MIT Scheme User's Manual

Edition 1.89
for Scheme Release 7.7.1
17 June 2002

by Stephen Adams
Chris Hanson
and the MIT Scheme Team

Introduction

This document describes how to install and use MIT Scheme, the UnCommon Lisp. It gives installation instructions for all of the platforms that we support; complete documentation of the command-line options and environment variables that control how Scheme works; and rudimentary descriptions of how to interact with the evaluator, compile and debug programs, and use the editor. The release notes are included as an appendix.

This document discusses many operating-system specific features of the MIT Scheme implementation. In order to simplify the discussion, we use abbreviations to refer to some operating systems. When the text uses the term *unix*, this means any of the unix systems that we support, including GNU/Linux, *BSD, HP-UX, Ultrix, NeXT, and SunOS. The term *OS/2* means the IBM OS/2 operating system, version 2.1 or later. We use the term *Windows* to collectively refer to the Microsoft Windows 32-bit operating systems: Windows 95, Windows 98, Windows ME, Windows NT, Windows 2000, and Windows XP. We use the term *PC* to refer to any computer running OS/2 or Windows. Thus we consider a PC to be a system with a DOS-like file system, using backslashes for directory separators, drive letters, CR-LF line termination, and (potentially) the hideous 8.3 short filenames.

The primary distribution site for this software is

> http://www.swiss.ai.mit.edu/projects/scheme/
> ftp://ftp.swiss.ai.mit.edu/pub/mit-scheme/

Although our software is distributed from other sites and in other media, the complete distribution and the most recent release is always available at our site.

To report bugs, send email to 'bug-cscheme@zurich.ai.mit.edu'. Please include the output of the `identify-world` procedure (see Section 2.1 [Basics of Starting Scheme], page 9), so we know what version of the system you are using.

1 Installation

This chapter describes how to install MIT Scheme release 7.7. The release is supported under several different operating systems: unix, OS/2, and Windows. Read the section detailing the installation for the operating system that you are using.

1.1 Unix Installation

We will use as an example the installation for GNU/Linux. The installation for other unix systems is similar.

MIT Scheme is distributed as a compressed 'tar' file. The tar file contains two directories, called 'bin' and 'lib'. The 'bin' directory contains two executable files, 'scheme' and 'bchscheme'. The 'lib' directory contains one subdirectory, 'lib/mit-scheme', that Scheme uses while it is executing.

The goal of the installation is to put the executable files in a directory where they will be executed as commands, and to put the library files in some convenient place where Scheme can find them.

There are two ways to install this software: the conventional way in '/usr/local', and the alternative way, in locations of your choice. We encourage you to install this software in '/usr/local' if possible.

To install the software in '/usr/local', do the following

```
cd /usr/local
rm -f bin/scheme bin/bchscheme
rm -rf lib/mit-scheme
gzip -cd scheme-7.7.1-ix86-gnu-linux.tar.gz | tar xvf -
```

After executing these commands, the executable files will be in '/usr/local/bin', and the library files will be in '/usr/local/lib/mit-scheme'. No further configuration is required.

To install the files in directories of your choice:

- First unpack the distribution:
  ```
  mkdir temp
  cd temp
  gzip -cd scheme-7.7.1-ix86-gnu-linux.tar.gz | tar xvf -
  ```
- Next, move the contents of the 'bin' directory to somewhere convenient that is on your execution path. For example, if you had a directory '~/bin' on your path, you would do this:
  ```
  mv bin/* ~/bin/.
  ```
- Next, move or copy the 'mit-scheme' directory somewhere convenient. For example, you could move it to your home directory:
  ```
  mv lib/mit-scheme ~/.
  ```
  Note that if you have unpacked the distribution on a different drive than the one you plan to store the 'mit-scheme' directory on, you must use the command 'cp -pr' rather than 'mv'.
- Next, you must tell Scheme where to find the 'mit-scheme' directory. This can be done in one of two ways. The first way is to bind the environment variable MITSCHEME_LIBRARY_PATH to the full path to the directory, e.g. in bash you would do

```
        export MITSCHEME_LIBRARY_PATH=~/mit-scheme
```

You should put this environment-variable binding in one of your shell init files, e.g. for `bash` it might go in the '`.bashrc`' file.

The second way is to use a command-line argument when invoking Scheme, e.g.

```
        scheme -library ~/mit-scheme
```

- You should now be able to run MIT Scheme. See Chapter 2 [Running Scheme], page 9, for more information.

1.2 Windows Installation

This section describes how to install MIT Scheme on Windows 95, Windows 98, Windows Me, Windows NT 4.0, Windows 2000, or Windows XP. The software should also work on older versions of Windows NT, but we haven't tested it there.

MIT Scheme is distributed as a self-installing executable. Installation of the software is straightforward. Simply execute the downloaded file and answer the installer's questions. The installer will allow you to choose the directory in which MIT Scheme is to be installed, and the name of the folder in which the shortcuts are to be placed.

To uninstall the software, open up the '`Control Panel`', run '`Add/Remove Programs`', and double-click on '`MIT Scheme 7.7`'.

1.3 OS/2 Installation

This section describes how to install MIT Scheme on a machine running OS/2 2.1 or later. This release of MIT Scheme has been tested only on OS/2 Warp 4.0. It was compiled using IBM Visual Age C++ version 3.0 and the OS/2 Toolkit version 4.0.

1.3.1 OS/2 Installation Procedure

After unpacking the ZIP file, '`os2.zip`', you will have these directories containing the following files:

'`exe`'   The executable programs '`scheme.exe`' and '`bchschem.exe`'.

'`dll`'   The dynamic link libraries '`blowfish.dll`', '`gdbm.dll`', and '`md5.dll`'.

'`doc`'   Documentation files in HTML.

'`lib`'   A directory containing the data files needed by Scheme when it is running.

Perform the following steps to install Scheme:

1. Move the executable files '`scheme.exe`' and '`bchschem.exe`' from '`exe`' to any directory that appears in your PATH environment variable. You may either add the '`exe`' directory to your path by editing '`config.sys`' and rebooting, or you may move the files in '`exe`' to an existing directory that is already on your PATH.

   Depending on your needs, you may want to keep only one of these files; chances are you'll only be using one of them. Of course, you may also keep both programs around if you think you might use them both. See Section 2.3 [Memory Usage], page 10, for more information about the tradeoffs between these two programs.

2. Move the dynamic link libraries from 'dll' to any directory that appears in your LIBPATH environment variable. As above, you may either add 'dll' to your LIBPATH, or move the files in 'dll' to a directory that is already on your LIBPATH.

3. You may move the 'lib' directory anywhere you like. You may rename it to anything you like. (Here at MIT, we use 'c:\scheme\lib'.) After you have chosen where it will be located, set the MITSCHEME_LIBRARY_PATH environment variable in 'config.sys' to be that location.

   For example, if you decide to store the directory as 'c:\schdata', you would add the following to 'config.sys':

   ```
   SET MITSCHEME_LIBRARY_PATH=C:\SCHDATA
   ```

   (Remember that you must reboot OS/2 after editing 'config.sys' in order for the changes to take effect.)

   You can override the setting of this environment variable with the -library command-line option to Scheme, for example:

   ```
   scheme -library d:\myscm\mylib
   ```

   If you supply a -library option, it is not necessary to have the environment variable defined. For example, instead of editing 'config.sys', you might create a '.cmd' file to invoke Scheme and pass it the -library option automatically.

4. *Optional:* Move the 'doc' directory anywhere you like, or delete it if you do not want to keep the documentation.

5. *Optional:* Consider setting some of the environment variables described below.

1.3.2 Environment Variables

This section documents several of the environment variables that Scheme uses, and gives an example for each one. These are the environment variables that must usually be defined when installing Scheme under OS/2. For complete documentation of all of the environment variables used by Scheme, see Section 2.6 [Environment Variables], page 16.

Note that environment variables are usually defined in the OS/2 'config.sys' file. After editing the 'config.sys' file, it is necessary to reboot OS/2 before the changes will take effect.

