host-info "www.google.com"))
"www.1.google.com"
(host-info-aliases host-info)
  procedure
  Returns the aliases field of the host-information record host-info. This field is a possibly empty list of strings.
  For example:
  > (host-info-aliases (host-info "www.google.com"))
  ("www.google.com")

(host-info-addresses host-info)
  procedure
  Returns the addresses field of the host-information record host-info. This field is a non-empty list of u8vectors denoting IP addresses.
  For example:
  > (host-info-addresses (host-info "www.google.com"))
  (#u8(66 249 85 99) #u8(66 249 85 104))

(address-infos [host: host] [service: service] [family: family] [socket-type: socket-type] [protocol: protocol])
  procedure
  This procedure is an interface to the getaddrinfo system call. It accesses the internet host database to get information about the machine whose name is denoted by the string host and service is denoted by the string service and network address family is family (INET or INET6) and network socket-type is socket-type (STREAM or DGRAM or RAW) and network protocol is socket-type (TCP or UDP). A list of address-information records is returned.
  For example:
  > (address-infos host: "ftp.at.debian.org")
  (#<address-info #2
    family: INET6
    socket-type: DGRAM
    protocol: UDP
    socket-info:
      #<socket-info #3
        family: INET6
        port-number: 0
        address: #u16(8193 2136 2 1 0 0 0 16)>
    #<address-info #4
      family: INET6
      socket-type: STREAM
      protocol: TCP
      socket-info:
        #<socket-info #5
          family: INET6
          port-number: 0
          address: #u16(8193 2136 2 1 0 0 0 16)>
    #<address-info #6
      family: INET
      socket-type: DGRAM
      protocol: UDP
      socket-info:
        #<socket-info #7
          family: INET
          port-number: 0
          address: #u8(213 129 232 18)>
    #<address-info #8
      family: INET
      ...
socket-type: STREAM
protocol: TCP
socket-info:
  #<socket-info #9
    family: INET
    port-number: 0
    address: #u8(213 129 232 18)>)
> (address-infos host: "ftp.at.debian.org"
  family: 'INET
  protocol: 'TCP)
(#<address-info #10
  family: INET
  socket-type: STREAM
  protocol: TCP
  socket-info:
    #<socket-info #11
      family: INET
      port-number: 0
      address: #u8(213 129 232 18)>)
> (address-infos host: "unknown.domain")
*** ERROR IN (console)@5.1 -- nodename nor servname provided, or not known
(address-infos host: "unknown.domain")

(address-info? obj) procedure
This procedure returns #t when obj is an address-information record and #f otherwise.
For example:
> (map address-info? (address-infos host: "ftp.at.debian.org"))
(#t #t #t #t)
> (address-info? 123)
#f

(address-info-family address-info) procedure
Returns the family field of the address-information record address-info.
For example:
> (map address-info-family (address-infos host: "ftp.at.debian.org"))
(INET6 INET6 INET INET)

(address-info-socket-type address-info) procedure
Returns the socket-type field of the address-information record address-info.
For example:
> (map address-info-socket-type (address-infos host: "ftp.at.debian.org"))
(DGRAM STREAM DGRAM STREAM)

(address-info-protocol address-info) procedure
Returns the protocol field of the address-information record address-info.
For example:
> (map address-info-protocol (address-infos host: "ftp.at.debian.org"))
(UDP TCP UDP TCP)
(address-info-socket-info address-info) procedure

Returns the socket-info field of the address-information record address-info.

For example:

> (map address-info-socket-info
    (address-infos host: "ftp.at.debian.org"))

(#<socket-info #2
    family: INET6
    port-number: 0
    address: #u16(8193 2136 2 1 0 0 0 16)>
#<socket-info #3
    family: INET6
    port-number: 0
    address: #u16(8193 2136 2 1 0 0 0 16)>
#<socket-info #4
    family: INET
    port-number: 0
    address: #u8(213 129 232 18)>
#<socket-info #5
    family: INET
    port-number: 0
    address: #u8(213 129 232 18)>)

16.12 Service information

(service-info service-name-or-id) procedure

This procedure accesses the service database to get information about the service identified by service-name-or-id, which is the service’s symbolic name (string) or the service’s port number (exact integer). A service-information record is returned that contains the service’s symbolic name, a list of aliases (alternative names), the port number (exact integer), and the protocol name (string). An exception is raised when service-name-or-id does not appear in the database.

For example:

> (service-info "http")

#<service-info #2
    name: "http"
    aliases: ("www" "www-http")
    port-number: 80
    protocol: "udp">

> (service-info 80)

#<service-info #3
    name: "http"
    aliases: ("www" "www-http")
    port-number: 80
    protocol: "udp">

(service-info? obj) procedure

This procedure returns #t when obj is a service-information record and #f otherwise.

For example:

> (service-info? (service-info "http"))

#t

> (service-info? 123)

#f
(service-info-name service-info)

Returns the symbolic name field of the service-information record service-info.

For example:
>
(s service-info-name (service-info 80))
"http"

(service-info-aliases service-info)

Returns the aliases field of the service-information record service-info. This field is a possibly empty list of strings.

For example:
>
(service-info-aliases (service-info "http"))
("www" "www-http")

(service-info-port-number service-info)

Returns the service port number field of the service-information record service-info.

For example:
>
(service-info-port-number (service-info "http"))
80

(service-info-protocol service-info)

Returns the service protocol name field of the service-information record service-info.

For example:
>
(service-info-protocol (service-info "http"))
"udp"

16.13 Protocol information

(protocol-info protocol-name-or-id)

This procedure accesses the protocol database to get information about the protocol identified by protocol-name-or-id, which is the protocol’s symbolic name (string) or the protocol’s number (exact integer). A protocol-information record is returned that contains the protocol’s symbolic name, a list of aliases (alternative names), and the protocol number (32 bit unsigned exact integer). An exception is raised when protocol-name-or-id does not appear in the database.

For example:
>
(protocol-info "tcp")
#<protocol-info #2 name: "tcp" aliases: ("TCP") number: 6>
> (protocol-info 6)
#<protocol-info #2 name: "tcp" aliases: ("TCP") number: 6>

(protocol-info? obj)

This procedure returns #t when obj is a protocol-information record and #f otherwise.

For example:
>
(protocol-info? (protocol-info "tcp"))
#t
> (protocol-info? 123)
#f
(protocol-info-name protocol-info)  procedure
Returns the symbolic name field of the protocol-information record protocol-info.
For example:
  > (protocol-info-name (protocol-info 6))
  "tcp"

(protocol-info-aliases protocol-info)  procedure
Returns the aliases field of the protocol-information record protocol-info. This field is a possibly empty list of strings.
For example:
  > (protocol-info-aliases (protocol-info "tcp"))
  ("TCP")

(protocol-info-number protocol-info)  procedure
Returns the protocol number field of the protocol-information record protocol-info.
For example:
  > (protocol-info-number (protocol-info "tcp"))
  6

16.14 Network information

(network-info network-name-or-id)  procedure
This procedure accesses the network database to get information about the network identified by network-name-or-id, which is the network’s symbolic name (string) or the network’s number (exact integer). A network-information record is returned that contains the network’s symbolic name, a list of aliases (alternative names), and the network number (32 bit unsigned exact integer). An exception is raised when network-name-or-id does not appear in the database.
For example:
  > (network-info "loopback")
  #:network-info #2
  name: "loopback"
  aliases: ("loopback-net")
  number: 127>
  > (network-info 127)
  #:network-info #3
  name: "loopback"
  aliases: ("loopback-net")
  number: 127>

(network-info? obj)  procedure
This procedure returns #t when obj is a network-information record and #f otherwise.
For example:
  > (network-info? (network-info "loopback"))
  #t
  > (network-info? 123)
  #f
(network-info-name network-info)  procedure
 Returns the symbolic name field of the network-information record network-info.
 For example:
   > (network-info-name (network-info 127))
   "loopback"

(network-info-aliases network-info)  procedure
 Returns the aliases field of the network-information record network-info. This field is
 a possibly empty list of strings.
 For example:
   > (network-info-aliases (network-info "loopback")
   ("loopback-net")

(network-info-number network-info)  procedure
 Returns the network number field of the network-information record network-info.
 For example:
   > (network-info-number (network-info "loopback")
   127
17 I/O and ports

17.1 Unidirectional and bidirectional ports

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

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

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

17.2 Port classes

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

1. An object-port (or simply a port) provides operations to read and write Scheme data (i.e. any Scheme object) to/from the port. It also provides operations to force output to occur, to change the way threads block on the port, and to close the port. Note that the class of objects for which write/read invariance is guaranteed depends on the particular class of port.

2. A character-port provides all the operations of an object-port, and also operations to read and write individual characters to/from the port. When a Scheme object is written to a character-port, it is converted into the sequence of characters that corresponds to its external-representation. When reading a Scheme object, an inverse conversion occurs. Note that some Scheme objects do not have an external textual representation that can be read back.

3. A byte-port provides all the operations of a character-port, and also operations to read and write individual bytes to/from the port. When a character is written to a byte-port, some encoding of that character into a sequence of bytes will occur (for example, \newline will be encoded as the 2 bytes CR-LF when using ISO-8859-1 character encoding and cr-lf end-of-line encoding, and a non-ASCII character will generate more than 1 byte when using UTF-8 character encoding). When reading a character, a similar decoding occurs.

4. A device-port provides all the operations of a byte-port, and also operations to control the operating system managed device (file, network connection, terminal, etc) that is connected to the port.
17.3 Port settings

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

(open-input-file "foo")

This will use default settings for the character encoding, buffering, etc. If the UTF-8 character encoding is desired, then the port could be opened using the call

(open-input-file (list path: "foo" char-encoding: 'UTF-8))

Here the argument of the procedure open-input-file has been replaced by a port settings list which specifies the value of each port setting that should not be set to the default value. Note that some port settings have no useful default and it is therefore required to specify a value for them, such as the path: in the case of the file opening procedures. All port creation procedures (i.e. named open-...) take a single argument that can either be a port settings list or a value of a type that depends on the kind of port being created (a path string for files, an IP port number for socket servers, etc).

17.4 Object-ports

17.4.1 Object-port settings

The following is a list of port settings that are valid for all types of ports.

- **direction:** (input | output | input-output)
  
  This setting controls the direction of the port. The symbol input indicates a unidirectional input-port, the symbol output indicates a unidirectional output-port, and the symbol input-output indicates a bidirectional port. The default value of this setting depends on the port creation procedure.

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

17.4.2 Object-port operations

(input-port? obj)
(output-port? obj)
(port? obj)

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

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

For example:

```scheme
> (input-port? (current-input-port))
#t
> (call-with-input-string "some text" output-port?)
#f
> (port? (current-output-port))
#t
```

**procedure**

**(read [port])**

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

For example:

```scheme
> (call-with-input-string "some text" read)
some
> (call-with-input-string "" read)
#!eof
```

**procedure**

**(read-all [port [reader]])**

This procedure repeatedly calls the procedure `reader` with `port` as the sole argument and accumulates a list of each value returned up to the end-of-file object. The procedure `read-all` returns the accumulated list without the end-of-file object. If it is not specified, `port` defaults to the current input-port. If it is not specified, `reader` defaults to the procedure `read`.

For example:

```scheme
> (call-with-input-string "3,2,1\ngo!" read-all)
(3 ,2 ,1 go!)
> (call-with-input-string "3,2,1\ngo!"
 (lambda (p) (read-all p read-char)))
("3,2,1\ngo!"
 (lambda (p) (read-all p read-line)))
```

**procedure**

**(write obj [port])**

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

For example:

```scheme
> (write (list 'compare (list 'quote '@x) 'and (list 'unquote '@x)))
(compare '@x and , '@x)
```

**procedure**

**(newline [port])**

This procedure writes an “object separator” to the output-port `port` and the value returned is unspecified. The separator ensures that the next Scheme datum written with the `write` procedure will not be confused with the latest datum that was written. On character-ports this is done by writing the character `\newline`. On ports where successive objects are implicitly distinct (such as “vector ports”) this procedure does nothing.
Regardless of the class of a port \( p \) and assuming that the external textual representation of the object \( x \) is readable, the expression \( \text{begin (write } x \text{ } p \text{) (newline } p\text{)} \) will write to \( p \) a representation of \( x \) that can be read back with the procedure \( \text{read} \). If it is not specified, \( \text{port} \) defaults to the current output-port.

For example:

\[
\begin{align*}
\text{> (begin (write 123) (newline) (write 456) (newline))} \\
\text{123} \\
\text{456}
\end{align*}
\]

\textit{procedure} \( \text{force-output [port [level]]} \)

The procedure \( \text{force-output} \) causes the data that was written to the output-port \( \text{port} \) to be moved closer to its destination according to \( \text{level} \), an exact integer in the range \( 0 \) to \( 2 \). If \( \text{port} \) is not specified, the current output-port is used. If \( \text{level} \) is not specified, it defaults to \( 0 \). Values of \( \text{level} \) above \( 0 \) are equivalent to \( \text{level} = 0 \) except for device ports as explained below.

When \( \text{level} \) is \( 0 \), the output buffers of \( \text{port} \) which are managed in the Scheme process are drained (i.e. the output operation that was delayed due to buffering is actually performed). In the case of a device port the data is passed to the operating system and it becomes its responsibility to transmit the data to the device. The operating system may implement its own buffering approach which delays the transmission of the data to the device.

When \( \text{level} \) is \( 1 \), in addition to the operations for \( \text{level} = 0 \) and if the operating system supports the functionality, the operating system is requested to transmit the data to the device. On UNIX this corresponds to a \texttt{fsync} system call.

When \( \text{level} \) is \( 2 \), in addition to the operations for \( \text{level} = 1 \) and if the operating system supports the functionality, the operating system is requested to wait until the device reports that the data was saved by the device (e.g. actually written to disk in the case of a file). This operation can take a long time on some operating systems. On Mac OS X this corresponds to a \texttt{fcntl} system call with operation \texttt{F\_FULLFSYNC}.

For example:

\[
\begin{align*}
\text{> (define p (open-tcp-client "www.iro.umontreal.ca:80"))} \\
\text{> (display "GET /n" p)} \\
\text{> (force-output p)} \\
\text{> (read-line p)} \\
\text{"<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\""}
\end{align*}
\]

\textit{procedure} \( \text{close-input-port port} \)

\textit{procedure} \( \text{close-output-port port} \)

\textit{procedure} \( \text{close-port port} \)

The \( \text{port} \) argument of these procedures must be a unidirectional or a bidirectional port. For all three procedures the value returned is unspecified.

The procedure \( \text{close-input-port} \) closes the input-port side of \( \text{port} \), which must not be a unidirectional output-port.

The procedure \( \text{close-output-port} \) closes the output-port side of \( \text{port} \), which must not be a unidirectional input-port. The output buffers are drained before \( \text{port} \) is closed.

The procedure \( \text{close-port} \) closes all sides of the \( \text{port} \). Unless \( \text{port} \) is a unidirectional input-port, the output buffers are drained before \( \text{port} \) is closed.
For example:

```scheme
(define p (open-tcp-client "www.iro.umontreal.ca:80"))
(display "GET /\n" p)
(close-output-port p)
(read-line p)
"<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

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

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

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

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

When a port is created the timeout is set to infinity (`+inf.0`). This causes the thread to wait as long as needed for the operation to become possible. Setting the timeout to a point in the past (`-inf.0`) will cause the thread to attempt the I/O operation and never block (i.e. the timeout-thunk is called if the operation is not immediately possible).

The following example shows how to cause the REPL to terminate when the user does not enter an expression within the next 60 seconds.

```scheme
(input-port-timeout-set! (repl-input-port) 60)
*** EOF again to exit
```

17.5 Character-ports

17.5.1 Character-port settings

The following is a list of port settings that are valid for character-ports.
• **readtable**: `readtable`  
This setting determines the readtable attached to the character-port. To set each direction separately the keywords `input-readtable:` and `output-readtable:` must be used instead of `readtable:`. Readtables control the external textual representation of Scheme objects, that is the encoding of Scheme objects using characters. The behavior of the `read` procedure depends on the port's input-readtable and the behavior of the procedures `write`, `pretty-print`, and related procedures is affected by the port's output-readtable. The default value of this setting is the value bound to the parameter object `current-readtable`.

• **output-width**: `positive-integer`  
This setting indicates the width of the character output-port in number of characters. This information is used by the pretty-printer. The default value of this setting is 80.

### 17.5.2 Character-port operations

```
(input-port-line port)           procedure
(input-port-column port)         procedure
(output-port-line port)          procedure
(output-port-column port)        procedure
```

The current character location of a character input-port is the location of the next character to read. The current character location of a character output-port is the location of the next character to write. Location is denoted by a line number (the first line is line 1) and a column number, that is the location on the current line (the first column is column 1). The procedures `input-port-line` and `input-port-column` return the line location and the column location respectively of the character input-port `port`. The procedures `output-port-line` and `output-port-column` return the line location and the column location respectively of the character output-port `port`.

For example:

```
> (call-with-output-string
   '()  
   (lambda (p)
      (display "abc\n123def" p)
      (write (list (output-port-line p) (output-port-column p)))
   ))  
"abc\n123def(2 7)"
```

```
(output-port-width port)         procedure
```

This procedure returns the width, in characters, of the character output-port `port`. The value returned is the port's output-width setting.

For example:

```
> (output-port-width (repl-output-port))  
80
```

```
(read-char [port])              procedure
```

This procedure reads the character input-port `port` and returns the character at the current character location and advances the current character location to the next character, unless the `port` is already at end-of-file in which case `read-char` returns the end-of-file object. If it is not specified, `port` defaults to the current input-port.
For example:

```scheme
> (call-with-input-string
   "some text"
   (lambda (p)
     (let ((a (read-char p))) (list a (read-char p)))))))
(\s \o)
> (call-with-input-string "" read-char)
#!eof
```

(procedure peek-char [port])

This procedure returns the same result as read-char but it does not advance the current character location of the input-port port. If it is not specified, port defaults to the current input-port.

For example:

```scheme
> (call-with-input-string
   "some text"
   (lambda (p)
     (let ((a (peek-char p))) (list a (read-char p))))))
(\s \s)
> (call-with-input-string "" peek-char)
#!eof
```

(procedure write-char char [port])

This procedure writes the character char to the character output-port port and advances the current character location of that output-port. The value returned is unspecified. If it is not specified, port defaults to the current output-port.

For example:

```scheme
> (write-char #\=)
=>
```

(procedure read-line [port [separator [include-separator? [max-length]]]])

This procedure reads characters from the character input-port port until a specific separator or the end-of-file is encountered and returns a string containing the sequence of characters read. If it is specified, max-length must be a nonnegative exact integer and it places an upper limit on the number of characters that are read.

The separator is included at the end of the string only if it was the last character read and include-separator? is not #f. The separator must be a character or #f (in which case all the characters until the end-of-file are read). If it is not specified, port defaults to the current input-port. If it is not specified, separator defaults to #\newline. If it is not specified, include-separator? defaults to #f.

For example:

```scheme
> (define (split sep)
   (lambda (str)
     (call-with-input-string
      str
      (lambda (p)
        (read-all p (lambda (p) (read-line p sep)))))))
> ((split #\,)
   "a,b,c")
("a" "b" "c")
> (map (split #\,)
      (call-with-input-string "1,2,3\n4,5")
      "a" "b" "c")
(lambda (p) (read-all p read-line)))

(("1" "2" "3") ("4" "5"))

> (read-line (current-input-port) #\newline #\f 2)1234
"12"
> 34

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

These procedures support bulk character I/O. The part of the string string starting at index start and ending just before index end is used as a character buffer that will be the target of read-substring or the source of the write-substring. The read-substring also accepts a need parameter which must be a nonnegative fixnum. Up to end-start characters will be transferred. The number of characters transferred, possibly zero, is returned by these procedures. Fewer characters will be read by read-substring if an end-of-file is read, or a timeout occurs before all the requested characters are transferred and the timeout thunk returns #f (see the procedure input-port-timeout-set!), or need is specified and at least that many characters have been read (in other words the procedure does not block for more characters but may transfer more characters if they are immediately available). Fewer characters will be written by write-substring if a timeout occurs before all the requested characters are transferred and the timeout thunk returns #f (see the procedure output-port-timeout-set!). If it is not specified, port defaults to the current input-port and current output-port respectively.

For example:

> (define s (make-string 10 #\x))
> (read-substring s 2 5)123456789
3
> 456789
> s
"xx123xxxxx"
> (read-substring s 2 10 (current-input-port) 3)abcd
5
> s
"xxabcd\nxxxx"

(input-port-readtable port) procedure
(output-port-readtable port) procedure

These procedures return the readtable attached to the character-port port. The port parameter of input-port-readtable must be an input-port. The port parameter of output-port-readtable must be an output-port.

(input-port-readtable-set! port readtable) procedure
(output-port-readtable-set! port readtable) procedure

These procedures change the readtable attached to the character-port port to the readable readtable. The port parameter of input-port-readtable-set! must be an input-port. The port parameter of output-port-readtable-set! must be an output-port. The value returned is unspecified.

17.6 Byte-ports
17.6.1 Byte-port settings

The following is a list of port settings that are valid for byte-ports.

- **char-encoding**: *encoding*

  This setting controls the character encoding of the byte-port. For bidirectional byte-ports, the character encoding for input and output is set. To set each direction separately the keywords **input-char-encoding** and **output-char-encoding** must be used instead of **char-encoding**. The default value of this setting is operating system dependent, but this can be overridden through the runtime options (see Chapter 4 [Runtime options], page 23). The following encodings are supported:

  - **ISO-8859-1**: ISO-8859-1 character encoding. Each character is encoded by a single byte. Only Unicode characters with a code in the range 0 to 255 are allowed.

  - **ASCII**: ASCII character encoding. Each character is encoded by a single byte. In principle only Unicode characters with a code in the range 0 to 127 are allowed but most types of ports treat this exactly like ISO-8859-1.

  - **UTF-8**: UTF-8 character encoding. Each character is encoded by a sequence of one to four bytes. The minimum length UTF-8 encoding is used. If a BOM is needed at the beginning of the stream then it must be explicitly written.

  - **UTF-16**: UTF-16 character encoding. Each character is encoded by one or two 16 bit integers (2 or 4 bytes). The 16 bit integers may be encoded using little-endian encoding or big-endian encoding. If the port is an input-port and the first two bytes read are a BOM (“Byte Order Mark” character with hexadecimal code FEFF) then the BOM will be discarded and the endianness will be set accordingly, otherwise the endianness depends on the operating system and how the Gambit runtime was compiled. If the port is an output-port then a BOM will be output at the beginning of the stream and the endianness depends on the operating system and how the Gambit runtime was compiled.

  - **UTF-16LE**: UTF-16 character encoding with little-endian endianness. It is like UTF-16 except the endianness is set to little-endian and there is no BOM processing. If a BOM is needed at the beginning of the stream then it must be explicitly written.

  - **UTF-16BE**: UTF-16 character encoding with big-endian endianness. It is like UTF-16LE except the endianness is set to big-endian.


    These encodings combine the UTF-8 and UTF-16 encodings. When one of these character encodings is used for an output port, characters will be encoded using the UTF-8 encoding. The first character,
if there is one, is prefixed with a UTF-8 BOM (the three byte sequence EF BB BF in hexadecimal). When one of these character encodings is used for an input port, the character encoding depends on the first few bytes. If the first bytes of the stream are a UTF-16LE BOM (FF FE in hexadecimal), or a UTF-16BE BOM (FE FF in hexadecimal), or a UTF-8 BOM (EF BB BF in hexadecimal), then the BOM is discarded and the remaining bytes of the stream are decoded using the corresponding character encoding. If a BOM is not present, then the stream is decoded using the fallback encoding specified. The encoding UTF is a synonym for UTF-fallback-UTF-8. Note that the UTF character encoding for input will correctly handle streams produced using the encodings UTF, UTF-8, UTF-16, ASCII, and if an explicit BOM is output, the encodings UTF-16LE, and UTF-16BE.

**UCS-2**

UCS-2 character encoding. Each character is encoded by a 16 bit integer (2 bytes). The 16 bit integers may be encoded using little-endian encoding or big-endian encoding. If the port is an input-port and the first two bytes read are a BOM (“Byte Order Mark” character with hexadecimal code FEFF) then the BOM will be discarded and the endianness will be set accordingly, otherwise the endianness depends on the operating system and how the Gambit runtime was compiled. If the port is an output-port then a BOM will be output at the beginning of the stream and the endianness depends on the operating system and how the Gambit runtime was compiled.

**UCS-2LE**

UCS-2 character encoding with little-endian endianness. It is like UCS-2 except the endianness is set to little-endian and there is no BOM processing. If a BOM is needed at the beginning of the stream then it must be explicitly written.

**UCS-2BE**

UCS-2 character encoding with big-endian endianness. It is like UCS-2LE except the endianness is set to big-endian.

**UCS-4**

UCS-4 character encoding. Each character is encoded by a 32 integer (4 bytes). The 32 bit integers may be encoded using little-endian encoding or big-endian encoding. If the port is an input-port and the first four bytes read are a BOM (“Byte Order Mark” character with hexadecimal code 0000FEFF) then the BOM will be discarded and the endianness will be set accordingly, otherwise the endianness depends on the operating system and how the Gambit runtime was compiled. If the port is an output-port then a BOM will be output at the beginning of the stream and the endianness depends on the operating system and how the Gambit runtime was compiled.

**UCS-4LE**

UCS-4 character encoding with little-endian endianness. It is like UCS-4 except the endianness is set to little-endian and there is
no BOM processing. If a BOM is needed at the beginning of the stream then it must be explicitly written.

**UCS-4BE**

UCS-4 character encoding with big-endian endianness. It is like UCS-4LE except the endianness is set to big-endian.

- **char-encoding-errors:** (#f | #t)
  
  This setting controls whether illegal character encodings are silently replaced with the Unicode character #xfffd (replacement character) or raise an error. To set each direction separately the keywords `input-char-encoding-errors:` and `output-char-encoding-errors:` must be used instead of `char-encoding-errors:`. The default value of this setting is #t.

- **eol-encoding:** `encoding`
  
  This setting controls the end-of-line encoding of the byte-port. To set each direction separately the keywords `input-eol-encoding:` and `output-eol-encoding:` must be used instead of `eol-encoding:`. The default value of this setting is operating system dependent, but this can be overridden through the runtime options (see Chapter 4 [Runtime options], page 23). Note that for output-ports the end-of-line encoding is applied before the character encoding, and for input-ports it is applied after. The following encodings are supported:

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

  - **cr**
    
    For an output-port, writing a `\newline` character outputs a `\return` character to the stream (Unicode character code 13). For an input-port, a `\newline` character is read when a `\linefeed` character or a `\return` character is encountered on the stream. Text files created by Classic Mac OS applications typically use this end-of-line encoding.

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

(read-u8 [port]) procedure
This procedure reads the byte input-port port and returns the byte at the current
byte location and advances the current byte location to the next byte, unless the port
is already at end-of-file in which case read-u8 returns the end-of-file object. If it is
not specified, port defaults to the current input-port.

This procedure must be called when the port’s input character buffer is empty other-
wise the character-stream and byte-stream may be out of sync due to buffering. The
input character buffer is used for bulk decoding of encoded characters (i.e. to trans-
late the byte-stream into a character-stream). The input character buffer is initially
empty. It is only when characters are read that it is filled with characters obtained
by decoding the byte-stream.

One way to ensure that the port’s input character buffer is empty is to call read-
u8 strictly before any use of the port in a character input operation (i.e. a call to
the procedures read, read-char, peek-char, etc). Alternatively input-port-
characters-buffered can be used to get the number of characters in the port’s
input character buffer, and to empty the buffer with calls to read-char or read-
substring.

For example:
> (call-with-input-u8vector '#u8(11 22 33 44)
  (lambda (p) (let ((a (read-u8 p))) (list a (read-u8 p))))))
(11 22)
> (call-with-input-u8vector '#u8() read-u8)
#!eof

(write-u8 n [port]) procedure
This procedure writes the byte n to the byte output-port port and advances the
current byte location of that output-port. The value returned is unspecified. If it is
not specified, port defaults to the current output-port.

For example:
> (call-with-output-u8vector '() (lambda (p) (write-u8 33 p)))
#u8(33)

(read-subu8vector u8vector start end [port [need]]) procedure
(write-subu8vector u8vector start end [port]) procedure
These procedures support bulk byte I/O. The part of the u8vector u8vector starting
at index start and ending just before index end is used as a byte buffer that will be
the target of read-subu8vector or the source of the write-subu8vector. The
read-subu8vector also accepts a need parameter which must be a nonnegative
fixnum. Up to end-start bytes will be transferred. The number of bytes transferred,
possibly zero, is returned by these procedures. Fewer bytes will be read by read-
subu8vector if an end-of-file is read, or a timeout occurs before all the requested
bytes are transferred and the timeout thunk returns #f (see the procedure input-
port-timeout-set!), or need is specified and at least that many bytes have been
read (in other words the procedure does not block for more bytes but may transfer
more bytes if they are immediately available). Fewer bytes will be written by write-subu8vector if a timeout occurs before all the requested bytes are transferred and the timeout thunk returns #f (see the procedure output-port-timeout-set!). If it is not specified, port defaults to the current input-port and current output-port respectively.

The procedure read-subu8vector must be called before any use of the port in a character input operation (i.e. a call to the procedures read, read-char, peek-char, etc) because otherwise the character-stream and byte-stream may be out of sync due to the port buffering.

For example:

```scheme
> (define v (make-u8vector 10))
> (read-subu8vector v 2 5)12345
3
> 456789
> v
#u8(0 0 49 50 51 0 0 0 0 0)
> (read-subu8vector v 2 10 (current-input-port) 3)abcd
5
> v
#u8(0 0 97 98 99 100 10 0 0 0)
```

### 17.7 Device-ports

#### 17.7.1 Filesystem devices

- `(open-file path-or-settings)`
- `(open-input-file path-or-settings)`
- `(open-output-file path-or-settings)`
- `(call-with-input-file path-or-settings proc)`
- `(call-with-output-file path-or-settings proc)`
- `(with-input-from-file path-or-settings thunk)`
- `(with-output-to-file path-or-settings thunk)`

All of these procedures create a port to interface to a byte-stream device (such as a file, console, serial port, named pipe, etc) whose name is given by a path of the filesystem. The direction: setting will default to the value input for the procedures open-input-file, call-with-input-file and with-input-from-file, to the value output for the procedures open-output-file, call-with-output-file and with-output-to-file, and to the value input-output for the procedure open-file.

The procedures open-file, open-input-file and open-output-file return the port that is created. The procedures call-with-input-file and call-with-output-file call the procedure proc with the port as single argument, and then return the value(s) of this call after closing the port. The procedures with-input-from-file and with-output-to-file dynamically bind the current input-port and current output-port respectively to the port created for the duration of a call to the procedure thunk with no argument. The value(s) of the call to thunk are returned after closing the port.
The first argument of these procedures is either a string denoting a filesystem path or a list of port settings which must contain a path: setting. Here are the settings allowed in addition to the generic settings of byte-ports:

- **path: string**
  This setting indicates the location of the file in the filesystem. There is no default value for this setting.

- **append: ( #f | #t )**
  This setting controls whether output will be added to the end of the file. This is useful for writing to log files that might be open by more than one process. The default value of this setting is #f.

- **create: ( #f | #t | maybe )**
  This setting controls whether the file will be created when it is opened. A setting of #f requires that the file exist (otherwise an exception is raised). A setting of #t requires that the file does not exist (otherwise an exception is raised). A setting of maybe will create the file if it does not exist. The default value of this setting is maybe for output-ports and #f for input-ports and bidirectional ports.

- **permissions: 12-bit-exact-integer**
  This setting controls the UNIX permissions that will be attached to the file if it is created. The default value of this setting is #o666.

- **truncate: ( #f | #t )**
  This setting controls whether the file will be truncated when it is opened. For input-ports, the default value of this setting is #f. For output-ports, the default value of this setting is #t when the append: setting is #f, and #f otherwise.

For example:

```scheme
> (with-output-to-file
   (list path: "nofile"
         create: #f)
   (lambda ()
    (display "hello world!\n"))
*** ERROR IN (console)@1.1 -- No such file or directory
(with-output-to-file '(path: "nofile" create: #f) '#<procedure #2>)
```

```
(input-port-byte-position port [position [whence]]) procedure
(output-port-byte-position port [position [whence]]) procedure
```

When called with a single argument these procedures return the byte position where the next I/O operation would take place in the file attached to the given port (relative to the beginning of the file). When called with two or three arguments, the byte position for subsequent I/O operations on the given port is changed to position, which must be an exact integer. When whence is omitted or is 0, the position is relative to the beginning of the file. When whence is 1, the position is relative to the current byte position of the file. When whence is 2, the position is relative to the end of the file. The return value is the new byte position. On most operating systems the byte position for reading and writing of a given bidirectional port are the same.

When input-port-byte-position is called to change the byte position of an input-port, all input buffers will be flushed so that the next byte read will be the one at the given position.
When `output-port-byte-position` is called to change the byte position of an output-port, there is an implicit call to `force-output` before the position is changed.