MITSCHEME_LIBRARY_PATH
          says where to find Scheme's data files. This is the only required environment variable (but is not required when Scheme is invoked with the -library command-line option).

          ```
          SET MITSCHEME_LIBRARY_PATH=C:\SCHEME\LIB
          ```

MITSCHEME_INF_DIRECTORY
          tells Scheme where to find debugging information for the runtime system. The default value for this environment variable is a subdirectory 'src' located in the directory specified by MITSCHEME_LIBRARY_PATH.

          ```
          SET MITSCHEME_INF_DIRECTORY=C:\SCHEME\LIB\SRC
          ```

TMPDIR   tells Scheme the name of a directory where it can store temporary files.

          ```
          SET TMPDIR=C:\TMP
          ```

HOME   tells Scheme where your "home" directory is located. This is where Scheme
              looks for init files, and it is also what the '~/' (or '~\\') filename prefix expands
              to. If not specified, Scheme uses the root directory of the current drive.

                   SET HOME=C:\CPH

USER   tells Scheme your user name. This is used for several purposes, including the
              name that will be used as your email address.

                   SET USER=cph

SHELL   tells Edwin what shell program to use in shell buffers and for running shell
              commands. If not specified, this defaults to the standard OS/2 shell, 'cmd.exe'.

                   SET SHELL=C:\4OS2251\4OS2.EXE

1.4 Optional Configuration

As distributed, Scheme contains several large files. You might not need all of them, so
this section will tell you what each is for so that you can decide if you want to delete some
of them. Also, we will discuss the two different Scheme executables, which are each useful
in different situations; you should read this to decide which is right for you.

The Scheme runtime environment and associated tools are normally stored in *bands*,
which are large memory images stored in files (see Section 3.3 [World Images], page 28).
Scheme requires at least one band to work properly. The Scheme distribution includes two
bands. These bands are stored in the 'lib/mit-scheme' directory on unix systems, and the
'lib' directory on PC systems.

'runtime.com'
              The is the basic world image, and the smallest. It contains just the runtime
              files. This is the band that is chosen when Scheme is invoked with no special
              command-line options.

'all.com'   This contains the runtime files, the native-code compiler, and Edwin. This
              band is chosen when either the -compiler or -edwin command-line options are
              supplied.

Depending on your needs, you may not need both of these files. For example, if
you always want the full development environment, you might keep 'all.com' and delete
'runtime.com'. Remember that you must keep at least one of these files to use Scheme.

In addition to bands, Scheme is distributed with two executable programs: 'scheme'
(called 'scheme.exe' on PC systems), and 'bchscheme' (called 'bchschem.exe' on PC sys-
tems). Normally you will need only one of these files.

The only difference between these two programs is in how they handle garbage collection.
'scheme' allocates two memory heaps, and copies objects between the heaps to preserve
them. This means that most of the time the other heap is occupying valuable memory but
doesn't hold any interesting data. 'bchscheme' allocates only one memory heap, creates a
disk file during garbage collection, copies objects into the file, then copies them back into
memory.

These programs provide you with some important performance trade-offs. If you have
plenty of memory and want the best performance, use 'scheme'. If you don't have enough

memory, or if you want to use less memory and will accept slower performance, use 'bchscheme'. One way to tell that you don't have enough memory is to run 'scheme' for a while and see if your machine is paging during garbage collection.

You might consider trying to use 'scheme' and letting the operating system's paging handle the lack of RAM. But usually you will find that using 'bchscheme' without paging is much faster than using 'scheme' with paging. Of course, if you are using 'bchscheme' and you're still paging, the best solution is to install more RAM.

2 Running Scheme

This chapter describes how to run MIT Scheme. It also describes how you can customize the behavior of MIT Scheme using command-line options and environment variables.

2.1 Basics of Starting Scheme

Under unix and OS/2, MIT Scheme is invoked by typing

```
scheme
```

at your operating system's command interpreter. Under Windows, MIT Scheme is invoked by double-clicking on a shortcut. In either case, Scheme will load itself and print something like this:

```
Scheme saved on Monday June 17, 2002 at 12:10:46 PM
  Release 7.7.1
  Microcode 14.9
  Runtime 15.1
```

This information, which can be printed again by evaluating

```
(identify-world)
```

tells you the following version information. 'Release' is the release number for the entire Scheme system. This number is changed each time a new version of Scheme is released. 'Microcode' is the version number for the part of the system that is written in C. 'Runtime' is the version number for the part of the system that is written in Scheme.

Following this there may be additional version numbers for specific subsystems. 'SF' refers to the scode optimization program sf, 'Liar' is the native-code compiler, 'Edwin' is the Emacs-like text editor, and '6.001' is the SICP compatibility package.

You can load the compiler by giving Scheme the -compiler option:

```
scheme -compiler
```

This option causes Scheme to use a larger constant space and heap, and to load the world image containing the compiler.

2.2 Customizing Scheme

You can customize your setup by using a variety of tools:

- *Command-line options.* Many parameters, like memory usage and the location of libraries, may be varied by command-line options. See Section 2.4 [Command-Line Options], page 11.

- *Command scripts or batch files.* You might like to write scripts that invoke Scheme with your favorite command-line options. For example, you might not have enough memory to run Edwin or the compiler with its default memory parameters (it will print something like "Not enough memory for this configuration" and halt when started), so you can write a shell script (unix), '.bat' file (Windows), or '.cmd' file (OS/2) that will invoke Scheme with the appropriate -heap and other parameters.

- Scheme supports *init files*: an init file is a file containing Scheme code that is loaded when Scheme is started, immediately after the identification banner, and before the input prompt is printed. This file is stored in your home directory, which is normally specified by the `HOME` environment variable. Under unix, the file is called '`.scheme.init`'; on the PC it is called '`scheme.ini`'.

  In addition, when Edwin starts up, it loads a separate init file from your home directory into the Edwin environment. This file is called '`.edwin`' under unix, and '`edwin.ini`' on the PC (see Section 7.1 [Starting Edwin], page 59).

  You can use both of these files to define new procedures or commands, or to change defaults in the system.

  The `-no-init-file` command-line option causes Scheme to ignore the '`.scheme.init`' file (see Section 2.4 [Command-Line Options], page 11).
- *Environment variables.* Most microcode parameters, and some runtime system and Edwin parameters, can be specified by means of environment variables. See Section 2.6 [Environment Variables], page 16.
- *Icons.* Under OS/2 and Windows, and with some window managers under X11, it is possible to create icons that invoke Scheme with different parameters.

2.3 Memory Usage

Some of the parameters that can be customized determine how much memory Scheme uses and how that memory is used. This section describes how Scheme's memory is organized and used; subsequent sections describe command-line options and environment variables that you can use to customize this usage for your needs.

Scheme uses four kinds of memory:

- A *stack* that is used for recursive procedure calls.
- A *heap* that is used for dynamically allocated objects, like '`cons`' cells and strings. Storage used for objects in the heap that become unreferenced is eventually reclaimed by *garbage collection*.
- A *constant space* that is used for allocated objects, like the heap. Unlike the heap, storage used for objects in constant space is not reclaimed by garbage collection. Constant space is used for objects that are essentially permanent, like procedures in the runtime system.
- Some extra storage that is used by the microcode (the part of the system that is implemented in C).

All kinds of memory except the last may be controlled either by command-line options or by environment variables.

MIT Scheme uses a two-space copying garbage collector for reclaiming storage in the heap. There are two versions of Scheme which handle garbage collection differently. The standard Scheme, called '`scheme`' under unix and '`scheme.exe`' on the PC, has two heaps, one for each "space". An alternative, called '`bchscheme`' under unix and '`bchschem.exe`' on the PC, has one heap and uses a disk file for the other "space", thus trading memory usage against garbage collection speed (see Section 1.4 [Optional Configuration], page 6).

The total storage required by '`scheme`' is:

$$stack\ +\ (constant\ +\ 2*heap)\ +\ extra$$

where *stack*, *constant* and *heap* are parameters that are selected when 'scheme' starts. For 'bchscheme', which has only one heap in memory, the equation is

$$stack\ +\ (constant\ +\ heap)\ +\ extra$$

Once the storage is allocated for the constant space and the heap, Scheme will dynamically adjust the proportion of the total that is used for constant space; the stack and extra microcode storage is not included in this adjustment. Previous versions of MIT Scheme needed to be told the amount of constant space that was required when loading bands with the -band option. Dynamic adjustment of the heap and constant space avoids this problem.

If the size of the constant space is not specified, it is automatically set to the correct size for the band being loaded. Thus, in general it is rarely necessary to explicitly set the size of the constant space. Additionally, each band requires a small amount of heap space; this amount is added to any specified heap size, so that the specified heap size is the amount of free space available.

The Scheme expression '(print-gc-statistics)' shows how much heap and constant space is available (see Section 3.4 [Garbage Collection], page 29).

2.4 Command-Line Options

Scheme accepts the command-line options detailed in the following sections. The options may appear in any order, with the restriction that the microcode options must appear before the runtime options, and the runtime options must appear before any other arguments on the command line. Any arguments other than these options will generate a warning message when Scheme starts. If you want to define your own command-line options, see Section 2.5 [Custom Command-line Options], page 15.

These are the microcode options:

-compiler
>   This option specifies defaults appropriate for loading the compiler. It specifies the use of large sizes, exactly like -large; if the -band option is also specified, that is the only effect of this option. Otherwise, the default band's filename is the value of the environment variable MITSCHEME_COMPILER_BAND, if defined, or 'compiler.com'; the library directories are searched to locate this file.

-edwin   This option specifies defaults appropriate for loading the editor. It specifies the use of large sizes, exactly like -large; if the -band option is also specified, that is the only effect of this option. Otherwise, the default band's filename is the value of the environment variable MITSCHEME_EDWIN_BAND, if defined, or 'edwin.com'; the library directories are searched to locate this file.

-compiler -edwin
>   If both the -compiler and -edwin options are given, Scheme will load an environment containing both the compiler and the editor. The default band's filename is the value of the environment variable MITSCHEME_ALL_BAND, if defined, or 'all.com'; the library directories are searched to locate this file.

-band *filename*
>   Specifies the initial world image file (*band*) to be loaded. Searches for *filename* in the working directory and the library directories, using the full pathname of

the first readable file of that name. If *filename* is an absolute pathname (on unix, this means it starts with '`/`'), then no search occurs — *filename* is tested for readability and then used directly. If this option isn't given, the filename is the value of the environment variable `MITSCHEME_BAND`, or if that isn't defined, '`runtime.com`'; in these cases the library directories are searched, but not the working directory.

`-large`   Specifies that large heap, constant, and stack sizes should be used. These are specified by the environment variables

> `MITSCHEME_LARGE_HEAP`
> `MITSCHEME_LARGE_CONSTANT`
> `MITSCHEME_LARGE_STACK`

If this option isn't given, the small sizes are used, specified by the environment variables

> `MITSCHEME_SMALL_HEAP`
> `MITSCHEME_SMALL_CONSTANT`
> `MITSCHEME_SMALL_STACK`

There are reasonable built-in defaults for all of these environment variables, should any of them be undefined. Note that any or all of the defaults can be individually overridden by the `-heap`, `-constant`, and `-stack` options.

Note: the Scheme expression '`(print-gc-statistics)`' shows how much heap and constant space is available and in use (see Section 3.4 [Garbage Collection], page 29).

`-heap` *blocks*

Specifies the size of the heap in 1024-word blocks. Overrides any default. Normally two such heaps are allocated; '`bchscheme`' allocates only one, and uses a disk file for the other.

The size specified by this option is incremented by the amount of heap space needed by the band being loaded. Consequently, `-heap` specifies how much free space will be available in the heap when Scheme starts, independent of the amount of heap already consumed by the band.

`-constant` *blocks*

Specifies the size of constant space in 1024-word blocks. Overrides any default. Constant space holds the compiled code for the runtime system and other subsystems.

`-stack` *blocks*

Specifies the size of the stack in 1024-word blocks. Overrides any default. This is Scheme's stack, *not* the unix stack used by C programs.

`-option-summary`

Causes Scheme to write an option summary to standard error. This shows the values of all of the settable microcode option variables.

`-emacs`   Specifies that Scheme is running as a subprocess of GNU Emacs. This option is automatically supplied by GNU Emacs, and should not be given under other circumstances.

-interactive

> If this option isn't specified, and Scheme's standard i/o is not a terminal, Scheme will detach itself from its controlling terminal, which prevents it from getting signals sent to the process group of that terminal. If this option is specified, Scheme will not detach itself from the controlling terminal.
>
> This detaching behavior is useful for running Scheme as a background job. For example, using `bash`, the following will run Scheme as a background job, redirecting its input and output to files, and preventing it from being killed by keyboard interrupts or by logging out:
>
> ```
> scheme < /usr/cph/foo.in > /usr/cph/foo.out 2>&1 &
> ```
>
> This option is ignored under non-unix operating systems.

-nocore

> Specifies that Scheme should not generate a core dump under any circumstances. If this option is not given, and Scheme terminates abnormally, you will be prompted to decide whether a core dump should be generated.
>
> This option is ignored under non-unix operating systems.

-library *path*

> Sets the library search path to *path*. This is a list of directories that is searched to find various library files, such as bands. If this option is not given, the value of the environment variable `MITSCHEME_LIBRARY_PATH` is used; if that isn't defined, the default is used.
>
> On unix, the elements of the list are separated by colons, and the default value is '`/usr/local/lib/mit-scheme`'. On PCs, the elements of the list are separated by semicolons, and the default value is '`c:\scheme\lib`'.

-utabmd *filename*

> Specifies that *filename* contains the microcode tables (the microcode tables are information that informs the runtime system about the microcode's structure). *Filename* is searched for in the working directory and the library directories. If this option isn't given, the filename is the value of the environment variable `MITSCHEME_UTABMD_FILE`, or if that isn't defined, '`utabmd.bin`'; in these cases the library directories are searched, but not the working directory.
>
> `-utab` is an alternate name for the `-utabmd` option; at most one of these options may be given.

-fasl *filename*

> Specifies that a *cold load* should be performed, using *filename* as the initial file to be loaded. If this option isn't given, a normal load is performed instead. This option may not be used together with the `-compiler`, `-edwin`, or `-band` options. This option is useful only for maintenance and development of the MIT Scheme runtime system.

In addition to the above, '`bchscheme`' recognizes the following command-line options, all of which specify parameters affecting how '`bchscheme`' uses disk storage to do garbage collection:

-gc-directory *directory*

> Specifies that *directory* should be used to create files for garbage collection. If the option is not given, the value of environment variable `MITSCHEME_GC_`

DIRECTORY is used instead, and if that is not defined, a standard temporary directory is used (see TMPDIR in see Section 2.6.3 [Runtime Environment Variables], page 18).

-gc-file *filename*

Specifies that *filename* should be used for garbage collection. If the option is not given, the value of environment variable MITSCHEME_GC_FILE is used, and if this is not defined, a unique filename is generated in the directory specified with -gc-directory.

-gcfile is an alias for -gc-file; at most one of these options should be specified.

-gc-keep   Specifies that the GC file used for garbage collection should not be deleted when Scheme terminates. The GC file is deleted only if the file was created by this invocation of Scheme, and this option is not set.

-gc-start-position *number*

Specifies the first byte position in the GC file at which the Scheme process can write. If not given, the value of the environment variable MITSCHEME_GC_START_POSITION is used, and if that is not defined, '0' is used, meaning the beginning of the file. The area of the file used (and locked if possible) is the region between -gc-start-position and -gc-end-position.

-gc-end-position *number*

Specifies the last byte position in the GC file at which the Scheme process can write. If not given, the value of the environment variable MITSCHEME_GC_END_POSITION is used, and if that is not defined, the sum of the start position (as specified by -gc-start-position) and the heap size is used. The area of the file used (and locked if possible) is the region between -gc-start-position and -gc-end-position.