For example:

```scheme
(define p ; p is an input-output-port
  (open-file 'path: "test" char-encoding: ISO-8859-1 create: maybe)))
(list (input-port-byte-position p) (output-port-byte-position p))
(0 0)
(display "abcdefghijklmnopqrstuvwxyz\n" p)
(list (input-port-byte-position p) (output-port-byte-position p))
(0 0)
(force-output p)
(list (input-port-byte-position p) (output-port-byte-position p))
(11 11)
(input-port-byte-position p 2)
2
(list (input-port-byte-position p) (output-port-byte-position p))
(2 2)
(peek-char p)
#\c
(list (input-port-byte-position p) (output-port-byte-position p))
(11 11)
(output-port-byte-position p -7 2)
4
(list (input-port-byte-position p) (output-port-byte-position p))
(4 4)
(write-char #\! p)
(list (input-port-byte-position p) (output-port-byte-position p))
(4 4)
(force-output p)
(list (input-port-byte-position p) (output-port-byte-position p))
(5 5)
(input-port-byte-position p 1)
1
(read p)
bcd!fghij
```

### 17.7.2 Process devices

```scheme
(open-process path-or-settings) ; procedure
(open-input-process path-or-settings) ; procedure
(open-output-process path-or-settings) ; procedure
(call-with-input-process path-or-settings proc) ; procedure
(call-with-output-process path-or-settings proc) ; procedure
(with-input-from-process path-or-settings thunk) ; procedure
(with-output-to-process path-or-settings thunk) ; procedure
```

All of these procedures start a new operating system process and create a bidirectional port which allows communication with that process on its standard input and standard output. The `direction` setting will default to the value `input` for the procedures `open-input-process`, `call-with-input-process` and `with-input-from-process`, to the value `output` for the procedures `open-output-process`, `call-with-output-process` and `with-output-to-process`, and to the value `input-output` for the procedure `open-process`. If the `direction`: 
setting is input, the output-port side is closed. If the direction: setting is output, the input-port side is closed.

The procedures open-process, open-input-process and open-output-process return the port that is created. The procedures call-with-input-process and call-with-output-process call the procedure proc with the port as single argument, and then return the value(s) of this call after closing the port and waiting for the process to terminate. The procedures with-input-from-process and with-output-to-process dynamically bind the current input-port and current output-port respectively to the port created for the duration of a call to the procedure thunk with no argument. The value(s) of the call to thunk are returned after closing the port and waiting for the process to terminate.

The first argument of this procedure is either a string denoting a filesystem path of an executable program or a list of port settings which must contain a path: setting. Here are the settings allowed in addition to the generic settings of byte-ports:

- **path: string**
  This setting indicates the location of the executable program in the filesystem. There is no default value for this setting.

- **arguments: list-of-strings**
  This setting indicates the string arguments that are passed to the program. The default value of this setting is the empty list (i.e. no arguments).

- **environment: list-of-strings**
  This setting indicates the set of environment variable bindings that the process receives. Each element of the list is a string of the form “VAR=VALUE”, where VAR is the name of the variable and VALUE is its binding. When list-of-strings is #f, the process inherits the environment variable bindings of the Scheme program. The default value of this setting is #f.

- **directory: dir**
  This setting indicates the current working directory of the process. When dir is #f, the process uses the value of (current-directory). The default value of this setting is #f.

- **stdin-redirection: (#f | #t)**
  This setting indicates how the standard input of the process is redirected. A setting of #t will redirect the standard input from the process-port (i.e. what is written to the process-port will be available on the standard input). A setting of #f will leave the standard input as-is, which typically results in input coming from the console. The default value of this setting is #t.

- **stdout-redirection: (#f | #t)**
  This setting indicates how the standard output of the process is redirected. A setting of #t will redirect the standard output to the process-port (i.e. all output to standard output can be read from the process-port). A setting of #f will leave the standard output as-is, which typically results in the output going to the console. The default value of this setting is #t.
• **stderr-redirection:** ( #f | #t )
  This setting indicates how the standard error of the process is redirected. A setting of #t will redirect the standard error to the process-port (i.e. all output to standard error can be read from the process-port). A setting of #f will leave the standard error as-is, which typically results in error messages being output to the console. The default value of this setting is #f.

• **pseudo-terminal:** ( #f | #t )
  This setting applies to UNIX. It indicates what type of device will be bound to the process’ standard input and standard output. A setting of #t will use a pseudo-terminal device (this is a device that behaves like a tty device even though there is no real terminal or user directly involved). A setting of #f will use a pair of pipes. The difference is important for programs which behave differently when they are used interactively, for example shells. The default value of this setting is #f.

• **show-console:** ( #f | #t )
  This setting applies to Microsoft Windows. It controls whether the process’ console window will be hidden or visible. The default value of this setting is #t (i.e. show the console window).

For example:

```scheme
> (with-input-from-process "date" read-line)
"Sun Jun 14 15:06:41 EDT 2009"
> (define p (open-process (list path: "ls" arguments: '("./examples")))))
> (read-line p)
"README"
> (read-line p)
"Xlib-simple"
> (close-port p)
> (define p (open-process "/usr/bin/dc")
> (display "2 100 \n" p)
> (force-output p)
> (read-line p)
"1267650600228229401496703205376"
```

(process-pid process-port)

This procedure returns the PID (Process Identifier) of the process of process-port. The PID is a small exact integer.

For example:

```scheme
> (let ((p (open-process "sort")))
   (process-pid p))
318
```

(process-status process-port [timeout [timeout-val]])

This procedure causes the current thread to wait until the process of process-port terminates (normally or not) or until the timeout is reached if timeout is supplied. If the timeout is reached, process-status returns timeout-val if it is supplied, otherwise an unterminated-process-exception object is raised. The procedure returns the process exit status as encoded by the operating system. Typically, if the process exited normally the return value is the process exit status multiplied by 256.
For example:

```scheme
> (let ((p (open-process "sort"))
   (for-each (lambda (x) (pretty-print x p))
     '(22 11 33))
   (close-output-port p)
   (let ((r (read-all p)))
     (close-input-port p)
     (list (process-status p) r)))
(0 (11 22 33))
```

### Unterminated-process-exception?

This procedure returns `#t` when `obj` is an unterminated-process-exception object and `#f` otherwise.

### Unterminated-process-exception-procedure

This procedure returns the procedure that raised `exc`.

### Unterminated-process-exception-arguments

This procedure returns the list of arguments of the procedure that raised `exc`.

For example:

```scheme
> (define (handler exc)
   (if (unterminated-process-exception? exc)
     (list (unterminated-process-exception-procedure exc)
           (unterminated-process-exception-arguments exc))
     'not-unterminated-process-exception))
> (with-exception-catcher
   handler
   (lambda ()
     (let ((p (open-process "sort")))
       (process-status p 1))))
```

### 17.7.3 Network devices

This procedure opens a network connection to a socket server and returns a tcp-client-port (a subtype of device-port) that represents this connection and allows communication with that server. The default value of the direction: setting is `input-output`, i.e. the Scheme program can send information to the server and receive information from the server. The sending direction can be “shutdown” using the `close-output-port` procedure and the receiving direction can be “shutdown” using the `close-input-port` procedure. The `close-port` procedure closes both directions of the connection.

The parameter of this procedure is an IP port number (16-bit nonnegative exact integer), a string of the form "HOST:PORT" or a list of port settings. When the parameter is the number `PORT` it is handled as if it was the setting `port-number`:
PORT. When the parameter is the string "HOST:PORT" it is handled as if it was the setting server-address: "HOST:PORT".

Here are the settings allowed in addition to the generic settings of byte-ports:

- **server-address: string-or-ip-address**
  This setting indicates the internet address of the server, and possibly the IP port number. When this parameter is not specified or is "", the connection requests are sent to the loopback interface (with IP address 127.0.0.1). The parameter can be a string denoting a host name, which will be translated to an IP address by the host-info procedure, or a 4 element u8vector which contains the 32-bit IPv4 address or an 8 element u16vector which contains the 128-bit IPv6 address. A string of the form "HOST:PORT" is handled as if it was the combination of settings server-address: "HOST" port-number: PORT.

- **port-number: 16-bit-exact-integer**
  This setting indicates the IP port number of the server to connect to (e.g. 80 for the standard HTTP server, 23 for the standard telnet server). There is no default value for this setting.

- **keep-alive: ( #f | #t )**
  This setting controls the use of the “keep alive” option on the connection. The “keep alive” option will periodically send control packets on otherwise idle network connections to ensure that the server host is active and reachable. The default value of this setting is #f.

- **coalesce: ( #f | #t )**
  This setting controls the use of TCP’s “Nagle algorithm” which reduces the number of small packets by delaying their transmission and coalescing them into larger packets. A setting of #t will coalesce small packets into larger ones. A setting of #f will transmit packets as soon as possible. The default value of this setting is #t. Note that this setting does not affect the buffering of the port.

Below is an example of the client-side code that opens a connection to an HTTP server on port 8080 of the loopback interface (with IP address 127.0.0.1). For the server-side code see the example for the procedure open-tcp-server.

```scheme
> (define p (open-tcp-client (list port-number: 8080
eol-encoding: 'cr-lf)))
> p
#<input-output-port #2 (tcp-client #u8(127 0 0 1) 8080)>
> (display "GET /
"
p)
> (force-output p)
> (read-line p)
"<HTML>"
```

(open-tcp-server port-number-or-address-or-settings) procedure

This procedure sets up a socket to accept network connection requests from clients and returns a tcp-server-port from which network connections to clients are obtained. Tcp-server-ports are a direct subtype of object-ports (i.e. they are not character-ports) and are input-ports. Reading from a tcp-server-port with the read procedure will block until a network connection request is received from a client. The read procedure will then return a tcp-client-port (a subtype of device-port) that represents
this connection and allows communication with that client. Closing a tcp-server-port with either the close-input-port or close-port procedures will cause the network subsystem to stop accepting connections on that socket.

The parameter of this procedure is an IP port number (16-bit nonnegative exact integer), a string of the form "INTF:PORT" or a list of port settings which must contain a port-number: setting. When the parameter is the number PORT it is handled as if it was the setting port-number: PORT. When the parameter is the string "INTF:PORT" it is handled as if it was the setting server-address: "INTF:PORT".

Below is a list of the settings allowed in addition to the settings keep-alive: and coalesce: allowed by the open-tcp-client procedure and the generic settings of byte-ports. The settings which are not listed below apply to the tcp-client-port that is returned by read when a connection is accepted and have the same meaning as if they were used in a call to the open-tcp-client procedure.

- **server-address: string-or-ip-address**
  This setting indicates the internet address of the network interface on which connections requests are accepted, and possibly the IP port number. When this parameter is not specified or is "", the connection requests are accepted only on the loopback interface (with IP address 127.0.0.1). When this parameter is "*", the connection requests are accepted on all network interfaces (i.e. address INADDR_ANY). The parameter can be a string denoting a host name, which will be translated to an IP address by the host-info procedure, or a 4 element u8vector which contains the 32-bit IPv4 address or an 8 element u16vector which contains the 128-bit IPv6 address. A string of the form "INTF:PORT" is handled as if it was the combination of settings server-address: "INTF" port-number: PORT.

- **port-number: 16-bit-exact-integer**
  This setting indicates the IP port number assigned to the socket which accepts connection requests from clients. So called “well-known ports”, which are reserved for standard services, have a port number below 1024 and can only be assigned to a socket by a process with superuser privileges (e.g. 80 for the HTTP service, 23 for the telnet service). No special privileges are needed to assign higher port numbers to a socket. The special value 0 requests that a currently unused port number be assigned to the socket (the port number assigned can be retrieved using the procedure tcp-server-socket-info). There is no default value for this setting.

- **backlog: positive-exact-integer**
  This setting indicates the maximum number of connection requests that can be waiting to be accepted by a call to read (technically it is the value passed as the second argument of the UNIX listen() function). The default value of this setting is 128.

- **reuse-address: ( #f | #t )**
  This setting controls whether it is possible to assign a port number that is currently active. Note that when a server process terminates, the socket it was using to accept connection requests does not become inactive immediately. Instead it
remains active for a few minutes to ensure clean termination of the connections. A setting of \#f will cause an exception to be raised in that case. A setting of \#t will allow a port number to be used even if it is active. The default value of this setting is \#t.

Below is an example of the server-side code that accepts connections on port 8080 of any network interface. For the client-side code see the example for the procedure open-tcp-client.

```scheme
(define s (open-tcp-server (list server-address: "*" port-number: 8080 eol-encoding: 'cr-lf)))
(define p (read s)) ; blocks until client connects
(p)  ;<input-output-port #2 (tcp-client 8080)>
(read-line p) "GET /
(display "<HTML>
(force-output p)
```

The procedure tcp-service-register! sets up a socket to accept network connection requests from clients and creates a “service” thread which processes the incoming connections. The parameter port-number-or-address-or-settings has the same meaning as for the procedure open-tcp-server.

For each connection established the service thread creates a “handler” thread which executes a call to the procedure thunk with no argument. The handler thread’s current input-port and current output-port are both set to the tcp-client-port created for the connection. There is no need for the thunk to close the tcp-client-port, as this is done by the handler thread when the thunk returns normally.

The procedure tcp-service-unregister! terminates the service thread which was registered by tcp-service-register! with the same network interface and port number (if a service thread is still registered). The procedure tcp-service-register! implicitly calls tcp-service-unregister! before registering the new service thread.

```scheme
(tcp-service-register! port-number-or-address-or-settings thunk) procedure
(tcp-service-unregister! port-number-or-address-or-settings) procedure
```

### 17.8 Directory-ports

(open-directory path-or-settings) procedure

This procedure opens a directory of the filesystem for reading its entries and returns a directory-port from which the entries can be enumerated. Directory-ports are a direct subtype of object-ports (i.e. they are not character-ports) and are input-ports.
Reading from a directory-port with the read procedure returns the next file name in the directory as a string. The end-of-file object is returned when all the file names have been enumerated. Another way to get the list of all files in a directory is the directory-files procedure which returns a list of the files in the directory. The advantage of using directory-ports is that it allows iterating over the files in a directory in constant space, which is interesting when the number of files in the directory is not known in advance and may be large. Note that the order in which the names are returned is operating-system dependent.

The parameter of this procedure is either a string denoting a filesystem path to a directory or a list of port settings which must contain a path: setting. Here are the settings allowed in addition to the generic settings of object-ports:

- **path: string**
  This setting indicates the location of the directory in the filesystem. There is no default value for this setting.

- **ignore-hidden: ( #f | #t | dot-and-dot-dot )**
  This setting controls whether hidden-files will be returned. Under UNIX and Mac OS X hidden-files are those that start with a period (such as '.', '..', and '.profile'). Under Microsoft Windows hidden files are the '.' and '..' entries and the files whose “hidden file” attribute is set. A setting of #f will enumerate all the files. A setting of #t will only enumerate the files that are not hidden. A setting of dot-and-dot-dot will enumerate all the files except for the '.' and '..' hidden files. The default value of this setting is #t.

For example:

```lisp
> (let ((p (open-directory (list path: "./examples" ignore-hidden: #f))))
  (let loop ()
    (let ((fn (read p)))
      (if (string? fn)
        (begin
          (pp (path-expand fn))
          (loop))))))
"/u/feeley/examples/.
"/u/feeley/examples/..
"/u/feeley/examples/complex"
"/u/feeley/examples/README"
"/u/feeley/examples/simple"
> (define x (open-directory "/examples")
> (read-all x)
"complex" "README" "simple")
```

### 17.9 Vector-ports

- **(open-vector [vector-or-settings])** procedure
- **(open-input-vector [vector-or-settings])** procedure
- **(open-output-vector [vector-or-settings])** procedure
- **(call-with-input-vector vector-or-settings proc)** procedure
- **(call-with-output-vector vector-or-settings proc)** procedure
- **(with-input-from-vector vector-or-settings thunk)** procedure
Vector-ports represent streams of Scheme objects. They are a direct subtype of object-ports (i.e. they are not character-ports). All of these procedures create vector-ports that are either unidirectional or bidirectional. The direction: setting will default to the value input for the procedures open-input-vector, call-with-input-vector and with-input-from-vector, to the value output for the procedures open-output-vector, call-with-output-vector and with-output-to-vector, and to the value input-output for the procedure open-vector. Bidirectional vector-ports behave like FIFOs: data written to the port is added to the end of the stream that is read. It is only when a bidirectional vector-port’s output-side is closed with a call to the close-output-port procedure that the stream’s end is known (when the stream’s end is reached, reading the port returns the end-of-file object).

The procedures open-vector, open-input-vector and open-output-vector return the port that is created. The procedures call-with-input-vector and call-with-output-vector create a vector port, call the procedure proc with the port as single argument and then close the port. The procedures with-input-from-vector and with-output-to-vector create a vector port, dynamically bind the current input-port and current output-port respectively to the port created for the duration of a call to the procedure thunk with no argument, and then close the port. The procedures call-with-input-vector and with-input-from-vector return the value returned by the procedures proc and thunk respectively. The procedures call-with-output-vector and with-output-to-vector return the vector accumulated in the port (see get-output-vector).

The first parameter of these procedures is either a vector of the elements used to initialize the stream or a list of port settings. If it is not specified, the parameter of the open-vector, open-input-vector, and open-output-vector procedures defaults to an empty list of port settings. Here are the settings allowed in addition to the generic settings of object-ports:

- **init**: vector
  This setting indicates the initial content of the stream. The default value of this setting is an empty vector.