-gc-window-size *blocks*

Specifies the size of the windows into new space during garbage collection. If this option is not given, the value of environment variable MITSCHEME_GC_WINDOW_SIZE is used instead, and if that is not defined, the value '16' is used.

The following command-line options are only used by an experimental version of 'bchscheme' that uses unix System V-style shared memory, and then only if the 'gcdrone' program is installed in the library directory.

-gc-drone *program*

Specifies that *program* should be used as the drone program for overlapped I/O during garbage collection. If the option is not given, the value of environment variable MITSCHEME_GC_DRONE is used instead, and if that is not defined, 'gcdrone' is used.

-gc-read-overlap *n*

Specifies that Scheme should delegate at most *n* simultaneous disk read operations during garbage collection. If the option is not given, the value of environment variable MITSCHEME_GC_READ_OVERLAP is used instead, and if that is not defined, '0' is used, disabling overlapped reads.

`-gc-write-overlap` *n*

>   Specifies that Scheme should delegate at most *n* simultaneous disk write operations during garbage collection. If the option is not given, the value of environment variable `MITSCHEME_GC_WRITE_OVERLAP` is used instead, and if that is not defined, '`0`' is used, disabling overlapped writes.

The following options are runtime options. They are processed after the microcode options and after the image file is loaded.

`-no-init-file`

>   This option causes Scheme to ignore the '`~/.scheme.init`' or '`scheme.ini`' file, normally loaded automatically when Scheme starts (if it exists).

`-suspend-file`

>   Under some circumstances Scheme can write out a file called '`scheme_suspend`' in the user's home directory.[1] This file is a world image containing the complete state of the Scheme process; restoring this file continues the computation that Scheme was performing at the time the file was written.

>   Normally this file is never written, but the `-suspend-file` option enables writing of this file.

`-eval` *expression* `...`

>   This option causes Scheme to evaluate the *expression*s following it on the command line, up to (but not including) the next option that starts with a hyphen. The expressions are evaluated in the `user-initial-environment`. Unless explicitly handled, errors during evaluation are silently ignored.

`-load` *file* `...`

>   This option causes Scheme to load the *file*s (or lists of files) following it on the command line, up to (but not including) the next option that starts with a hyphen. The files are loaded in the `user-initial-environment`. Unless explicitly handled, errors during loading are silently ignored.

The following option is supported only when Edwin is loaded.

`-edit`   This option causes Edwin to start immediately when Scheme is started.

2.5 Custom Command-line Options

MIT Scheme provides a mechanism for you to define your own command-line options. This is done by registering handlers to identify particular named options and to process them when Scheme starts. Unfortunately, because of the way this mechanism is implemented, you must define the options and then save a world image containing your definitions (see Section 3.3 [World Images], page 28). Later, when you start Scheme using that world image, your options will be recognized.

The following procedures define command-line parsers. In each, the argument *keyword* defines the option that will be recognized on the command line. The *keyword* must be a string starting with a hyphen and containing at least one additional character.

---

[1]   Under unix, this file is written when Scheme is terminated by the '`SIGUSR1`', '`SIGHUP`', or '`SIGPWR`' signals. Under other operating systems, this file is never written.

simple-command-line-parser *keyword thunk*   procedure+

> Defines *keyword* to be a simple command-line option. When this keyword is seen on the command line, it causes *thunk* to be executed.

argument-command-line-parser *keyword multiple? procedure*   procedure+

> Defines *keyword* to be a command-line option that is followed by one or more command-line arguments. *Procedure* is a procedure that accepts one argument; when *keyword* is seen, it is called once for each argument.
>
> *Multiple?*, if true, says that *keyword* may be followed by more than one argument on the command line. In this case, procedure is called once for each argument that follows *keyword* and does not start with a hyphen. If *multiple?* is `#f`, *procedure* is called once, with the command-line argument following *keyword*. In this case, it does not matter if the following argument starts with a hyphen.

set-command-line-parser! *keyword procedure*   procedure+

> This low-level procedure defines *keyword* to be a command-line option that is defined by *procedure*. When *keyword* is seen, *procedure* is called with all of the command-line arguments, starting with *keyword*, as a single list argument. *Procedure* must return two values (using the `values` procedure): the unused command-line arguments (as a list), and a thunk that is executed to implement the behavior of the option.

2.6 Environment Variables

Scheme refers to many environment variables. This section lists these variables and describes how each is used. The environment variables are organized according to the parts of MIT Scheme that they affect.

Environment variables that affect the microcode must be defined before you start Scheme; under unix or Windows, others can be defined or overwritten within Scheme by using the `set-environment-variable!` procedure, e.g.

```
(set-environment-variable! "EDWIN_FOREGROUND" "32")
```

2.6.1 Environment Variables for the Microcode

These environment variables are referred to by the microcode (the executable C programs called 'scheme' and 'bchscheme' under unix, and 'scheme.exe' and 'bchschem.exe' on the PC).

MITSCHEME_ALL_BAND (default: 'all.com' on the library path)

> The initial band to be loaded if both the `-compiler` and `-edwin` options are given. Overridden by `-band`.

MITSCHEME_BAND (default: 'runtime.com' on the library path)

> The initial band to be loaded. Overridden by `-band`, `-compiler`, or `-edwin`.

MITSCHEME_COMPILER_BAND (default: 'compiler.com' on the library path)

> The initial band to be loaded if the `-compiler` option is given. Overridden by `-band`.

MITSCHEME_EDWIN_BAND (default: 'edwin.com' on the library path)
>   The initial band to be loaded if the -edwin option is given. Overridden by
>   -band.

MITSCHEME_LARGE_CONSTANT (default: as needed)
>   The size of constant space, in 1024-word blocks, if the -large, -compiler,
>   or -edwin options are given. Overridden by -constant. Note: the default is
>   computed to be the correct size for the band being loaded.

MITSCHEME_LARGE_HEAP (default: '1000')
>   The size of the heap, in 1024-word blocks, if the -large, -compiler, or -edwin
>   options are given. Overridden by -heap.

MITSCHEME_LARGE_STACK (default: '100')
>   The size of the stack, in 1024-word blocks, if the -large, -compiler, or -edwin
>   options are given. Overridden by -stack.

MITSCHEME_LIBRARY_PATH
>   A list of directories. These directories are searched, left to right, to find bands
>   and various other files. On unix systems the list is colon-separated, with the
>   default '/usr/local/lib/mit-scheme'. On PC systems the list is semicolon-
>   separated with the default 'c:\scheme\lib'.

MITSCHEME_SMALL_CONSTANT (default: as needed)
>   The size of constant space, in 1024-word blocks, if the size options are not given.
>   Overridden by -constant, -large, -compiler, or -edwin. Note: the default
>   is computed to be the correct size for the band being loaded.

MITSCHEME_SMALL_HEAP (default: '250')
>   The size of the heap, in 1024-word blocks, if the size options are not given.
>   Overridden by -heap, -large, -compiler, or -edwin.

MITSCHEME_SMALL_STACK (default: '100')
>   The size of the stack, in 1024-word blocks, if the size options are not given.
>   Overridden by -stack, -large, -compiler, or -edwin.

MITSCHEME_UTABMD_FILE (default: 'utabmd.bin' in the library path)
>   The file containing the microcode tables. Overridden by -utabmd and -utab.

2.6.2 Environment Variables for 'bchscheme'

These environment variables are referred to by 'bchscheme' (*not* by 'scheme').

MITSCHEME_GC_DIRECTORY
>   The directory in which GC files are written. Overridden by -gc-directory.
>   The default for this variable is the standard temporary directory (see TMPDIR
>   in see Section 2.6.3 [Runtime Environment Variables], page 18).

MITSCHEME_GC_FILE (default: 'GCXXXXXX')
>   The name of the file to use for garbage collection. If it ends in 6 Xs, the Xs
>   are replaced by a letter and process id of the scheme process, thus generating
>   a unique name. Overridden by -gc-file.

MITSCHEME_GC_START_POSITION (default: '0')
> The first position in the GC file to use. Overridden by `-gc-start-position`.

MITSCHEME_GC_END_POSITION (default: *start-position*+*heap-size*)
> The last position in the GC file to use. Overridden by `-gc-end-position`.

MITSCHEME_GC_WINDOW_SIZE (default: '16')
> The size in blocks of windows into new space (in the GC file).
> Overridden by `-gc-window-size`.

The following environment variables are only used by an experimental version of Bchscheme that uses unix System V-style shared memory, and then only if the 'gcdrone' program is installed:

MITSCHEME_GC_DRONE (default: 'gcdrone')
> The program to use as the I/O drone during garbage collection.
> Overridden by `-gc-drone`.

MITSCHEME_GC_READ_OVERLAP (default: '0')
> The maximum number of simultaneous read operations.
> Overridden by `-gc-read-overlap`.

MITSCHEME_GC_WRITE_OVERLAP (default: '0')
> The maximum number of simultaneous write operations.
> Overridden by `-gc-write-overlap`.

2.6.3 Environment Variables for the Runtime System

These environment variables are referred to by the runtime system.

HOME
> Directory in which to look for init files. E.g. 'c:\users\joe' or '/home/joe'.
> This variable needs to be set on OS/2 and Windows 9x. Under Windows
> NT/2000/XP, the environment variables HOMEDRIVE and HOMEPATH, set by the
> operating system, are used instead. Under unix, HOME is set by the login shell.

TMPDIR
TEMP
TMP
> Directory for various temporary files. The variables are tried in the given order.
> If none of them is suitable, built-in defaults are used: under unix, '/var/tmp',
> '/usr/tmp', '/tmp'; under OS/2 and Windows, '\temp', '\tmp', and '\' (all on
> the system drive).

MITSCHEME_INF_DIRECTORY (default: 'SRC' on the library path)
> Directory containing the debugging information files for the Scheme system.
> Should contain subdirectories corresponding to the subdirectories in the source
> tree. For example, if its value is 'f:\random', runtime system debugging files
> will be expected in 'f:\random\runtime', while Edwin debugging files will be
> expected in 'f:\random\edwin'.

MITSCHEME_LOAD_OPTIONS (default: 'optiondb.scm' on the library path)
> Specifies the location of the options database file used by the `load-option`
> procedure.

2.6.4 Environment Variables for Edwin

These environment variables are referred to by Edwin.

EDWIN_BINARY_DIRECTORY (default: '`edwin/autoload`' on the library path)
:   Directory where Edwin expects to find files providing autoloaded facilities.

EDWIN_INFO_DIRECTORY (default: '`edwin/info`' on the library path)
:   Directory where Edwin expects to find files for the 'info' documentation subsystem.

EDWIN_ETC_DIRECTORY (default: '`edwin/etc`' on the library path)
:   Directory where Edwin expects to find utility programs and documentation strings.

ESHELL
:   Filename of the shell program to use in shell buffers. If not defined, the `SHELL` environment variable is used instead.

SHELL (default: '`/bin/sh`' (unix), '`cmd.exe`' (PC))
:   Filename of the shell program to use in shell buffers and when executing shell commands. Used to initialize the `shell-path-name` editor variable.

PATH
:   Used to initialize the `exec-path` editor variable, which is subsequently used for finding programs to be run as subprocesses.

DISPLAY
:   Used when Edwin runs under unix and uses X11. Specifies the display on which Edwin will create windows.

TERM
:   Used when Edwin runs under unix on a terminal. Terminal type.

LINES (default: auto-sense)
:   Used when Edwin runs under unix on a terminal. Number of text lines on the screen, for systems that don't support '`TIOCGWINSZ`'.

COLUMNS (default: auto-sense)
:   Used when Edwin runs under unix on a terminal. Number of text columns on the screen, for systems that don't support '`TIOCGWINSZ`'.

2.6.5 Environment Variables for Microsoft Windows

These environment variables are specific to the Microsoft Windows implementation.

MITSCHEME_FONT (default: determined by operating system)
:   A string specifying a font name and characteristics, for example '`Courier New 16 bold`'. Allowed characteristics are *integer*, specifying the font size in points, and the following style modifiers: '`bold`', '`italic`', '`regular`', '`underline`' and '`strikeout`'. You should specify only fixed-width fonts as variable-width fonts are not drawn correctly.

    Once in Edwin, the font can be changed with the `set-font` and `set-default-font` commands.

MITSCHEME_GEOMETRY (default: '`-1,-1,-1,-1`')
:   Four integers separated by commas or spaces that specify the placement and size of the MIT Scheme window as a *left,top,width,height* quadruple. The units are

screen pixels, and '`-1`' means allow the system to choose this parameter. E.g.
'`-1,-1,500,300`' places a 500 by 300 pixel window at some system-determined
position on the screen. The width and height include the window border and
title.

`MITSCHEME_FOREGROUND` (default: according to desktop color scheme)
A value specifying the window text color. The color is specified as hex blue,
green and red values (*not* RGB): e.g. `0xff0000` for blue.

`MITSCHEME_BACKGROUND` (default: according to desktop color scheme)
A value specifying the window background color. See `MITSCHEME_FOREGROUND`.

`HOMEDRIVE`
`HOMEPATH`   These variables are used together to indicate the user's home directory. This is
the preferred way to specify the home directory.

`USERNAME`
`USER`   Specifies the login name of the user running Scheme. This is used for several
different purposes. `USERNAME` is preferred; `USER` is used if `USERNAME` is not
defined. If neither of these variables is defined, an error is signalled when the
username is required.

`USERDIR`   Specifies a directory that contains the home directories of users. One of the
places in which Scheme looks for the user's home directory, by searching for a
subdirectory with the user's login name.

2.6.6 Environment Variables for OS/2

These environment variables are specific to the OS/2 implementation.

`USER`   Specifies the login name of the user running Scheme. This is used for several
different purposes. If this variable is undefined, an error is signalled when the
username is required.

`USERDIR`   Specifies a directory that contains the home directories of users. One of the
places in which Scheme looks for the user's home directory, by searching for a
subdirectory with the user's login name. This variable is used only when `HOME`
is not defined; we recommend using `HOME` rather than `USERDIR`.

`COMSPEC`   Specifies the command shell. This is set in all versions of OS/2 (and is required
for proper operation of the operating system). Scheme uses this to determine
the user's shell if the environment variable `SHELL` is not defined.

2.7 Starting Scheme from Microsoft Windows

The Microsoft Windows version of MIT Scheme runs as a graphics-based application.
Scheme is normally started using shortcuts; the installer automatically generates several
different predefined shortcuts for your convenience.

The rest of this section gives some tips on how to set up shortcuts that run Scheme. If
you are unfamiliar with this concept you should read about it in the system help.

- Under Windows NT, Windows 2000, or Windows XP, shortcuts can be *common* or *personal*. When setting common shortcuts it is important to make the shortcut properties independent of the vagaries of the environment of the user who is running them.

- Give the shortcut an accurate *Description*.

- Include absolute pathnames to '`scheme.exe`' and '`bchscheme.exe`' in the shortcut *Command line*.

- If you specify the `-library` command-line option then you do not have to worry about the `MITSCHEME_LIBRARY_PATH` environment variable.

- Set the shortcut's *Working Directory* to something sensible. On Windows NT/2000/XP you can use '`%HOMEDRIVE%%HOMEPATH%`' to make Scheme start up in the user's home directory. On Windows 9x/ME you can use '`%HOME%`' to achieve the same effect, provided that you have set the `HOME` environment variable as we recommend.

- There are several icons available in the Scheme executable — choose one that best represents the options given on the command line.

- Specifying a band that contains Edwin is not sufficient to invoke the editor. You also have to put `-edit` at the end of the command line.

2.8 Leaving Scheme

There are several ways that you can leave Scheme: there are two Scheme procedures that you can call; there are several Edwin commands that you can execute; and there are are graphical-interface buttons (and their associated keyboard accelerators) that you can activate.