- **permanent-close**: ( #f | #t )
  This setting controls whether a call to the procedures close-output-port will close the output-side of a bidirectional vector-port permanently or not. A permanently closed bidirectional vector-port whose end-of-file has been reached on the input-side will return the end-of-file object for all subsequent calls to the read procedure. A non-permanently closed bidirectional vector-port will return to its opened state when its end-of-file is read. The default value of this setting is #t.

For example:

```scheme
  > (define p (open-vector))
  > (write 1 p)
  > (write 2 p)
  > (write 3 p)
```
Chapter 17: I/O and ports

> (force-output p)
> (read p)
1
> (read p)
2
> (close-output-port p)
> (read p)
3
> (read p)
#!eof
> (with-output-to-vector '() (lambda () (write 1) (write 2)))
#(1 2)

(open-vector-pipe [vector-or-settings1 [vector-or-settings2]]) procedure
The procedure open-vector-pipe creates two vector-ports and returns these two ports. The two ports are interrelated as follows: the first port's output-side is connected to the second port's input-side and the first port's input-side is connected to the second port's output-side. The value vector-or-settings1 is used to setup the first vector-port and vector-or-settings2 is used to setup the second vector-port. The same settings as for open-vector are allowed. The default direction: setting is input-output (i.e. a bidirectional port is created). If it is not specified vector-or-settings1 defaults to the empty list. If it is not specified vector-or-settings2 defaults to vector-or-settings1 but with the init: setting set to the empty vector and with the input and output settings exchanged (e.g. if the first port is an input-port then the second port is an output-port, if the first port's input-side is non-buffered then the second port's output-side is non-buffered).

For example:

> (define (server op)
   (receive (c s) (open-vector-pipe) ; client-side and server-side ports
   (thread-start!
      (make-thread
         (lambda ()
            (let loop ()
               (let ((request (read s)))
                 (if (not (eof-object? request))
                   (begin
                      (write (op request) s)
                      (newline s)
                      (force-output s)
                      (loop))))))))
   c))
> (define a (server (lambda (x) (expt 2 x))))
> (define b (server (lambda (x) (expt 10 x))))
> (write 100 a)
> (write 30 b)
> (read a)
126765060022829401496703205376
> (read b)
1000000000000000000000000000000

(get-output-vector vector-port) procedure
The procedure get-output-vector takes an output vector-port or a bidirectional vector-port as parameter and removes all the objects currently on the output-side,
Chapter 17: I/O and ports

returning them in a vector. The port remains open and subsequent output to the port and calls to the procedure `get-output-vector` are possible.

For example:

```plaintext
> (define p (open-vector '#(1 2 3)))
> (write 4 p)
> (get-output-vector p)
#(1 2 3 4)
> (write 5 p)
> (write 6 p)
> (get-output-vector p)
#(5 6)
```

17.10 String-ports

`object->string obj [n]`  procedure

This procedure converts the object `obj` to its external representation and returns it in a string. The parameter `n` specifies the maximal width of the resulting string. If the external representation is wider than `n`, the resulting string will be truncated to `n` characters and the last 3 characters will be set to periods. Note that the current readtable is used.
For example:

```lisp
> (object->string (expt 2 100))
"1267650600228229401496703205376"
> (object->string (expt 2 100) 30)
"126765060022822940149670320..."
> (object->string (cons car cdr))
"(#<procedure #2 car> . #<procedure #3 cdr>)"
```

### 17.11 U8vector-ports

- `(open-u8vector [u8vector-or-settings])`: procedure
- `(open-input-u8vector [u8vector-or-settings])`: procedure
- `(open-output-u8vector [u8vector-or-settings])`: procedure
- `(call-with-input-u8vector u8vector-or-settings proc)`: procedure
- `(call-with-output-u8vector u8vector-or-settings proc)`: procedure
- `(with-input-from-u8vector u8vector-or-settings thunk)`: procedure
- `(with-output-to-u8vector u8vector-or-settings thunk)`: procedure
- `(open-u8vector-pipe [u8vector-or-settings1 [u8vector-or-settings2]])`: procedure
- `(get-output-u8vector u8vector-port)`: procedure

U8vector-ports represent streams of bytes. They are a direct subtype of byte-ports. These procedures are the u8vector-port analog of the procedures specified in the vector-ports section.

For example:

```lisp
> (define p (open-u8vector))
> (write 1 p)
> (write 2 p)
> (write 3 p)
> (force-output p)
> (read-u8 p)
49
> (read-u8 p)
50
> (close-output-port p)
> (read-u8 p)
51
> (read-u8 p)
#eof
> (with-output-to-u8vector '() (lambda () (write 1) (write 2)))
#u8(49 50)
```

### 17.12 Other procedures related to I/O

- `(current-input-port [new-value])`: procedure
- `(current-output-port [new-value])`: procedure
- `(current-error-port [new-value])`: procedure
- `(current-readtable [new-value])`: procedure

These procedures are parameter objects which represent respectively: the current input-port, the current output-port, the current error-port, and the current readtable.

- `(print [port: port] obj...)`: procedure
(println [port: port] obj...) procedure

The print procedure writes a representation of each obj, from left to right, to port. When a compound object is encountered (pair, list, vector, box) the elements of that object are recursively written without the surrounding tokens (parentheses, spaces, dots, etc). Strings, symbols, keywords and characters are written like the display procedure. If there are more than one obj, the first obj must not be a keyword object. If it is not specified, port defaults to the current output-port. The procedure print returns an unspecified value.

The println procedure does the same thing as the print procedure and then writes an end of line to port.

For example:

> (println "2*2 is " (* 2 2) " and 2+2 is " (+ 2 2))
2*2 is 4 and 2+2 is 4
> (define x (list "<i>" (list "<tt>" 123 "</tt>")) "</i>")
> (println x)
<i><tt>123</tt></i>
> (define p (open-output-string))
> (print port: p 1 \2 "345")
> (get-output-string p)
"12345"
18 Lexical syntax and readtables

18.1 Readtables

Readtables control the external textual representation of Scheme objects, that is the encoding of Scheme objects using characters. Readtables affect 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, print, println, pretty-print, and pp, and the procedure used by the REPL to print results). To preserve write/read invariance the printer and reader must be using compatible readtables. 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.

Readtables are immutable records whose fields specify various textual representation aspects. There are accessor procedures to retrieve the content of specific fields. There are also functional update procedures that create a copy of a readtable, with a specific field set to a new value.

(readtable? obj) procedure

This procedure returns #t when obj is a readtable and #f otherwise.

For example:

> (readtable? (current-readtable))
#t
> (readtable? 123)
#f

(readtable-case-conversion? readtable) procedure

(readtable-case-conversion?-set readtable new-value) procedure

The procedure readtable-case-conversion? returns the content of the ‘case-conversion?’ field of readtable. When the content of this field is #f, the reader preserves the case of symbols and keyword objects that are read (i.e. Ice and ice are distinct symbols). When the content of this field is the symbol upcase, the reader converts lowercase letters to uppercase when reading symbols and keywords (i.e. Ice is read as the symbol (string->symbol "ICE")) . Otherwise the reader converts uppercase letters to lowercase when reading symbols and keywords (i.e. Ice is read as the symbol (string->symbol "ice")).

The procedure readtable-case-conversion?-set returns a copy of readtable where only the ‘case-conversion?’ field has been changed to new-value.

For example:

> (output-port-readtable-set! (repl-output-port)
  (readtable-case-conversion?-set (output-port-readtable (repl-output-port)) #f))
> (output-port-readtable-set! (repl-output-port)
  (readtable-case-conversion?-set (output-port-readtable (repl-output-port)) #f))
> 'Ice
The procedure readtable-keywords-allowed? returns the content of the ‘keywords-allowed?’ field of readtable. When the content of this field is #f, the reader does not recognize keyword objects (i.e. :foo and foo: are read as the symbols (string->symbol "foo") and (string->symbol "foo:"), respectively). When the content of this field is the symbol prefix, the reader recognizes keyword objects that start with a colon, as in Common Lisp (i.e. :foo is read as the keyword (string->keyword "foo")). Otherwise the reader recognizes keyword objects that end with a colon, as in DSSSL (i.e. foo: is read as the symbol (string->symbol "foo")).

The procedure readtable-keywords-allowed?-set returns a copy of readtable where only the ‘keywords-allowed?’ field has been changed to new-value. For example:

```
> (input-port-readtable-set! (repl-input-port)
  (readtable-keywords-allowed?-set (input-port-readtable (repl-input-port)) #f))
> (map keyword? '(foo :foo foo:))
(#f #f #f)
> (input-port-readtable-set! (repl-input-port)
  (readtable-keywords-allowed?-set (input-port-readtable (repl-input-port)) #t))
> (map keyword? '(foo :foo foo:))
(#f #t #t)
> (input-port-readtable-set! (repl-input-port)
  (readtable-keywords-allowed?-set (input-port-readtable (repl-input-port)) 'prefix))
> (map keyword? '(foo :foo foo:))
(#f #t #f)
```

(readtable-sharing-allowed? readtable) procedure

(readtable-keywords-allowed?-set readtable new-value) procedure
The procedure `readtable-sharing-allowed?` returns the content of the
‘sharing-allowed?’ field of `readtable`. The reader recognizes the `#n#` and
`#n=datum` notation for circular structures and the printer uses this notation if
and only if the content of the ‘sharing-allowed?’ field is not `#f`. Moreover
when the content of the ‘sharing-allowed?’ field is the symbol `serialize`,
the printer uses a special external representation that the reader understands and
that extends write/read invariance to the following types: records, procedures and
continuations. Note that an object can be serialized and deserialized if and only if
all of its components are serializable.

The procedure `readtable-sharing-allowed?-set` returns a copy of `readtable
where only the ‘sharing-allowed?’ field has been changed to `new-value`.

Here is a simple example:

```scheme
> (define (wr obj allow?)
  (call-with-output-string
   "()"
   (lambda (p)
    (output-port-readtable-set!
     p
     (readtable-sharing-allowed?-set
      (output-port-readtable p)
      allow?)))
    (write obj p)))
> (define (rd str allow?)
  (call-with-input-string str
   (lambda (p)
    (input-port-readtable-set!
     p
     (readtable-sharing-allowed?-set
      (input-port-readtable p)
      allow?)))
    (read p)))
> (define x (list 1 2 3))
> (set-car! (cdr x) (cddr x))
> (wr x #f)
"(1 (3) 3)"
> (wr x #t)
"(1 #0=(3) . #0#)"
> (define y (rd (wr x #t) #t))
> y
(1 (3) 3)
> (eq? (cadr y) (cddr y))
#t
> (define f #f)
> (let ((free (expt 2 10)))
  (set! f (lambda (x) (+ x free))))
> (define s (wr f 'serialize))
> (string-length s)
4196
> (define g (rd s 'serialize))
> (eq? f g)
#f
> (g 4)
1028
Continuations are tricky to serialize because they contain a dynamic environment and this dynamic environment may contain non-serializable objects, in particular ports attached to operating-system streams such as files, the console or standard input/output. Indeed, all dynamic environments contain a binding for the current-input-port and current-output-port. Moreover, any thread that has started a REPL has a continuation which refers to the repl-context object in its dynamic environment. A repl-context object contains the interaction channel, which is typically connected to a non-serializable port, such as the console. Another problem is that the parameterize form saves the old binding of the parameter in the continuation, so it is not possible to eliminate the references to these ports in the continuation by using the parameterize form alone.

Serialization of continuations can be achieved dependably by taking advantage of string-ports, which are serializable objects (unless there is a blocked thread), and the following features of threads: they inherit the dynamic environment of the parent thread and they start with an initial continuation that contains only serializable objects. So a thread created in a dynamic environment where current-input-port and current-output-port are bound to a dummy string-port has a serializable continuation.

Here is an example where continuations are serialized:

```scheme
> (define (wr obj)
   (call-with-output-string
    '()
    (lambda (p)
     (output-port-readtable-set! p
      (readtable-sharing-allowed?-set
       (output-port-readtable p)
       'serialize))
     (write obj p))))
> (define (rd str)
   (call-with-input-string
    str
    (lambda (p)
     (input-port-readtable-set!
      p
      (readtable-sharing-allowed?-set
       (input-port-readtable p)
       'serialize))
     (read p))))
> (define fifo (open-vector))
> (define (suspend-and-die!)
   (call-with-current-continuation
    (lambda (k)
     (write (wr k) fifo)
     (newline fifo)
     (force-output fifo)
     (thread-terminate! (current-thread)))))
> (let ((dummy-port (open-string)))
   (parameterize ((current-input-port dummy-port)
                  (current-output-port dummy-port))
     (thread-start!
      (make-thread
       (lambda ()
        (suspend-and-die!))))))
```
Chapter 18: Lexical syntax and readtables

(* 100
  (suspend-and-die!)))
)

#<thread #2>
> (define s (read fifo))
> (thread-join!
  (thread-start!
   (make-thread
    (lambda ()
      ((rd s) 111)))))
 11100
> (thread-join!
  (thread-start!
   (make-thread
    (lambda ()
      ((rd s) 222)))))
 22200
> (string-length s)
13114

(readtable-eval-allowed? readtable) procedure
(readtable-eval-allowed?-set readtable new-value) procedure

The procedure readtable-eval-allowed? returns the content of the
‘eval-allowed?’ field of readtable. The reader recognizes the #.expression
notation for read-time evaluation if and only if the content of the ‘eval-allowed?’
field is not #f.

The procedure readtable-eval-allowed?-set returns a copy of readtable
where only the ‘eval-allowed?’ field has been changed to new-value.

For example:

> (input-port-readtable-set!
   (repl-input-port)
   (readtable-eval-allowed?-set
    (input-port-readtable (repl-input-port))
    #t))
> ’(5 plus 7 is #.(+ 5 7))
(5 plus 7 is 12)
> ’(buf = #.(make-u8vector 25))
(buf = #u8(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))

(readtable-write-cdr-read-macros? readtable) procedure
(readtable-write-cdr-read-macros?-set readtable new-value) procedure

(readtable-write-extended-read-macros? readtable) procedure
(readtable-write-extended-read-macros?-set readtable new-value) procedure

The procedure readtable-write-cdr-read-macros? returns the content
of the ‘write-cdr-read-macros?’ field of readtable. The procedure
readtable-write-extended-read-macros? returns the content of the
‘write-extended-read-macros?’ field of readtable.

At all times the printer uses read-macros in its output for datums of the form
(quote datum), (quasiquote datum), (unquote datum), and (unquote-
splicing datum). That is the following read-macro notations will be used respectively: ‘datum, ‘datum, ,datum, and ,@datum. Moreover, normally the read-
macros will not be used when the form appears in the cdr of a list, for example (foo quote bar), (foo . (quote bar)) and (foo . ’bar) will all be printed as (foo quote bar).

When the content of the ‘write-cdr-read-macros?’ field is not #f, the printer will use read-macros when the forms appear in the cdr of a list. For example (foo quote bar) will be printed as (foo . ’bar). When the content of the ‘write-extended-read-macros?’ field is not #f, the printer will also use extended read-macros, for example #’datum in place of (syntax datum).

The procedure readable-write-cdr-read-macros?-set returns a copy of readtable where only the ‘write-cdr-read-macros?’ field has been changed to new-value. The procedure readable-write-extended-read-macros?-set returns a copy of readtable where only the ‘write-extended-read-macros?’ field has been changed to new-value.

For example:

```lisp
> (output-port-readtable-set! (repl-output-port)
  (readtable-write-extended-read-macros?-set (output-port-readtable (repl-output-port)) #t))
> ’(foo (syntax bar))
(foo #’bar)
> ’(foo syntax bar)
(foo syntax bar)
> (output-port-readtable-set! (repl-output-port)
  (readtable-write-cdr-read-macros?-set (output-port-readtable (repl-output-port)) #t))
> ’(foo syntax bar)
(foo . #’bar)
```

The procedure readable-max-write-level returns the content of the ‘max-write-level’ field of readtable. The printer will display an ellipsis for the elements of lists and vectors that are nested deeper than this level.

The procedure readable-max-write-level-set returns a copy of readtable where only the ‘max-write-level’ field has been changed to new-value, which must be an nonnegative fixnum.

For example:

```lisp
> (define (wr obj n)
   (call-with-output-string ’()
     (lambda (p)
       (output-port-readtable-set!
         p
         (readtable-max-write-level-set
           (output-port-readtable p)
           n))
       (write obj p))))
> (wr ’(a #(b (c c) #u8(9 9 9) b) a) 3)
```
Chapter 18: Lexical syntax and readtables

(readtable-max-write-length readtable)  procedure
=readtable-max-write-length (readtable readtable)  procedure

The procedure readtable-max-write-length returns the content of the
'max-write-length' field of readtable. The printer will display an ellipsis for the
elements of lists and vectors that are at an index beyond that length.

The procedure readtable-max-write-length-set returns a copy of readtable
where only the 'max-write-length' field has been changed to new-value, which
must be an nonnegative fixnum.

For example:

> (define (wr obj n)
  (call-with-output-string
   '()
   (lambda (p)
     (output-port-readtable-set!
      p
      (readtable-max-write-length-set
       (output-port-readtable p)
       n))
     (write obj p))))

> (wr 'a #(b (c c) #u8(9 9 9) b) . a) 4)
"(a #(b (c c) #u8(9 9 9) b) . a)"
> (wr 'a #(b (c c) #u8(9 9 9) b) . a) 3)
"(a #(b (c c) #u8(9 9 9) ...) . a)"
> (wr 'a #(b (c c) #u8(9 9 9) b) . a) 2)
"(a #(b (c c) #u8(9 9 9) ...) . a)"
> (wr 'a #(b (c c) #u8(9 9 9) b) . a) 1)
"(a ...)
> (wr 'a #(b (c c) #u8(9 9 9) b) . a) 0)
"(...)"

(readtable-max-unescaped-char readtable)  procedure
=readtable-max-unescaped-char (readtable readtable)  procedure

The procedure readtable-max-unescaped-char returns the content of the
'max-unescaped-char' field of readtable. The printer will display using an escape
sequence any character within symbols, strings and character objects greater than
'max-unescaped-char'

The procedure readtable-max-unescaped-char-set returns a copy of readtable
where only the 'max-unescaped-char' field has been changed to new-value, which
must be a character.

For example:

> (define rt (output-port-readtable (repl-output-port)))
> (readtable-max-unescaped-char rt)
Chapter 18: Lexical syntax and readtables

18.2 Boolean syntax

Booleans are required to be followed by a delimiter (i.e. #f64() is not the boolean #f followed by the number 64 and the empty list).

18.3 Character syntax

Characters are required to be followed by a delimiter (i.e. \spaceballs is not the character \space followed by the symbol balls). The lexical syntax of characters is extended to allow the following:

#\nul  Unicode character 0
#\alarm  Unicode character 7
18.4 String syntax

The lexical syntax of quoted strings is extended to allow the following escape codes:

\a Unicode character 7
\b Unicode character 8
\t Unicode character 9
\n Unicode character 10 (newline character)
\v Unicode character 11
\f Unicode character 12
\r Unicode character 13
\" "
\\ \n\| |
\? ?
\ooo character encoded in octal (1 to 3 octal digits, first digit must be less than 4 when there are 3 octal digits)
\xhh character encoded in hexadecimal (>= 1 hexadecimal digit)
\uhhhh character encoded in hexadecimal (exactly 4 hexadecimal digits)
\Uhhhhhhhh character encoded in hexadecimal (exactly 8 hexadecimal digits)
\<space> Unicode character 32 (space character)
This sequence expands to nothing (it is useful for splitting a long string literal on multiple lines while respecting proper indentation of the source code)

Gambit also supports a “here string” syntax that is similar to shell “here documents”. For example:

```scheme
> (pretty-print #\<<THE-END
  hello
  world
  THE-END
)
"hello\nworld"
```

The here string starts with the sequence ‘#<<’. The part of the line after the ‘#<<’ up to and including the newline character is the key. The first line afterward that matches the key marks the end of the here string. The string contains all the characters between the start key and the end key, with the exception of the newline character before the end key.

18.5 Symbol syntax

The lexical syntax of symbols is extended to allow a leading and trailing vertical bar (e.g. |a\b"c:|). 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.

For example:

```scheme
> (symbol->string ' |a\b"c:|)
"a\b"c:
```

18.6 Keyword syntax

The lexical syntax of keywords is like symbols, but with a colon at the end (note that this can be changed to a leading colon by setting the ‘keywords-allowed?’ field of the readtable to the symbol prefix). A colon by itself is not a keyword, it is a symbol. Vertical bars can be used like symbols but the colon must be outside the vertical bars. Note that the string returned by the keyword->string procedure does not include the colon.

For example:

```scheme
> (keyword->string 'foo:)
"foo"
> (map keyword? '(|ab()cd:| |ab()cd|: : ::))
(#f #t #f #t)
```

18.7 Box syntax

The lexical syntax of boxes is #&obj where obj is the content of the box.

For example:

```scheme
> (list '#&"hello" '#&123)
("hello" #&123)
> (box (box (+ 10 20)))
#&#&30
```
18.8 Number syntax
The lexical syntax of the special inexact real numbers is as follows:
+inf.0    positive infinity
-inf.0    negative infinity
+nan.0    "not a number"
-0.       negative zero (‘0.’ is the positive zero)

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

18.10 Special #! syntax
The lexical syntax of the special #! objects is as follows:
#!eof           end-of-file object
#!void          void object
#!optional      optional object
#!rest          rest object
#!key           key object

18.11 Multiline comment syntax
Multiline comments are delimited by the tokens ‘#!’ and ‘!#’. These comments can be
nested.

18.12 Scheme infix syntax extension
The reader supports an infix syntax extension which is called SIX (Scheme Infix eXtension).
This extension is both supported by the ‘read’ procedure and in program source code.
   The backslash character is a delimiter that marks the beginning of a single datum ex-
pressed in the infix syntax (the details are given below). One way to think about it is that
the backslash character escapes the prefix syntax temporarily to use the infix syntax. For example, a three element list could be written as \((X \\backslash Y Z)\), the elements \(X\) and \(Z\) are expressed using the normal prefix syntax and \(Y\) is expressed using the infix syntax.

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

### 18.12.1 SIX grammar

The SIX grammar is given below. On the left hand side are the production rules. On the right hand side is the datum that is constructed by the reader. The notation \$$i$$ denotes the datum that is constructed by the reader for the \(i\)th part of the production rule.

\[
\text{<infix datum> ::=} \\
\text{<stat> ::=} \\
\text{<if stat> ::=} \\
\text{<for stat> ::=} \\
\text{<while stat> ::=} \\
\text{<do stat> ::=} \\
\text{<label stat> ::=} \\
\text{<goto stat> ::=} \\
\text{<return stat> ::=} \\
\text{<expression stat> ::=} \\
\text{<procedure definition> ::=} \\
\text{<variable definition> ::=} \\
\text{<clause stat> ::=} \\
\text{<compound stat> ::=} \\
\text{; (six.compound)}
\]

\[
\text{<if stat> ::=} \\
\text{\quad if ( <pexpr> ) <stat> (six.if $3$ $5$)} \\
\text{\quad | if ( <pexpr> ) <stat> else <stat> (six.if $3$ $5$ $7$)}
\]

\[
\text{<for stat> ::=} \\
\text{\quad for ( <stat> ; <oexpr> ; <oexpr> ) <stat> (six.for $3$ $5$ $7$ $9$)}
\]

\[
\text{<while stat> ::=} \\
\text{\quad while ( <pexpr> ) <stat> (six.while $3$ $5$)}
\]

\[
\text{<do stat> ::=} \\
\text{\quad do <stat> while ( <pexpr> ) ; (six.do-while $2$ $5$)}
\]
\[
\begin{align*}
\text{<switch stat> ::=} & \quad \text{switch ( <pexpr> ) <stat>} \\
\text{<case stat> ::=} & \quad \text{case <expr> : <stat>} \\
\text{<break stat> ::=} & \quad \text{break ;} \\
\text{<continue stat> ::=} & \quad \text{continue ;} \\
\text{<label stat> ::=} & \quad \text{<identifier> : <stat>} \\
\text{<goto stat> ::=} & \quad \text{goto <expr> ;} \\
\text{<return stat> ::=} & \quad \text{return ;} \\
\text{<expression stat> ::=} & \quad \text{<expr> ;} \\
\text{<clause stat> ::=} & \quad \text{<expr> .} \\
\text{<pexpr> ::=} & \quad \text{<procedure definition>} \\
& \quad \text{| <variable definition>} \\
& \quad \text{| <expr>} \\
\text{<procedure definition> ::=} & \quad \text{<type> <id-or-prefix> ( <parameters> ) <body>} \\
\text{<variable definition> ::=} & \quad \text{<type> <id-or-prefix> <dimensions> <iexpr>} \\
\text{<iexpr> ::=} & \quad = \text{<expr>} \\
& \quad #f \\
\text{<dimensions> ::=} & \quad [ <expr> ] <dimensions> \\
& \quad () \\
\text{<oexpr> ::=} & \quad \text{<expr>} \\
& \quad #f \\
\text{<expr> ::=}
\end{align*}
\]
<expr18> ::= $1
<expr17> ::=
<expr17> :- <expr18> (six.x:-y $1 $3)
<expr17> ::=
<expr16> , <expr16> ($1 $3)
<expr16> ::= <expr15>
<expr15> := <expr16> (six.x:=y $1 $3)
<expr15> ::= <expr14>
<expr14> %= <expr15> (six.x%=y $1 $3)
<expr14> ::= <expr13>
<expr13> : <expr14> (six.x:y $1 $3)
<expr13> ::= <expr12> ? <expr> : <expr13> (six.x?y:z $1 $3 $5)
<expr12> ::= <expr12> | <expr11> ($1 $3)
<expr11> ::= <expr10> & & <expr10> (six.x&&y $1 $3)
<expr10> ::= <expr10> | <expr9> ($1 $3)
<expr9> ::= <expr9> ^ <expr8> (six.x^y $1 $3)
<expr9> ::= <expr8> ($1 $3)
<expr8> ::=  
  <expr8> & <expr7>  
  | <expr7>  
(six.x&y $1 $3)  

$1

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

$1

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

$1

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

$1

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

$1

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

$1

<expr2> ::=  
  & <expr2>  
  | + <expr2>  
  | - <expr2>  
  | * <expr2>  
  | ! <expr2>  
  | ++ <expr2>  
  | -- <expr2>  
  | ~ <expr2>  
  | new <id-or-prefix> ( <arguments> )  
  | <expr1>  
(six.&x $2)  

(six.+x $2)  

(six.-x $2)  

(six.*x $2)  

(six.!x $2)  

(six.+x $2)  

(six.-x $2)  

(six.^x $2)  

(six.new $2 . $4)  

$1

<expr1> ::=  
  <expr1> ++  
  | <expr1> --  
  | <expr1> ( <arguments> )  
  | <expr1> [ <expr> ]  
(six.x++ $1)  

(six.x-- $1)  

(six.call $1 . $3)  

(six.index $1 $3)
\[
\begin{align*}
\text{expr1} & \rightarrow \text{id-or-prefix} & \text{(six.arrow } \$1 \ $3) \\
\text{expr1} & . \ \text{id-or-prefix} & \text{(six.dot } \$1 \ $3) \\
\text{expr0} & & \text{($1)} \\
\text{expr0} & ::= \\
\text{id-or-prefix} & & \text{($1)} \\
\text{string} & & \text{(six.literal } \$1) \\
\text{char} & & \text{(six.literal } \$1) \\
\text{number} & & \text{(six.literal } \$1) \\
\text{expr} & & \text{($2)} \\
\text{stat list} & & \text{($1$ . $2)} \\
\text{parameters} & ::= \\
\text{nonempty parameters} & & \text{($1)} \\
\text{parameter} & , \ \text{nonempty parameters} & \text{($1$ . $3)} \\
\text{parameter} & & \text{($1$)} \\
\text{type} & \text{id-or-prefix} & \text{($2$ $1)} \\
\text{arguments} & ::= \\
\text{nonempty arguments} & & \text{($1)} \\
\text{expr} & , \ \text{nonempty arguments} & \text{($1$ . $3)} \\
\text{expr} & & \text{($1)} \\
\text{elements} & ::= \\
\text{nonempty elements} & & \text{($1)} \\
\text{expr} & & \text{(six.null)} \\
\text{nonempty elements} & ::= \\
\text{expr} & & \text{(six.list } \$1 \ \text{(six.null))} \\
\text{expr} & , \ \text{nonempty elements} & \text{(six.list } \$1 \ $3)
\end{align*}
\]
18.12.2 SIX semantics

The semantics of SIX depends on the definition of the six.xxx identifiers (as functions and macros). Many of these identifiers are predefined macros which give SIX a semantics that is close to C’s. The user may override these definitions to change the semantics either globally or locally. For example, six.x^y is a predefined macro that expands (six.x^y x y) into (bitwise-xor x y). If the user prefers the ‘ˆ’ operator to express exponentiation in a specific function, then in that function six.x^y can be redefined as a macro that expands (six.x^y x y) into (expt x y). Note that the associativity and precedence of operators cannot be changed as that is a syntactic issue.

Note that the following identifiers are not predefined, and consequently they do not have a predefined semantics: six.label, six.goto, six.switch, six.case, six.break, six.continue, six.return, six.clause, six.x:-y, and six.!. The following is an example showing some of the predefined semantics of SIX:

```
> (list (+ 1 2) \3+4; (+ 5 6))
(3 7 11)
> \[ 1+2, \(+ 3 4), 5+6 ];
(3 7 11)
> (map (lambda (x) \(x*x-1)/log(x+1);) ’(1 2 3 4))
(0.2730717679880512 5.7707801635558535 9.320024018394177) 0.320024018394177)
> \obj n = expt(10,5);
> n
100000
> \obj t[3][10] = 88;
> \t[0][9] = t[2][1] = 11;
11
> t
#((88 88 88 88 88 88 88 88 88 11)
#((88 88 88 88 88 88 88 88 88 88))
> \obj radix = new parameter (10);
> \radix(2);
> \radix();
2
> \for (int i=0; i<5; i++) pp(1<<i*8);
1
256
```
\(65536\)
\(16777216\)
\(4294967296\)

\(\text{obj}\) \make-adder (obj x) \{ obj (obj y) \{ x+y; \}; \}

\(\text{map}\) (new adder (100), [1,2,3,4]);

\(101\ 102\ 103\ 104\)

\(\text{map}\) (make-adder 100) (list 1 2 3 4))

\(101\ 102\ 103\ 104\)
Chapter 19: C-interface

19 C-interface

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.

19.1 The mapping of types between C and Scheme

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

- **scheme-object** denotes the universal type of Scheme objects (type ___SCMOBJ defined in ‘gambit.h’)
- **bool** denotes the C++ ‘bool’ type or the C ‘int’ type (type ___BOOL defined in ‘gambit.h’)
- **int8** 8 bit signed integer (type ___S8 defined in ‘gambit.h’)
- **unsigned-int8** 8 bit unsigned integer (type ___U8 defined in ‘gambit.h’)
- **int16** 16 bit signed integer (type ___S16 defined in ‘gambit.h’)
- **unsigned-int16** 16 bit unsigned integer (type ___U16 defined in ‘gambit.h’)
- **int32** 32 bit signed integer (type ___S32 defined in ‘gambit.h’)
- **unsigned-int32** 32 bit unsigned integer (type ___U32 defined in ‘gambit.h’)
- **int64** 64 bit signed integer (type ___S64 defined in ‘gambit.h’)
- **unsigned-int64** 64 bit unsigned integer (type ___U64 defined in ‘gambit.h’)
- **float32** 32 bit floating point number (type ___F32 defined in ‘gambit.h’)
- **float64** 64 bit floating point number (type ___F64 defined in ‘gambit.h’)
- **ISO-8859-1** denotes ISO-8859-1 encoded characters (8 bit unsigned integer, type ___ISO_8859_1 defined in ‘gambit.h’)
- **UCS-2** denotes UCS-2 encoded characters (16 bit unsigned integer, type ___UCS_2 defined in ‘gambit.h’)

...
UCS-4 denotes UCS-4 encoded characters (32 bit unsigned integer, type \_\_\_UCS_4 defined in ‘gambit.h’)

char-string denotes the C ‘char*’ type when used as a null terminated string

nonnull-char-string denotes the nonnull C ‘char*’ type when used as a null terminated string

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

ISO-8859-1-string denotes ISO-8859-1 encoded strings (null terminated string of 8 bit unsigned integers, i.e. \_\_\_ISO_8859_1*)

nonnull-ISO-8859-1-string denotes nonnull ISO-8859-1 encoded strings (null terminated string of 8 bit unsigned integers, i.e. \_\_\_ISO_8859_1*)

nonnull-ISO-8859-1-stringlist denotes an array of nonnull ISO-8859-1 encoded strings terminated with a null pointer

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

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

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

UTF-16-string denotes UTF-16 encoded strings (null terminated string of char, i.e. char*)

nonnull-UTF-16-string denotes nonnull UTF-16 encoded strings (null terminated string of char, i.e. char*)

nonnull-UTF-16-string-list denotes an array of nonnull UTF-16 encoded strings terminated with a null pointer

UCS-2-string denotes UCS-2 encoded strings (null terminated string of 16 bit unsigned integers, i.e. \_\_\_UCS_2*)

nonnull-UCS-2-string denotes nonnull UCS-2 encoded strings (null terminated string of 16 bit unsigned integers, i.e. \_\_\_UCS_2*)

nonnull-UCS-2-string-list denotes an array of nonnull UCS-2 encoded strings terminated with a null pointer
UCS-4-string denotes UCS-4 encoded strings (null terminated string of 32 bit unsigned integers, i.e. ___UCS_4*)

nonnull-UCS-4-string denotes nonnull UCS-4 encoded strings (null terminated string of 32 bit unsigned integers, i.e. ___UCS_4*)

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

wchar_t-string denotes wchar_t encoded strings (null terminated string of wchar_t, i.e. wchar_t*)

nonnull-wchar_t-string denotes nonnull wchar_t encoded strings (null terminated string of wchar_t, i.e. wchar_t*)

nonnull-wchar_t-string-list denotes an array of nonnull wchar_t 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:

<table>
<thead>
<tr>
<th>Scheme notation</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>void</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>char</td>
<td>char (may be signed or unsigned depending on the C compiler)</td>
</tr>
<tr>
<td>signed-char</td>
<td>signed char</td>
</tr>
<tr>
<td>unsigned-char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>ISO-8859-1</td>
<td>ISO-8859-1</td>
</tr>
<tr>
<td>UCS-2</td>
<td>UCS-2</td>
</tr>
<tr>
<td>UCS-4</td>
<td>UCS-4</td>
</tr>
<tr>
<td>wchar_t</td>
<td>wchar_t</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>unsigned-short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>unsigned-int</td>
<td>unsigned int</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td>unsigned-long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>long-long</td>
<td>long long</td>
</tr>
</tbody>
</table>
unsigned-long-long
unsigned long long
float float
double double
int8 int8
unsigned-int8 unsigned-int8
int16 int16
unsigned-int16 unsigned-int16
int32 int32
unsigned-int32 unsigned-int32
int64 int64
unsigned-int64 unsigned-int64
float32 float32
float64 float64

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

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

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

(pointer type [tags [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 tags and release-function)

(nonnull-pointer type [tags [release-function]])
same as (pointer type [tags [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
ISO-8859-1-string
    ISO-8859-1-string
nonnull-ISO-8859-1-string
    nonnull-ISO-8859-1-string
nonnull-ISO-8859-1-string-list
    nonnull-ISO-8859-1-string-list
UTF-8-string    UTF-8-string
nonnull-UTF-8-string
    nonnull-UTF-8-string
nonnull-UTF-8-string-list
    nonnull-UTF-8-string-list
UTF-16-string    UTF-16-string
nonnull-UTF-16-string
    nonnull-UTF-16-string
nonnull-UTF-16-string-list
    nonnull-UTF-16-string-list
UCS-2-string    UCS-2-string
nonnull-UCS-2-string
    nonnull-UCS-2-string
nonnull-UCS-2-string-list
    nonnull-UCS-2-string-list
UCS-4-string    UCS-4-string
nonnull-UCS-4-string
    nonnull-UCS-4-string
nonnull-UCS-4-string-list
    nonnull-UCS-4-string-list
wchar_t-string
    wchar_t-string
nonnull-wchar_t-string
    nonnull-wchar_t-string
nonnull-wchar_t-string-list
    nonnull-wchar_t-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).

The optional tags field of foreign type specifications is used for type checking on the Scheme side. The tags field must be #f, a symbol or a non-empty list of symbols. When it is not specified the tags field defaults to a symbol whose name, as returned by symbol->string, is the C type declaration for that type. For example the symbol ‘char**’ is the default for the type ‘(pointer (pointer char))’. A tags field that is a single symbol is equivalent to a list containing only that symbol. The first symbol in the list of tags is the primary tag. For example the primary tag of the type ‘(pointer char)’ is ‘char*’ and the primary tag of the type ‘(pointer char (foo bar))’ is ‘foo’.

Type compatibility between two foreign types depends on their tags. An instance of a foreign type T can be used where a foreign type E is expected if and only if
- T’s tags field is #f, or
- E’s tags field is #f, or
- T’s primary tag is a member of E’s tags.

For the safest code a tags field of #f should be used sparingly, as it completely bypasses type checking. The external representation of Scheme foreign objects (used by the write procedure) contains the primary tag (if the tags field is not #f), and the hexadecimal address denoted by the internal pointer, for example ‘#<char** #2 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 (foreign-release! obj) which invokes the release-function if the foreign object is not yet released, and does nothing otherwise. The call (foreign-released? obj) returns a boolean indicating whether the foreign object obj has been released yet or not. The call (foreign-address obj) returns the address denoted by the internal pointer of foreign object obj or 0 if it has been released. The call (foreign-tags obj) tests that obj is a foreign object. Finally the call (foreign-tags obj) returns the list of tags of foreign object obj, or #f.

The following table gives the C types to which each Scheme type can be converted:

<table>
<thead>
<tr>
<th>Scheme type</th>
<th>Allowed target C types</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean #f</td>
<td>scheme-object; bool; pointer; function; char-string;</td>
</tr>
<tr>
<td></td>
<td>ISO-8859-1-string; UTF-8-string; UTF-16-string; UCS-2-string;</td>
</tr>
<tr>
<td></td>
<td>UCS-4-string; wchar_t-string</td>
</tr>
<tr>
<td>boolean #t</td>
<td>scheme-object; bool</td>
</tr>
<tr>
<td>character</td>
<td>scheme-object; bool; [[un]signed] char; ISO-8859-1; UCS-2;</td>
</tr>
<tr>
<td></td>
<td>UCS-4; wchar_t</td>
</tr>
<tr>
<td>exact integer</td>
<td>scheme-object; bool; [unsigned-] int8/int16/int32/int64;</td>
</tr>
<tr>
<td></td>
<td>[unsigned] short/int/long</td>
</tr>
<tr>
<td>inexact real</td>
<td>scheme-object; bool; float; double; float32; float64</td>
</tr>
<tr>
<td>string</td>
<td>scheme-object; bool; char-string; nonnull-char-string;</td>
</tr>
<tr>
<td></td>
<td>ISO-8859-1-string; nonnull-ISO-8859-1-string; UTF-8-string;</td>
</tr>
<tr>
<td></td>
<td>nonnull-UTF-8-string; UTF-16-string; nonnull-UTF-16-string;</td>
</tr>
<tr>
<td></td>
<td>UCS-2-string; nonnull-UCS-2-string; UCS-4-string; nonnull-UCS-4-string; wchar_t-string; nonnull-wchar_t-string</td>
</tr>
<tr>
<td>foreign object</td>
<td>scheme-object; bool; struct/union/type-pointer/nonnull-pointer with the appropriate tags</td>
</tr>
<tr>
<td>vector</td>
<td>scheme-object; bool</td>
</tr>
<tr>
<td>symbol</td>
<td>scheme-object; bool</td>
</tr>
<tr>
<td>procedure</td>
<td>scheme-object; bool; function; nonnull-function</td>
</tr>
</tbody>
</table>
other objects  

scheme-object; bool

The following table gives the Scheme types to which each C type will be converted:

<table>
<thead>
<tr>
<th>C type</th>
<th>Resulting Scheme type</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme-object</td>
<td>the Scheme object encoded</td>
</tr>
<tr>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td>[[un]signed] char; ISO-8859-1; UCS-2; UCS-4; wchar_t</td>
<td>character</td>
</tr>
<tr>
<td>[unsigned-] int8/int16/int32/int64; [unsigned] short/int/long</td>
<td>exact integer</td>
</tr>
<tr>
<td>float; double; float32; float64</td>
<td>inexact real</td>
</tr>
<tr>
<td>char-string; ISO-8859-1-string; UTF-8-string; UTF-16-string; UCS-2-string; UCS-4-string; wchar_t-string</td>
<td>string or #f if it is equal to ‘NULL’</td>
</tr>
<tr>
<td>nonnull-char-string; nonnull-ISO-8859-1-string; nonnull-UTF-8-string; nonnull-UTF-16-string; nonnull-UCS-2-string; nonnull-UCS-4-string; nonnull-wchar_t-string</td>
<td>string</td>
</tr>
<tr>
<td>struct/union/type/pointer/nonnull-pointer</td>
<td>foreign object with the appropriate tags or #f in the case of a pointer equal to ‘NULL’</td>
</tr>
<tr>
<td>function</td>
<td>procedure or #f if it is equal to ‘NULL’</td>
</tr>
<tr>
<td>nonnull-function</td>
<td>procedure</td>
</tr>
<tr>
<td>void</td>
<td>void object</td>
</tr>
</tbody>
</table>

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, ISO-8859-1-string, UTF-8-string, UTF-16-string, UCS-2-string, UCS-4-string, wchar_t-string, pointer or function type is expected. In this case, #f is converted to the ‘NULL’ pointer. C bools 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 cdefine 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 tags (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 compatible 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 cdefine 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.

19.2 The cdeclare special form

(c-declare c-declaration) special form
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 #<<c-declare-end
#include <stdio.h>
extern char *getlogin ();
#ifdef sparc
char *host = "sparc";
#else
char *host = "unknown";
#endif
FILE *tfile;
c-declare-end)

19.3 The c-initialize special form

(c-initialize c-code) special form

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 ();")

19.4 The c-lambda special form

(c-lambda (type1...) result-type c-name-or-code) special form

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 when the result is of type `struct`, `union`, `type`, `pointer`, `nonnull-pointer`, `function` or `nonnull-function` in which case a pointer must be assigned to the variable `___result_voidstar` which is of type `void*`. For results of type `pointer`, `nonnull-pointer`, `function` and `nonnull-function`, the value assigned to the variable `___result_voidstar` must be the pointer or function cast to `void*`. For results of type `struct`, `union`, and `type`, the value assigned to the variable `___result_voidstar` must be a pointer to a memory allocated block containing a copy of the result. Note that this block will be reclaimed by the `release-function` associated with the type. If no result needs to be returned, the `result-type` should be `void` and no assignment to the variable `___result` or `___result_voidstar` should take place. Note that the C code should not contain return statements as this is meaningless. Control must always fall off the end of the C code. The C code is copied to the C file on a new line so it can start with preprocessor directives. Moreover the C code is always placed at the head of a compound statement whose lifetime encloses the C to Scheme conversion of the result. Consequently, temporary storage (strings in particular) declared at the head of the C code can be returned by assigning them to `___result` or `___result_voidstar`. In the `c-name-or-code`, the macro `___AT_END` may be defined as the piece of C code to execute before control is returned to Scheme but after the result is converted to its Scheme representation. This is mainly useful to deallocate temporary storage contained in the result.

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

For example:

```scheme
(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_voidstar = stdin;"))

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

(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);
 c-lambda-end
 ))

(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;
 c-lambda-end
 ))

19.5 The c-define special form

(c-define (variable define-formals) (type1...) result-type c-name special form
 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’.

19.6 The c-define-type special form

(c-define-type name type [c-to-scheme scheme-to-c [cleanup]]) special form

This form associates the type identifier name to the C type type. The name must not clash with predefined types (e.g. char-string, ISO-8859-1, 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 19.7 [continuations], page 218). 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:
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:

```c
if ((___err = conversion_operation) == ___FIX(___NO_ERR)) {
```

The last ___BEGIN macro must expand to an unterminated compound statement:

```c
{ ___err = conversion_operation;
```

If type check errors are impossible then a ___BEGIN macro can simply expand to an unterminated compound statement performing the conversion:

```c
{ 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 #<<c-declare-end
typedef char EBCDIC; /* EBCDIC encoded characters */

void put_char (EBCDIC c) { ... } /* EBCDIC I/O functions */
EBCDIC get_char (void) { ... }

```
Chapter 19: C-interface

char EBCDIC_to_ISO_8859_1[256] = { ... }; /* conversion tables */
char ISO_8859_1_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 ___FIX(___UNKNOWN_ERR);
  *dst = ISO_8859_1_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_ISO_8859_1[src]);
#define ___END_CFUN_EBCDIC_to_SCMOBJ(src,dst) }

#define ___BEGIN_SFUN_EBCDIC_to_SCMOBJ(src,dst,i) \ 
  { dst = ___CHR(EBCDIC_to_ISO_8859_1[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-declare-end

(c-declare #<<c-declare-end

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)

c-declare

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-declare-type is #f).

(c-declare #<<c-declare-end

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)
Chapter 19: C-interface

```c
{  
___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(car,dst->x,arg_num)
    ___END_CFUN_SCMOBJ_TO_INT(cdr,dst->y,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 ___BEGIN_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 ___BEGIN_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 ___BEGIN_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-declare-end
)

(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 #<<c-declare-end
#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 re-claim 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) \

Chapter 19: C-interface

19.7 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:

```
int p (char *); /* forward declarations */
int q (void);

int a (char *x) { return 2 * p (x+1); }
int b (short y) { return y + q (); }

c-declare-end

(c-declare #<<c-declare-end

int p (char *); /* forward declarations */
int q (void);

int a (char *x) { return 2 * p (x+1); }
int b (short y) { return y + q (); }

c-declare-end

(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.
20 System limitations

- On some systems floating point overflows will cause the program to terminate with a floating point exception.
- On some systems floating point operations involving ‘+nan.0’ ‘+inf.0’, ‘-inf.0’, or ‘-0.’ do not return the value required by the IEEE 754 floating point standard.
- 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!).
- The maximum number of arguments that can be passed to a procedure by the apply procedure is 8192.
21 Copyright and license

The Gambit-C system release v4.6.1 is Copyright © 1994-2009 by Marc Feeley, all rights reserved. The Gambit-C system release v4.6.1 is licensed under two licenses: the Apache License, Version 2.0, and the GNU LESSER GENERAL PUBLIC LICENSE, Version 2.1. You have the option to choose which of these two licenses to abide by. The licenses are copied below.
Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the
Chapter 21: Copyright and license

Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents
of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason
of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.
GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts
as the successor of the GNU Library Public License, version 2, hence
the version number 2.1.]

Preamble

The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it. You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use,
not price. Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.

To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights. These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you. You must make sure that they, too, receive or can get the source
code. If you link other code with the library, you must provide
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that
there is no warranty for the free library. Also, if the library is
modified by someone else and passed on, the recipients should know
that what they have is not the original version, so that the original
author’s reputation will not be affected by problems that might be
introduced by others.
Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user’s freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users’ freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.
GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification").

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library’s complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) The modified work must itself be a software library.

   b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

      (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.
Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.
6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally
accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients’ exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.
11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.
14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS
How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library’s name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library ‘Frob’ (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That’s all there is to it!
General index

#
# ...........................................  33
## ........................................  33
+
+2 ...........................................  19
,
, (b expr) ..................................  30
, (be expr) ..................................  30
, (bed expr) ................................  30
, (c expr) ..................................  28
, (e expr) ..................................  30
, (ed expr) ..................................  31
, (h subject) ................................  28
, (st expr) ..................................  31
, (v expr) ..................................  31
,+ .............................................  29
,++ ...........................................  29
,-- ...........................................  29
,? ............................................  28
,b .............................................  29
,be ..........................................  30
,bed .........................................  30
,c .............................................  28
,d .............................................  28
,e .............................................  30
,ed ..........................................  30
,h .............................................  30
,i .............................................  30
,i .............................................  29
,N ............................................  29
,N+ ..........................................  29
,N- ..........................................  29
,q .............................................  28
,qt ..........................................  28
,s .............................................  28
,st ..........................................  31
,t .............................................  28
,y .............................................  29
-
-: The `@[INTE][:PORT]’ option ..................  12
-:d ..........................................  23
-:d@[HOST][:PORT] .........................  24
-:da ........................................  24
-:dc ........................................  24
-:dD ........................................  24
-:dI ........................................  24
-:dLEVEL ...................................  24
-:dp ........................................  23
-:dq ........................................  24
-:dQ ........................................  24
-:dr ........................................  24
-:ds ........................................  24
-:f ...........................................  24
-:fR ..........................................  24
-:h ...........................................  23
-:l ...........................................  23
-:m ...........................................  23
-:s ..........................................  23
-:S ..........................................  49
-:t ...........................................  24
-:c ...........................................  12
-:call_shared ................................  19
-:cc-options ................................  11
-:D_DYNAMIC ................................  15
-:D_LIBRARY ................................  18
-:D_PRIMAL ................................  18
-:D_SHARED ..................................  18
-:D_SINGLE_HOST .........................  18
-:debug ....................................  11, 51
-:debug-environments ..................  12, 52
-:debug-location .........................  11, 52
-:debug-source .........................  12, 52
-:dynamic ..................................  12
-:e ...........................................  12
-:exe ........................................  9, 12
-:extension ...............................  11
-:flat .....................................  12
-:fPIC ......................................  19
-:fPIC ......................................  19
-:gvm ......................................  11
-:G ..........................................  19
-:i ..........................................  11
-:I/usr/local/Gambit-C/include ........  18
-:keep-c ...................................  12
-:Kpic ......................................  19
-:Kpic ......................................  19
-:l base ...................................  12
-:L/usr/local/Gambit-C/lib ..........  18
-:ld-options ............................  11
-:ld-options-include .................  11
-:link ......................................  9, 12
-:o output ................................  12
-:obj .......................................  12
-:O ..........................................  18
-:pic .......................................  19
-postlude                       11
-prelude                        11
-rdynamic                      19
-report                        11
-shared                        19
-track-scheme                  12
-verbose                       11
-warnings                      11