- *Two Scheme procedures that you can call.* The first is to evaluate

      (exit)

  which will halt the Scheme system, after first requesting confirmation. Any information that was in the environment is lost, so this should not be done lightly.

  The second procedure suspends Scheme; when this is done you may later restart where you left off. Unfortunately this is not possible in all operating systems; currently it works under unix versions that support job control (i.e. all of the unix versions for which we distribute Scheme). To suspend Scheme, evaluate

      (quit)

  If your system supports suspension, this will cause Scheme to stop, and you will be returned to the shell. Scheme remains stopped, and can be continued using the job-control commands of your shell. If your system doesn't support suspension, this procedure does nothing. (Calling the `quit` procedure is analogous to typing `C-z`, but it allows Scheme to respond by typing a prompt when it is unsuspended.)

- *Several Edwin commands that you can execute,* including `save-buffers-kill-scheme`, normally bound to `C-x C-c`, and `suspend-scheme`, normally bound to `C-x C-z`. These two commands correspond to the procedures `exit` and `quit`, respectively.

- *Graphical-interface buttons that you can activate.* Under OS/2 and Windows, closing the console window (Scheme's main window) causes Scheme to be terminated. Under any operating system, closing an Edwin window causes that window to go away, and if it is the only Edwin window, it terminates Scheme as well.

Under OS/2, there are two distinct ways to close the console window. The first is to use any of the usual window-closing methods, such as the 'Close' system-menu item or double-clicking on the system-menu icon. When this is done, you will be presented with a dialog that gives you the option to close the window with or without termating Scheme. The second way is to select the 'Exit' item from the 'File' menu, which terminates Scheme immediately with no dialog.

3 Using Scheme

This chapter describes how to use Scheme to evaluate expressions and load programs. It also describes how to save custom "world images", and how to control the garbage collector. Subsequent chapters will describe how to use the compiler, and how to debug your programs.

3.1 The Read-Eval-Print Loop

When you first start up Scheme from the command line, you will be typing at a program called the *Read-Eval-Print Loop* (abbreviated *REPL*). It displays a prompt at the left hand side of the screen whenever it is waiting for input. You then type an expression (terminating it with ⟨RET⟩). Scheme evaluates the expression, prints the result, and gives you another prompt.

3.1.1 The Prompt and Level Number

The REPL *prompt* normally has the form

```
1 ]=>
```

The '1' in the prompt is a *level number*, which is always a positive integer. This number is incremented under certain circumstances, the most common being an error. For example, here is what you will see if you type *f o o* ⟨RET⟩ after starting Scheme:

```
;Unbound variable: foo
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of foo.
; (RESTART 2) => Define foo to a given value.
; (RESTART 1) => Return to read-eval-print level 1.

2 error>
```

In this case, the level number has been incremented to '2', which indicates that a new REPL has been started (also the prompt string has been changed to remind you that the REPL was started because of an error). The '2' means that this new REPL is "over" the old one. The original REPL still exists, and is waiting for you to return to it, for example, by entering (restart 1). Furthermore, if an error occurs while you are in this REPL, yet another REPL will be started, and the level number will be increased to '3'. This can continue ad infinitum, but normally it is rare to use more than a few levels.

The normal way to get out of an error REPL and back to the top level REPL is to use the *C-g* interrupt. This is a single-keystroke command executed by holding down the ⟨CTRL⟩ key and pressing the ⟨G⟩ key. *C-g* always terminates whatever is running and returns you to the top level REPL immediately.

Note: The appearance of the '`error>`' prompt does not mean that Scheme is in some weird inconsistent state that you should avoid. It is merely a reminder that your program was in error: an illegal operation was attempted, but it was detected and avoided. Often the best way to find out what is in error is to do some poking around in the error REPL. If you abort out of it, the context of the error will be destroyed, and you may not be able to find out what happened.

3.1.2 Interrupting

Scheme has several interrupt keys, which vary depending on the underlying operating system: under unix, `C-g` and `C-c`; under OS/2 and Windows, `C-g`, `C-b`, `C-x` and `C-u`. The `C-g` key stops any Scheme evaluation that is running and returns you to the top level REPL. `C-c` prompts you for another character and performs some action based on that character. It is not necessary to type (RET) after `C-g` or `C-c`, nor is it needed after the character that `C-c` will ask you for.

Here are the definitions of the more common interrupt keys; on unix, type `C-c ?` for more possibilities. Note that in any given implementation, only a subset of the following keys is available.

`C-c C-c`
`C-g`   Abort whatever Scheme evaluation is currently running and return to the top-level REPL. If no evaluation is running, this is equivalent to evaluating

       `(cmdl-interrupt/abort-top-level)`

`C-c C-x`
`C-x`   Abort whatever Scheme evaluation is currently running and return to the "current" REPL. If no evaluation is running, this is equivalent to evaluating

       `(cmdl-interrupt/abort-nearest)`

`C-c C-u`
`C-u`   Abort whatever Scheme evaluation is running and go up one level. If you are already at level number 1, the evaluation is aborted, leaving you at level 1. If no evaluation is running, this is equivalent to evaluating

       `(cmdl-interrupt/abort-previous)`

`C-c C-b`
`C-b`   Suspend whatever Scheme evaluation is running and start a *breakpoint* REPL. The evaluation can be resumed by evaluating

       `(continue)`

       in that REPL at any time.

`C-c q`   Similar to typing `(exit)` at the REPL, except that it works even if Scheme is running an evaluation, and does not request confirmation.

`C-c z`   Similar to typing `(quit)` at the REPL, except that it works even if Scheme is running an evaluation.

`C-c i`   Ignore the interrupt. Type this if you made a mistake and didn't really mean to type `C-c`.

`C-c ?`   Print help information. This will describe any other options not documented here.

3.1.3 Restarting

Another way to exit a REPL is to use the `restart` procedure:

restart [*k*]   procedure+
>   This procedure selects and invokes a *restart method*. The list of restart methods is
>   different for each REPL and for each error; in the case of an error REPL, this list is
>   printed when the REPL is started:

```
;Unbound variable: foo
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of foo.
; (RESTART 2) => Define foo to a given value.
; (RESTART 1) => Return to read-eval-print level 1.

2 error>
```

>   If the *k* argument is given, it must be a positive integer index into the list (in the
>   example it must be between one and three inclusive). The integer *k* selects an item
>   from the list and invokes it. If *k* is not given, `restart` prints the list and prompts for
>   the integer index:

```
2 error> (restart)
;Choose an option by number:
;  3: Specify a value to use instead of foo.
;  2: Define foo to a given value.
;  1: Return to read-eval-print level 1.

Option number:
```

>   The simplest restart methods just perform their actions. For example:

```
2 error> (restart 1)
;Abort!

1 ]=>
```

>   Other methods will prompt for more input before continuing:

```
2 error> (restart)
;Choose an option by number:
;  3: Specify a value to use instead of foo.
;  2: Define foo to a given value.
;  1: Return to read-eval-print level 1.

Option number: 3

Value to use instead of foo: '(a b)
;Value: (a b)

1 ]=>
```

3.1.4 The Current REPL Environment

Every REPL has a *current environment*, which is the place where expressions are evaluated
and definitions are stored. When Scheme is started, this environment is the value of the
variable `user-initial-environment`. There are a number of other environments in the
system, for example `system-global-environment`, where the runtime system's bindings
are stored.

You can get the current REPL environment by evaluating

```
(nearest-repl/environment)
```

There are several other ways to obtain environments. For example, if you have a procedure object, you can get a pointer to the environment in which it was closed by evaluating

```
(procedure-environment procedure)
```

Here is the procedure that changes the REPL's environment:

ge *environment*   procedure+

Changes the current REPL environment to be *environment* (`ge` stands for "Goto Environment"). *Environment* is allowed to be a procedure as well as an environment object. If it is a procedure, then the closing environment of that procedure is used in its place.

pe   procedure+

This procedure is useful for finding out which environment you are in (`pe` stands for "Print Environment"). If the current REPL environment belongs to a package, then `pe` returns the package name (a list of symbols). If the current REPL environment does not belong to a package then the environment is returned.

3.2 Loading Files

To load files of Scheme code, use the procedure `load`:

load *filename* [*environment* [*syntax-table* [*purify?*]]]   procedure

Filename may be a string naming a file, or a list of strings naming multiple files. *Environment*, if given, is the environment to evaluate the file in; if not given the current REPL environment is used.

Syntax-table is no longer used and if supplied will be ignored.

The optional argument *purify?* is a boolean that says whether to move the contents of the file into constant space after it is loaded but before it is evaluated. This is performed by calling the procedure `purify` (see Section 3.4 [Garbage Collection], page 29). If *purify?* is given and true, this is done; otherwise it is not.

`load` determines whether the file to be loaded is binary or source code, and performs the appropriate action. By convention, files of source code have a pathname type of `"scm"`, and files of binary SCode have pathname type `"bin"`. Native-code binaries have pathname type `"com"`. (See the description of `pathname-type` in section "Components of Pathnames" in *MIT Scheme Reference Manual*.)

load-noisily?   variable+

If `load-noisily?` is set to `#t`, `load` will print the value of each expression in the file as it is evaluated. Otherwise, nothing is printed except for the value of the last expression in the file. (Note: the noisy loading feature is implemented for source-code files only.)

load/default-types   variable+

> When load is given a pathname without a type, it uses the value of this variable to determine what pathname types to look for and how to load the file. **load/default-types** is a list of associations that maps pathname types (strings) to loader procedures. **load** tries the pathname types in the order that they appear in the list. The initial value of this variable has pathname types in this order:
>
>   "com" "so" "sl" "bin" "scm"
>
> This means that, for example, (`load "foo"`) will try to load 'foo.com' first, and 'foo.scm' only after looking for and failing to find the other pathname types.

All pathnames are interpreted relative to a working directory, which is initialized when Scheme is started. The working directory can be obtained by calling the procedure `pwd` or modified by calling the procedure `cd`; see section "Working Directory" in *MIT Scheme Reference Manual*. Files may be loaded when Scheme first starts; see the `-load` command-line option for details.

load-option *symbol* [*no-error?*]   procedure+

> Loads the option specified by *symbol*; if already loaded, does nothing. Returns *symbol*; if there is no such option, an error is signalled. However, if *no-error?* is specified and true, no error is signalled in this case, and `#f` is returned.
>
> A number of built-in options are defined:
>
> compress   Support to compress and uncompress files. Undocumented; see the source file 'runtime/cpress.scm'. Used by the runtime system for compression of compiled-code debugging information.
>
> format   The `format` procedure. See section "Format" in *MIT Scheme Reference Manual*.
>
> gdbm   Support to access `gdbm` databases. Undocumented; see the source files 'runtime/gdbm.scm' and 'microcode/prgdbm.c'.
>
> hash-table
>   The hash-table data type. See section "Hash Tables" in *MIT Scheme Reference Manual*.
>
> ordered-vector
>   Support to search and do completion on vectors of ordered elements. Undocumented; see the source file 'runtime/ordvec.scm'.
>
> rb-tree   The red-black tree data type. See section "Red-Black Trees" in *MIT Scheme Reference Manual*.
>
> regular-expression
>   Support to search and match strings for regular expressions. See section "Regular Expressions" in *MIT Scheme Reference Manual*.
>
> stepper   Support to step through the evaluation of Scheme expressions. Undocumented; see the source file 'runtime/ystep.scm'. Used by the Edwin command *step-expression*.

subprocess
> Support to run other programs as subprocesses of the Scheme process. Undocumented; see the source file 'runtime/process.scm'. Used extensively by Edwin.

synchronous-subprocess
> Support to run synchronous subprocesses. See section "Subprocesses" in *MIT Scheme Reference Manual*.

wt-tree   The weight-balanced tree data type. See section "Weight-Balanced Trees" in *MIT Scheme Reference Manual*.

In addition to the built-in options, you may define other options to be loaded by `load-options` by modifying the file 'optiondb.scm' on the library path. An example file is included with the distribution; normally this file consists of a series of calls to the procedure `define-load-option`, terminated by the expression

```
(further-load-options standard-load-options)
```

define-load-option *symbol thunk ...*   procedure+
> Each *thunk* must be a procedure of no arguments. Defines the load option named *symbol*. When the procedure `load-option` is called with *symbol* as an argument, the *thunk* arguments are executed in order from left to right.

3.3 World Images

A *world image*, also called a *band*, is a file that contains a complete Scheme system, perhaps additionally including user application code. Scheme provides a method for saving and restoring world images. The method writes a file containing all of the Scheme code and data in the running process. The file 'runtime.com' that is loaded by the microcode is just such a band. To make your own band, use the procedure `disk-save`.

disk-save *filename* [*identify*]   procedure+
> Causes a band to be written to the file specified by *filename*. The optional argument *identify* controls what happens when that band is restored, as follows:

not specified
> Start up in the top-level REPL, identifying the world in the normal way.

a string   Do the same thing except print that string instead of 'Scheme' when restarting.

the constant #t
> Restart exactly where you were when the call to `disk-save` was performed. This is especially useful for saving your state when an error has occurred and you are not in the top-level REPL.

the constant #f
> Just like #t, except that the runtime system will not perform normal restart initializations; in particular, it will not load your init file.

To restore a saved band, give the `-band` option when starting Scheme. Alternatively, evaluate (`disk-restore` *filename*), which will destroy the current world, replacing it with the saved world. The argument to `disk-restore` may be omitted, in which case it defaults to the filename from which the current world was last restored.

3.4 Garbage Collection

This section describes procedures that control garbage collection. See Section 2.3 [Memory Usage], page 10, for a discussion of how MIT Scheme uses memory.

gc-flip [*safety-margin*]   procedure+

Forces a garbage collection to occur. Returns the number of words of storage available after collection, an exact non-negative integer.

Safety-margin determines the number of words of storage available to system tasks after the need for a garbage collection is detected and before the garbage collector is started. (An example of such a system task is changing the run-light to show "gc" when scheme is running under Emacs.) **Note well**: you should not specify *safety-margin* unless you know what you are doing. If you specify a value that is too small, you can put Scheme in an unusable state.

purify *object* [*pure-space?* [*queue?*]]   procedure+

Moves *object* from the heap into constant space. Has no effect if *object* is already stored in constant space. *Object* is moved in its entirety; if it is a compound object such as a list, a vector, or a record, then all of the objects that *object* points to are also moved to constant space.

There are three important effects associated with moving an object to constant space. The first and most important effect is that the object takes up half as much space, because when in the heap, the system must reserve space for the object in both the active heap and the inactive heap; if the object is in constant space it is not copied and therefore no extra space is required. The second effect is that garbage collection will take less time, because *object* will no longer be copied. The third effect is that the space allocated to *object* is permanently allocated, because constant space is never cleaned; any unreachable objects in constant space remain there until the Scheme process is terminated.

The optional argument *pure-space?* is obsolete; it defaults to `#t` and when explicitly specified should always be `#t`.