.
.c                               9
.scm                             9
.six                             9

<
<                                   65
<=                              65

=
=                               65

>
>                             65
>=                             65

^C                                24, 27
^D                               27

| cleanup                      211
| setup                         211

|six.x,y|                             60
|six.x\|=y|                           60
|six.x\|\|y|                           61
|six.x\|\|y|                           60

~
~                              134
~                               134
~username                         134

A
abandoned-mutex-exception?         119
abort                                  116
absolute path                         134, 135
address-info-family                  149
address-info-protocol                149
address-info-socket-info             150
address-info-socket-type             149
address-info?                        149
address-infos                         148
all-bits-set?                        68
any-bits-set?                        68
append-f32vectors                   80
append-f64vectors                   80
append-s16vectors                   78
append-s32vectors                   79
append-s64vectors                   79
append-s8vectors                    77
append-strings                      46
append-u6vectors                    78
append-u32vectors                   79
append-u64vectors                   80
append-u8vectors                    77
append-vectors                      45
arithmetic-shift                   66

B
bit-count                           67
bit-set?                             68
bitwise-and                          66
bitwise-ior                          66
bitwise-merge                        66
bitwise-not                          67
bitwise-xor                          67
block                                50
box                                  47
box?                                 47
boxes                                47
break                                36

C
c-declare                         207
c-define                        210
c-define-type                    211
c-initialize                    208
c-lambda                         208
call-with-current-continuation..  42
call-with-input-file              166
call-with-input-process          168
call-with-input-string           178
call-with-input-u8vector         179
call-with-input-vector           175
call-with-output-file            166
call-with-output-process         168
call-with-output-string          178
call-with-output-u8vector        179
call-with-output-vector ................. 175
 call/cc .................................... 42
 central installation directory .......... 134
 cfun-conversion-exception-arguments ... 122
 cfun-conversion-exception-code ........ 122
 cfun-conversion-exception-message ..... 122
 cfun-conversion-exception-procedure ... 122
 cfun-conversion-exception? ............. 122
 char->integer ................................ 63
 char-ci<=? .................................. 63
 char-ci<=? .................................. 63
 char-ci=>? .................................. 63
 char-ci=>? .................................. 63
 char<=? ....................................... 63
 char<=? ....................................... 63
 char>=? ....................................... 63
 char>=? ....................................... 63
 clear-bit-field .............................. 69
 close-input-port .......................... 157
 close-output-port ......................... 157
 close-port ................................... 157
 command-line ................................ 6, 139
 compile-file ................................ 19
 compile-file-to-c ........................... 19
 compiler ....................................... 9
 compiler options ............................ 9
 cond-expand .................................. 59
 condition-variable-broadcast! ........... 108
 condition-variable-name ................... 107
 condition-variable-signal! ............... 107
 condition-variable-specific ............... 107
 condition-variable-specific-set! ........ 107
 condition-variable? .......................... 107
 configure-command-string ................. 58
 console-port .................................. 58
 constant-fold ................................ 51
 continuation-capture ........................ 54
 continuation-graft ........................... 54
 continuation-return .......................... 54
 continuation? ................................ 54
 continuations .................................. 218
 copy-bit-field ............................... 69
 copy-file ..................................... 138
 cpu-time ..................................... 140
 create-directory ............................ 137
 create-fifo ................................... 137
 create-link ................................... 137
 create-symbolic-link ....................... 137
 current exception-handler ................ 114
 current working directory ................ 134
 current-directory ......................... 134
 current-error-port ......................... 179
 current-exception-handler ............... 114
 current-input-port ....................... 179
 current-output-port ...................... 179
 current-readable ......................... 179
 current-thread .............................. 96
 current-time ................................. 140
 current-user-interrupt-handler .......... 58

datum-parsing-exception-kind ............ 125
datum-parsing-exception-parameters ...... 125
datum-parsing-exception-readenv ......... 125
datum-parsing-exception? ................. 125
deadlock-exception ......................... 119
default-random-source ..................... 72
deffer-user-interrupts ..................... 58
define ......................................... 42, 220
define-cond-expand-feature ............... 59
define-macro .................................. 49
define-record-type ........................... 59
define-structure ................................ 92
define-syntax .................................. 49
define-type .................................... 59
define-type-of-thread ....................... 57
delete-directory .............................. 138
delete-file ...................................... 138
deserialization .............................. 81, 183
directory-files ............................... 138
display-continuation-backtrace .......... 55
display-continuation-dynamic-environment ... 55
display-continuation-environment ....... 55
display-dynamic-environment? ............. 37
display-environment-set! ................. 37
display-exception ............................ 55
display-exception-in-context .............. 55
display-procedure-environment .......... 55
divide-by-zero-exception-arguments ...... 129
divide-by-zero-exception-procedure ...... 129
divide-by-zero-exception? ............... 129
Chapter 21: General index 239

E
Emacs ................................................. 40
eq?-hash ............................................. 83
equal?-hash ......................................... 84
eqv?-hash ............................................ 84
e-error-code->string .............................. 58
error .................................................. 133
error-exception-message ....................... 133
error-exception-parameters ................... 133
error-exception? ................................. 133
eval .................................................... 49
exit ................................................... 139
expression-parsing-exception-kind .... 126
expression-parsing-exception-parameters ... 126
expression-parsing-exception-source .... 126
expression-parsing-exception? ............ 126
extended-bindings ............................... 51
extract-bit-field ............................... 60

F
f32vector ............................................ 80
f32vector->list .................................. 80
f32vector-append ............................... 80
f32vector-copy .................................. 80
f32vector-fill! .................................... 80
f32vector-length ............................... 80
f32vector-ref .......... .......................... 80
f32vector-set! ................................. 80
f32vector-shrink! .............................. 80
f32vector? .......................... .......................... 80
f64vector ............................................ 80
f64vector->list .................................. 80
f64vector-append ......................... 80
f64vector-copy .................................. 80
f64vector-fill! .................................... 80
f64vector-length ............................... 80
f64vector-ref ................................... 80
f64vector-set! .................................... 80
f64vector-shrink! .............................. 81
f64vector? ........................................... 80
FFI .................................................. 199
file names ......................................... 134
file-attributes ..................................... 144
file-creation-time ......................... 144
file-device ........................................ 144
file-exists? ..................................... 141
file-group ........................................ 144
file-info ......................................... 142
file-info-attributes ............................ 144
file-info-creation-time ...................... 144
file-info-device ................................ 143
file-info-group .................................. 143
file-info-inode .................................. 143
file-info-last-access-time .................. 144
file-info-last-change-time .................. 144
file-info-last-modification-time .... 144
file-info-mode .................................... 144
file-info-number-of-links ................... 143
file-info-owner .................................. 143
file-info-size .................................... 143
file-info-type ................................... 142
file-info? ........................................... 142
file-inode ....................................... 144
file-last-access-time ....................... 144
file-last-change-time ....................... 144
file-last-modification-time .............. 144
file-mode ......................................... 144
file-number-of-links ....................... 144
file-owner ..................................... 144
file-size ......................................... 144
file-type ........................................... 144
file.c .............................................. 9
file.scm ........................................... 9
file.six ............................................. 9
finite? ............................................. 59
first-bit-set ..................................... 68
fixnum ............................................. 53
fixnum->flonum ................................. 71
fixnum-overflow-exception-arguments .... 70
fixnum-overflow-exception-procedure .... 70
fixnum-overflow-exception? ............... 70
fixnum? ............................................ 69
fl* .................................................... 71
fl+ .................................................... 71
fl- .................................................... 71
fl/ ..................................................... 71
fl<= ................................................... 71
fl=< ................................................... 71
fl>= ................................................... 71
flabs ............................................... 71
flacos ............................................. 71
flasin .............................................. 71
flatan ............................................. 71
flceiling ......................................... 72
flcos .............................................. 72
fldenominator ................................... 72
fleven? ............................................. 72
flexp ............................................... 72
flexpt .............................................. 72
flfinite? ......................................... 72
flfloor ............................................. 72
flmax .............................................. 72
flmin .............................................. 72
flnan? .............................................. 72
finegative? ........................................ 72
finumerator  ........................................ 72
floating point overflow  ....................... 220
floodd? ........................................ 72
fionum ........................................ 53
fionum? ........................................ 71
fipositive? ........................................ 72
firound ........................................ 72
fisin ........................................ 72
fisqrt ........................................ 72
fitan ........................................ 72
fitruncate ........................................ 72
fizerot ........................................ 72
force-output ..................................... 157
foreign function interface ...................... 199
foreign-address .................................. 58
foreign-release! .................................. 58
foreign-released? ................................ 58
foreign-tags ..................................... 58
foreign? ........................................ 58
future ........................................ 58
fx* ........................................ 69
fx+ ........................................ 69
fx- ........................................ 69
fx< ........................................ 69
fx<= ........................................ 69
fx= ........................................ 69
fx> ........................................ 69
fx>= ........................................ 69
fxabs ........................................ 69
fxand ........................................ 69
fxarithmetic-shift ................................ 70
fxarithmetic-shift-left ......................... 70
fxarithmetic-shift-right ......................... 70
fxbit-count ..................................... 70
fxbit-set? ...................................... 70
fxeven? ....................................... 70
fxfirst-bit-set .................................. 70
fxif ........................................ 70
fxior ........................................ 70
fxlength ....................................... 70
fxmax ....................................... 70
fxmin ....................................... 70
fxmodulo ...................................... 70
fxnegative? .................................... 70
fxnot ........................................ 70
fxodd? ....................................... 70
fxpositive? .................................... 70
fxquotient ..................................... 70
fxremainder ................................... 70
fxwrap* ....................................... 70
fxwrap+ ....................................... 70
fxwrap- ....................................... 70
fxwrapabs ..................................... 70
fxwraparithmetic-shift ......................... 70
fxwraparithmetic-shift-left ..................... 70
fxwraplogical-shift-right ...................... 70
fxwrapquotient ................................ 70
fxxor ....................................... 70
fxzero? ....................................... 70
G
GAMBCOPT, environment variable .................. 25
Gambit .......................................... 1
Gambit-C ........................................ 1
gambit-scheme ................................... 50
gambit.el ........................................ 40
gc-report-set! ................................... 38
GC ............................................ 38
generate-proper-tail-calls ....................... 36
generic ......................................... 53
gensym .......................................... 47
get-output-string ................................ 178
get-output-u8vector .............................. 179
get-output-vector ................................ 177
getenv .......................................... 139
group-info ....................................... 145
group-info-gid .................................. 145
group-info-members .............................. 145
group-info-name ................................ 145
group-info? ..................................... 145
gsc ............................................ 1, 9, 19, 21, 23
gsc-script ....................................... 6
gsi ............................................ 1, 9, 23
gsi-script ....................................... 6
H
hashing .......................................... 82
heap-overflow-exception? ....................... 116
help ........................................... 32
help-browser .................................... 32
home directory .................................. 134
homogeneous vectors ................................ 77, 191
host-info ....................................... 147
host-info-addresses .............................. 148
host-info-aliases ............................... 148
host-info-name ................................ 147
host-info? ...................................... 147
host-name ...................................... 147
I
ieee-scheme ..................................... 50
improper-length-list-exception-arg-num .... 130
improper-length-list-exception-arguments ... 130
improper-length-list-exception-procedure ... 130
improper-length-list-exception? .............. 130
inactive-thread-exception-arguments .......... 57
inactive-thread-exception-procedure.......................... 57
inactive-thread-exception?.......................... 57
include........................................ 49
infinite?.................................. 59
initialized-thread-exception-arguments............... 57
initialized-thread-exception-procedure............... 57
initialized-thread-exception?......................... 57
inline.................................... 50
inline-primitives................................ 50
inlining-limit.................................. 50
input-port-byte-position............................. 167
input-port-bytes-buffered............................ 57
input-port-char-position.............................. 59
input-port-characters-buffered...................... 57
input-port-column................................ 159
input-port-line.................................. 159
input-port-readable................................. 161
input-port-readable-set!............................. 161
input-port-timeout-set!.............................. 158
input-port?.................................... 155
installation directories............................... 134
integer->char................................... 63
integer-length.................................... 67
integer-nth-root.................................. 65
integer-sqrt..................................... 65
interpreter...................................... 3, 9
interrupts-enabled................................. 51
invalid-hash-number-exception-arguments............. 58
invalid-hash-number-exception-procedure............. 58
invalid-hash-number-exception?...................... 58
L
lambda......................................... 42
lambda-lift.................................... 51
last_c........................................ 12
limitations....................................... 220
link-flat....................................... 21
link-incremental................................. 21
list->f32vector................................ 80
list->f64vector................................ 80
list->s16vector................................ 78
list->s32vector................................ 78
list->s64vector................................ 79
list->s8vector................................ 77
list->table..................................... 89
list->u16vector................................ 78
list->u32vector................................ 79
list->u64vector................................ 80
list->u8vector.................................. 77
load............................................ 19
M
mailbox-receive-timeout-exception-arguments........... 103
mailbox-receive-timeout-exception-procedure........... 103
mailbox-receive-timeout-exception?.................... 103
main............................................. 59
make-condition-variable............................. 107
make-f32vector.................................. 80
make-f64vector.................................. 80
make-mutex...................................... 103
make-parameter................................... 111
make-random-source................................. 73
make-root-thread................................. 96
make-s16vector.................................. 78
make-s32vector.................................. 78
make-s64vector.................................. 79
make-s8vector................................... 77
make-table...................................... 86
make-thread..................................... 96
make-thread-group................................. 56
make-u16vector.................................. 78
make-u32vector.................................. 79
make-u64vector.................................. 79
make-u8vector................................... 77
make-uninterned-keyword............................ 48
make-uninterned-symbol............................. 48
make-will....................................... 84
mostly-fixnum................................... 53
mostly-fixnum-flonum............................... 53
mostly-flonum................................... 53
mostly-flonum-fixnum............................... 53
mostly-generic................................... 53
multiple-c-return-exception?......................... 124
mutex-lock!..................................... 105
mutex-name...................................... 104
J
join-timeout-exception-arguments......................... 120
join-timeout-exception-procedure....................... 120
join-timeout-exception?............................. 120
K
keyword->string................................... 47
keyword-expected-exception-arguments............... 132
keyword-expected-exception-procedure............... 132
keyword-expected-exception?........................ 132
keyword-hash.................................... 83
keyword?...................................... 47
keywords...................................... 47
Chapter 21: General index

mutex-specific .......................... 104
mutex-specific-set! ........................ 104
mutex-state .............................. 104
mutex-unlock! ................................ 106
mutex? .................................... 103

N
namespace .................................. 59
nan? ........................................ 59
network-info .............................. 152
network-info-aliases ....................... 153
network-info-name ........................ 153
network-info-number ...................... 153
network-info? ................................ 152
newline ...................................... 156
no-such-file-or-directory-
exception-arguments ..................... 117
no-such-file-or-directory-
exception-procedure ...................... 117
no-such-file-or-directory-
exception? .................................. 117
noncontinuous-exception-reason ..... 116
noncontinuous-exception? ................. 116
nonempty-input-port-character-
buffer-exception-arguments ............ 57
nonempty-input-port-character-
buffer-exception-procedure .......... 58
nonempty-input-port-character-
buffer-exception? ......................... 57
nonprocedure-operator-exception-
arguments .................................. 131
nonprocedure-operator-exception-
code ......................................... 131
nonprocedure-operator-exception-
operator .................................... 131
nonprocedure-operator-exception-
rt e ........................................... 131
nonprocedure-operator-exception? ...... 131
normalized path ............................ 135
number-of-arguments-limit-
exception-arguments ..................... 131
number-of-arguments-limit-
exception-procedure ...................... 131
number-of-arguments-limit-
exception? .................................. 131

O
object-file .................................. 19
object->serial-number .................... 82
object->string ............................ 178
object->u8vector .......................... 81
open-directory ............................ 174
open-dummy ................................ 57
open-event-queue .......................... 59
open-file ................................. 166
open-input-file ............................ 166
open-input-process ........................ 168
open-input-string ........................ 178
open-input-u8vector ....................... 179
open-input-vector ........................ 175
open-output-file .......................... 166
open-output-process ........................ 168
open-output-string ........................ 178
open-output-u8vector ..................... 179
open-output-vector ....................... 175
open-process .............................. 168
open-string ............................... 178
open-string-pipe .......................... 178
open-tcp-client ............................ 171
open-tcp-server ............................ 172
open-u8vector ............................. 179
open-u8vector-pipe ........................ 179
open-vector ............................... 175
open-vector-pipe .......................... 177
optimize-dead-local-variables ... 53
options, compiler .......................... 9
options, runtime ........................... 23
os-exception-arguments ................. 117
os-exception-code ........................ 117
os-exception-message ................. 117
os-exception-procedure ................. 117
os-exception? .............................. 117
output-port-byte-position .............. 167
output-port-char-position .............. 167
output-port-column ....................... 159
output-port-line ......................... 159
output-port-readable ..................... 161
output-port-readable-set! ............... 161
output-port-timeout-set! ................. 158
output-port-width ........................ 159
output-port? ............................... 155
overflow, floating point .................. 220

P
parameterize ............................... 112
path-directory ............................. 136
path-expand .............................. 135
path-extension ............................ 136
path-normalize ............................ 135
path-strip-directory ...................... 136
path-strip-extension ..................... 136
path-strip-trailing-directory-
separator .................................. 136
path-strip-volume ........................ 136
path-volume ............................... 136
peek-char ................................. 160
port-settings-set! ......................... 57
port? ....................................... 155
pp ........................................... 38
pretty-print ............................... 37
primordial-exception-handler ........... 58
print ........................................ 179
println .................................. 179
process-pid ................................ 170
process-status ............................. 170
process-times ............................. 140
proper-tail-calls ......................... 36, 52
proper-tail-calls ......................... 52
protocol-info ............................. 151
protocol-info-aliases ..................... 152
protocol-info-name ....................... 152
protocol-info-number .................... 152
protocol-info? ........................... 151
readtable-sharing-allowed? ............. 188
readtable-sharing-allowed? ............. 188
readtable-write-cdr-read-macros? ...... 185
readtable-write-cdr-read-macros?-set 185
readtable-write-cdr-read-macros?-set 185
readtable-write-extended-read-macros? 185
readtable-write-extended-read-macros?-set 185
readtable? ................................ 181
real-time .................................. 140
receive .................................... 59
relative path ............................. 134, 135
rename-file ................................ 138
repl-display-environment? ............... 37
repl-input-port ........................... 58
repl-output-port ......................... 58
repl-result-history-max-length-set ... 33
repl-result-history-ref ................. 33
replace-bit-field ........................ 69
rpc-remote-error-exception-arguments 57
rpc-remote-error-exception-message 57
rpc-remote-error-exception-procedure 57
rpc-remote-error-exception? .......... 57
run-time-bindings ......................... 51
runtime-options ........................... 23

S
s16vector ................................ 78
s16vector->list ......................... 78
s16vector-append ....................... 78
s16vector-copy ......................... 78
s16vector-fill! ......................... 78
s16vector-length ....................... 78
s16vector-ref ........................... 78
s16vector-set! ........................... 78
s16vector-shrink! ....................... 78
s16vector? ................................ 78
s32vector ............................... 78
s32vector->list ......................... 78
s32vector-append ....................... 79
s32vector-copy ......................... 79
s32vector-fill! ......................... 79
s32vector-length ....................... 78
s32vector-ref ........................... 78
s32vector-set! ........................... 78
s32vector-shrink! ....................... 79
s32vector? ................................ 79
s64vector ............................... 79
s64vector->list ......................... 79
s64vector-append ....................... 79
s64vector-copy ........................... 79
Chapter 21: General index

s64vector-fill!.................................................. 79
s64vector-length............................................. 79
s64vector-ref................................................ 79
s64vector-set!................................................ 79
s64vector-shrink!.......................................... 79
s64vector?..................................................... 79
s8vector......................................................... 77
s8vector->list.................................................. 77
s8vector-append.............................................. 77
s8vector-copy.................................................. 77
s8vector-fill!.................................................. 77
s8vector-length.............................................. 77
s8vector-ref................................................... 77
s8vector-set!................................................... 77
s8vector-shrink!............................................. 77
s8vector?......................................................... 77
safe............................................................ 51
scheduler-exception-reason ................................. 119
scheduler-exception?....................................... 119
Scheme.......................................................... 1
scheme-ieee-1178-1990...................................... 6
scheme-r4rs.................................................... 6
scheme-r5rs.................................................... 6
scheme-srfi-0................................................ 6
seconds->time.................................................. 140
separate....................................................... 50
serial-number->object....................................... 82
serialization................................................... 81, 183
service-info.................................................. 150
service-info-aliases........................................ 151
service-info-name........................................... 151
service-info-port-number................................. 151
service-info-protocol....................................... 151
service-info?.................................................. 150
set!............................................................... 220
set-box!.......................................................... 47
setenv.......................................................... 139
sfun-conversion-exception-arguments........................ 123
sfun-conversion-exception-code............................ 123
sfun-conversion-exception-message.......................... 123
sfun-conversion-exception.................................. 123
shell-command................................................ 138
six-script...................................................... 6
six.!.............................................................. 59
six.*x............................................................. 59
six.&x............................................................ 59
six.+x............................................................. 59
six.+x............................................................. 59
six.-x............................................................. 59
six.-x............................................................. 59
six.*x............................................................. 61
six.arrow....................................................... 59
six.break....................................................... 59
six.call......................................................... 59
six.case......................................................... 59
six.clause....................................................... 59
six.compound.................................................. 59
six.cons........................................................ 59
six.continue................................................... 59
six.define-procedure........................................ 59
six.define-variable.......................................... 59
six.do-while................................................... 59
six.dot.......................................................... 59
six.if............................................................ 59
six.index........................................................ 59
six.label......................................................... 59
six.list.......................................................... 60
six.make-array............................................... 60
six.null........................................................ 60
six.prefix....................................................... 60
six.procedure................................................ 60
six.procedure-body.......................................... 60
six.return..................................................... 60
six.switch..................................................... 60
six.x.............................. ............................... 60
six.x%............................ ............................... 60
six.x%y.......................................................... 60
six.x+y.......................................................... 60
six.x&........................................................... 60
six.x&y.......................................................... 60
six.x*y.......................................................... 60
six.x+y.......................................................... 60
six.x-y.......................................................... 60
six.x/=............................ ............................... 60
six.x/=y.......................................................... 60
six.x++.......................................................... 60
six.x+=........................................................ 60
six.x+-........................................................ 60
six.x-=......................................................... 60
six.x==......................................................... 60
six.x=y.......................................................... 60
six.x<=......................................................... 60
six.x<=y........................................................ 60
six.x<y.......................................................... 60
six.x=......................................................... 60
six.x=y.......................................................... 60
six.x+=......................................................... 60
six.x=......................................................... 60
six.x++=....................................................... 60
six.x>......................................................... 60
six.x>=.................................................... 60
six.x>=y....................................................... 60
six.x>y......................................................... 60
six.x?y:z...................................................... 60
six.x^z......................................................... 60
six.x?y:z...................................................... 60
six.x^z......................................................... 60
Chapter 21: General index

thread-mailbox-next .......................... 102
thread-mailbox-rewind ......................... 102
thread-name ................................. 97
thread-priority-boost ......................... 98
thread-priority-boost-set! ..................... 98
thread-quantum ............................... 98
thread-quantum-set! ............................ 98
thread-receive ................................ 102
thread-resume! ................................ 57
thread-send .................................... 101
thread-sleep! .................................. 99
thread-specific ............................... 97
thread-specific-set! ............................ 97
thread-start! ................................... 98
thread-state .................................... 57
thread-state-abnormally-terminated-result 57
thread-state-abnormally-terminated? ........ 57
thread-state-active-timeout .................... 57
thread-state-active-waiting-for ............... 57
thread-state-active? ............................ 57
thread-state-initialized? ......................... 57
thread-state-normally-terminated-result .... 57
thread-state-normally-terminated? ............. 57
thread-state-uninitialized? ..................... 57
thread-suspend! ................................. 57
thread-terminate! ................................ 99
thread-thread-group ........................... 57
thread-yield! ................................... 99
thread? ......................................... 96
treads ........................................... 93
time .............................................. 141
time->seconds .................................. 140
time? ............................................. 140
timeout->time .................................. 57
top ................................................ 57
touch ............................................. 58
trace ............................................. 33
transcript-off .................................. 42
transcript-on ................................... 42
tty-history ...................................... 58
tty-history-max-length-set! ..................... 58
tty-history-set! ................................ 58
tty-mode-set! .................................... 58
tty-paren-balance-duration-set! ............... 58
tty-text-attributes-set! ......................... 58
tty-type-set! .................................... 58
tty? .............................................. 58
type-exception-arg-num ......................... 128
type-exception-arguments ....................... 128
type-exception-procedure ....................... 128
type-exception-type-id ......................... 128
type-exception? ................................ 128

U
ul6vector ........................................ 78
ul6vector->list ................................ 78
ul6vector-append ............................... 78
ul6vector-copy ................................. 78
ul6vector-fill! ................................ 78
ul6vector-length ............................... 78
ul6vector-ref ................................... 78
ul6vector-set! ................................ 78
ul6vector-shrink! ............................... 78
ul6vector? ....................................... 78
u32vector ....................................... 79
u32vector->list ................................ 79
u32vector-append ............................... 79
u32vector-copy ................................. 79
u32vector-fill! ................................ 79
u32vector-length ............................... 79
u32vector-ref ................................... 79
u32vector-set! ................................ 79
u32vector-shrink! ............................... 79
u64vector ....................................... 79
u64vector->list ................................ 80
u64vector-append ............................... 80
u64vector-copy ................................. 80
u64vector-fill! ................................ 80
u64vector-length ............................... 80
u64vector-ref ................................... 80
u64vector-set! ................................ 80
u64vector-shrink! ............................... 80
u64vector? ....................................... 79
u8vector ........................................ 77
u8vector->list ................................ 77
u8vector->object ................................ 81
u8vector-append ............................... 77
u8vector-copy ................................ 77
u8vector-fill! ................................ 77
u8vector-length ............................... 77
u8vector-ref ................................... 77
u8vector-set! ................................ 77
u8vector-shrink! ............................... 78
u8vector? ....................................... 77
unbound-global-exception-code ............... 127
unbound-global-exception-rte ................. 127
unbound-global-exception-variable .......... 127
unbound-global-exception? ..................... 127
unbound-os-environment-variable-exception-arguments .......... 118
unbound-os-environment-variable-exception-procedure ........ 118
unbound-os-environment-variable-exception? .................. 118
unbound-serial-number-exception-arguments ........ 82
unbound-serial-number-exception-procedure .......... 82
unbound-serial-number-exception? ........................................ 82
unbound-table-key-exception-arguments ................................. 90
unbound-table-key-exception-procedure ................................ 90
unbound-table-key-exception? ............................................. 90
unbox ............................................................... 47
unbreak ............................................................. 36
uncaught-exception-arguments ............................................. 121
uncaught-exception-procedure ............................................. 121
uncaught-exception? ...................................................... 121
uninitialized-thread-exception-arguments ............................. 132
uninitialized-thread-exception-procedure ............................... 132
uninitialized-thread-exception? ......................................... 132
uninterned-keyword? ..................................................... 48
uninterned-symbol? ...................................................... 48
unknown-keyword-argument-exception-arguments .................... 132
unknown-keyword-argument-exception-procedure ....................... 132
unknown-keyword-argument-exception? ................................ 132
unterminated-process-exception-arguments ............................. 171
unterminated-process-exception-procedure ............................... 171
unterminated-process-exception? ........................................ 171
untrace ............................................................. 33
user-info ............................................................ 146
user-info-gid ......................................................... 146
user-info-home ....................................................... 147
user-info-name ....................................................... 146
user-info-shell ....................................................... 147
user-info-uid ....................................................... 146
user-info? .......................................................... 146
user-name .......................................................... 146

V
vector-append ............................................................... 45
vector-copy .............................................................. 46
vector-shrink! ............................................................ 48
void ................................................................. 84

W
weak references .......................................................... 82
will-execute! ............................................................ 84
will-testator ............................................................. 84
will? ................................................................. 84
with-exception-catcher ................................................... 115
with-exception-handler .................................................. 114
with-input-from-file ..................................................... 166
with-input-from-port ................................................... 168
with-input-from-string .................................................... 178
with-input-from-u8vector .................................................. 179
with-input-from-vector ................................................... 175
with-output-to-file ...................................................... 166
with-output-to-port ..................................................... 168
with-output-to-process ................................................... 178
with-output-to-string ..................................................... 179
write ................................................................. 156
write-char ............................................................ 160
write-substring ........................................................ 161
write-subu8vector ...................................................... 165
write-u8 ............................................................. 165
wrong-number-of-arguments-exception-arguments .................... 130
wrong-number-of-arguments-exception-procedure .................... 130
wrong-number-of-arguments-exception? ............................... 130

Chapter 21: General index 247
# Table of Contents

1 The Gambit-C system .......................... 1  
   1.1 Accessing the system files ...................... 1  

2 The Gambit Scheme interpreter ............... 3  
   2.1 Interactive mode .................................. 3  
   2.2 Batch mode ...................................... 4  
   2.3 Customization ................................... 4  
   2.4 Process exit status ................................ 5  
   2.5 Scheme scripts .................................. 6  
      2.5.1 Scripts under UNIX and Mac OS X ............... 7  
      2.5.2 Scripts under Microsoft Windows ............... 7  
      2.5.3 Compiling scripts .......................... 8  

3 The Gambit Scheme compiler ................... 9  
   3.1 Interactive mode ................................ 9  
   3.2 Customization ................................... 9  
   3.3 Batch mode ...................................... 9  
   3.4 Link files ...................................... 12  
      3.4.1 Building an executable program ............... 13  
      3.4.2 Building a loadable library .................. 15  
      3.4.3 Building a shared-library .................... 18  
      3.4.4 Other compilation options .................... 18  
   3.5 Procedures specific to compiler ................ 19  

4 Runtime options .............................. 23  

5 Debugging .................................... 27  
   5.1 Debugging model ................................ 27  
   5.2 Debugging commands ............................. 28  
   5.3 Debugging example ............................... 31  
   5.4 Procedures related to debugging ............... 32  
   5.5 Console line-editing ............................. 38  
   5.6 Emacs interface ................................. 40  
   5.7 GUIDE ......................................... 41  

6 Scheme extensions ............................ 42  
   6.1 Extensions to standard procedures ............... 42  
   6.2 Extensions to standard special forms ............... 42  
   6.3 Miscellaneous extensions ....................... 44  
   6.4 Undocumented extensions ....................... 54
## 15 Exceptions ........................................... 114

15.1 Exception-handling .................................. 114
15.2 Exception objects related to memory management ..... 116
15.3 Exception objects related to the host environment ..... 117
15.4 Exception objects related to threads ..................... 119
15.5 Exception objects related to C-interface ................. 122
15.6 Exception objects related to the reader ................. 125
15.7 Exception objects related to evaluation and compilation . 126
15.8 Exception objects related to type checking .............. 128
15.9 Exception objects related to procedure call ............. 130
15.10 Other exception objects ............................... 133

## 16 Host environment ................................. 134

16.1 Handling of file names ............................... 134
16.2 Filesystem operations ................................ 136
16.3 Shell command execution ............................. 138
16.4 Process termination .................................. 139
16.5 Command line arguments ............................. 139
16.6 Environment variables ................................. 139
16.7 Measuring time ....................................... 140
16.8 File information ..................................... 141
16.9 Group information .................................... 145
16.10 User information ..................................... 146
16.11 Host information ..................................... 147
16.12 Service information ................................ 150
16.13 Protocol information ................................ 151
16.14 Network information ................................. 152

## 17 I/O and ports ........................................ 154

17.1 Unidirectional and bidirectional ports .................. 154
17.2 Port classes .......................................... 154
17.3 Port settings ......................................... 155
17.4 Object-ports .......................................... 155
17.4.1 Object-port settings .............................. 155
17.4.2 Object-port operations ............................ 155
17.5 Character-ports ....................................... 158
17.5.1 Character-port settings ............................ 158
17.5.2 Character-port operations .......................... 159
17.6 Byte-ports ............................................ 161
17.6.1 Byte-port settings ................................. 162
17.6.2 Byte-port operations ............................... 165
17.7 Device-ports .......................................... 166
17.7.1 Filesystem devices ................................. 166
17.7.2 Process devices .................................... 168
17.7.3 Network devices ................................... 171
17.8 Directory-ports ....................................... 174
17.9 Vector-ports .......................................... 175
18 Lexical syntax and readtables .......... 181
  18.1 Readtables ........................................ 181
  18.2 Boolean syntax .................................. 188
  18.3 Character syntax .................................. 188
  18.4 String syntax ..................................... 189
  18.5 Symbol syntax .................................... 190
  18.6 Keyword syntax ................................... 190
  18.7 Box syntax ........................................ 190
  18.8 Number syntax .................................... 191
  18.9 Homogeneous vector syntax ..................... 191
  18.10 Special #! syntax ................................. 191
  18.11 Multiline comment syntax ....................... 191
  18.12 Scheme infix syntax extension ............... 191
      18.12.1 SIX grammar .................................. 192
      18.12.2 SIX semantics .................................. 197

19 C-interface ................................. 199
  19.1 The mapping of types between C and Scheme .... 199
  19.2 The c-declare special form ...................... 207
  19.3 The c-initialize special form .................... 208
  19.4 The c-lambda special form ....................... 208
  19.5 The c-define special form ....................... 210
  19.6 The c-define-type special form .................. 211
  19.7 Continuations and the C-interface .............. 218

20 System limitations ..................... 220

21 Copyright and license ..................... 221

General index ............................. 236