The optional argument *queue?*, if `#f`, specifies that *object* should be moved to constant space immediately; otherwise *object* is queued to be moved during the next garbage collection. This argument defaults to `#t`. The reason for queuing these requests is that moving an object to constant space requires a garbage collection to occur, a relatively slow process. By queuing the requests, this overhead is avoided, because moving an object during a garbage collection has no effect on the time of the garbage collection. Furthermore, if several requests are queued, they can all be processed together in one garbage collection, while if done separately they would each require their own garbage collection.

flush-purification-queue!   procedure+

> Forces any pending queued purification requests to be processed. This examines the
> `purify` queue, and if it contains any requests, forces a garbage collection to process
> them. If the queue is empty, does nothing.

print-gc-statistics   procedure+

> Prints out information about memory allocation and the garbage collector. The
> information is printed to the current output port. Shows how much space is "in use"
> and how much is "free", separately for the heap and constant space. The amounts are
> shown in words, and also in 1024-word blocks; the block figures make it convenient
> to use these numbers to adjust the arguments given to the `-heap` and `-constant`
> command-line options. Following the allocation figures, information about the most
> recent 8 garbage collections is shown, in the same format as a GC notification.
>
> Note that these numbers are accurate at the time that `print-gc-statistics` is
> called. In the case of the heap, the "in use" figure shows how much memory has
> been used since the last garbage collection, and includes all live objects as well as
> any uncollected garbage that has accumulated since then. The only accurate way to
> determine the size of live storage is to subtract the value of '`(gc-flip)`' from the
> size of the heap. The size of the heap can be determined by adding the "in use" and
> "free" figures reported by `print-gc-statistics`.
>
> ```
> (print-gc-statistics)
> constant in use:   534121 words =   521 blocks +  617 words
> constant free:        128 words =     0 blocks +  128 words
> heap in use:        34845 words =    34 blocks +   29 words
> heap free:         205530 words =   200 blocks +  730 words
> GC #1: took: 0.13 (81%) CPU time, 0.15 (1%) real time; free: 207210
> ;No value
> ```

set-gc-notification! [*on?*]   procedure+

> Controls whether the user is notified of garbage collections. If *on?* is true, notification
> is enabled; otherwise notification is disabled. If *on?* is not given, it defaults to `#t`.
> When Scheme starts, notification is disabled.
>
> The notification appears as a single line like the following, showing how many garbage
> collections have occurred, the time taken to perform the garbage collection and the
> free storage remaining (in words) after collection.
>
> ```
>     GC #5: took: 0.50 (8%) CPU time, 0.70 (2%) real time; free: 364346
> ```
>
> To operate comfortably, the amount of free storage after garbage collection should be
> a substantial proportion of the heap size. If the CPU time percentage is consistently
> high (over 20%), you should consider running with a larger heap. A rough rule of
> thumb to halve the GC overhead is to take the amount of free storage, divide by 1000,
> and add this figure to the current value used for the `-heap` command-line option.
> Unfortunately there is no way to adjust the heap size without restarting Scheme.

toggle-gc-notification!   procedure+

> Toggles GC notification on and off. If GC notification is turned on, turns it off;
> otherwise turns it on.

4 Compiling Programs

Note: the procedures described in this section are only available when the `-compiler` command-line option is specified.

4.1 Compilation Procedures

cf *filename* [*destination*]   procedure+

> This is the program that transforms a source-code file into native-code binary form. If *destination* is not given, as in
>
>   (cf "foo")
>
> `cf` compiles the file '`foo.scm`', producing the file '`foo.com`' (incidentally it will also produce '`foo.bin`', '`foo.bci`', and possibly '`foo.ext`'). If you later evaluate
>
>   (load "foo")
>
> '`foo.com`' will be loaded rather than '`foo.scm`'.
>
> If *destination* is given, it says where the output files should go. If this argument is a directory, they go in that directory, e.g.:
>
>   (cf "foo" "../bar/")
>
> will take '`foo.scm`' and generate the file '`../bar/foo.com`'. If *destination* is not a directory, it is the root name of the output:
>
>   (cf "foo" "bar")
>
> takes '`foo.scm`' and generates '`bar.com`'.

About the '`.bci`' files: these files contain the debugging information that Scheme uses when you call `debug` to examine compiled code. When you load a '`.com`' file, Scheme remembers where it was loaded from, and when the debugger (or `pp`) looks at the compiled code from that file, it attempts to find the '`.bci`' file in the same directory from which the '`.com`' file was loaded. Thus it is a good idea to leave these files together.

'`.bci`' files are stored in a compressed format. The debugger has to uncompress the files when it looks at them, and on a slow machine this can take a noticeable time. The system takes steps to reduce the impact of this behavior: debugging information is cached in memory, and uncompressed versions of '`.bci`' files are kept around. The default behavior is that a temporary file is created and the '`.bci`' file is uncompressed into it. The temporary file is kept around for a while afterwards, and during that time if the uncompressed '`.bci`' file is needed the temporary file is used. Each such reference updates an 'access time' that is associated with the temporary file. The garbage collector checks the access times of all such temporary files, and deletes any that have not been accessed in five minutes or more. All of the temporaries are deleted automatically when the Scheme process is killed.

Two other behaviors are available. One of them uncompresses the '`.bci`' file each time it is referenced, and the other uncompresses the '`.bci`' file and writes it back out as a '`.bif`' file. The '`.bif`' file remains after Scheme exits. The time interval and the behavior are controlled by the following variables.

save-uncompressed-files?   variable+

> This variable affects what happens when '`.bci`' files are uncompressed. It allows a trade-off between performance and disk space. There are three possible values:

#f
: The uncompressed versions of '.bci' files are never saved. Each time the information is needed the '.bci' file is uncompressed. This option requires the minimum amount of disk space and is the slowest.

automatic
: Uncompressed versions of '.bci' files are kept as temporary files. The temporary files are deleted when Scheme exits, or if they have not been used for a while. This is the default.

#t
: The '.bci' files are uncompressed to permanent '.bif' files. These files remain on disk after Scheme exits, and are rather large - about twice the size of the corresponding '.bci' files. If you choose this option and you are running out of disk space you may delete the '.bif' files. They will be regenerated as needed.

uncompressed-file-lifetime   variable+

The minimum length of time that a temporary uncompressed version of a '.bci' file will stay on disk after it is last used. The time is in milliseconds; the default is '300000' (five minutes).

load-debugging-info-on-demand?   variable+

If this variable is '#f', then printing a compiled procedure will print the procedure's name only if the debugging information for that procedure is already loaded. Otherwise, it will force loading of the debugging information. The default value is #f.

sf *filename* [*destination*]   procedure+

sf is the program that transforms a source-code file into binary SCode form; it is used on machines that do not support native-code compilation. It performs numerous optimizations that can make your programs run considerably faster than unoptimized interpreted code. Also, the binary files that it generates load very quickly compared to source-code files.

The simplest way to use sf is just to say:

        (sf *filename*)

This will cause your file to be transformed, and the resulting binary file to be written out with the same name, but with pathname type "bin". If you do not specify a pathname type on the input file, "scm" is assumed.

Like load, the first argument to sf may be a list of filenames rather than a single filename.

sf takes an optional second argument, which is the filename of the output file. If this argument is a directory, then the output file has its normal name but is put in that directory instead.

4.2 Declarations

Several declarations can be added to your programs to help cf and sf make them more efficient.

4.2.1 Standard Names

Normally, all files have a line

```
(declare (usual-integrations))
```

near their beginning, which tells the compiler that free variables whose names are defined in `system-global-environment` will not be shadowed by other definitions when the program is loaded. If you redefine some global name in your code, for example `car`, `cdr`, and `cons`, you should indicate it in the declaration:

```
(declare (usual-integrations car cdr cons))
```

You can obtain an alphabetically-sorted list of the names that the `usual-integrations` declaration affects by evaluating the following expression:

```
(eval '(sort (append usual-integrations/constant-names
                     usual-integrations/expansion-names)
          (lambda (x y)
            (string<=? (symbol->string x)
                       (symbol->string y))))
      (->environment '(scode-optimizer)))
```

4.2.2 In-line Coding

Another useful facility is the ability to in-line code procedure definitions. In fact, the compiler will perform full beta conversion, with automatic renaming, if you request it. Here are the relevant declarations:

integrate *name . . .*   declaration+

> The variables *name*s must be defined in the same file as this declaration. Any reference to one of the named variables that appears in the same block as the declaration, or one of its descendant blocks, will be replaced by the corresponding binding's value expression.

integrate-operator *name . . .*   declaration+

> Similar to the `integrate` declaration, except that it only substitutes for references that appear in the operator position of a combination. All other references are ignored.

integrate-external *filename*   declaration+

> Causes the compiler to use the top-level integrations provided by *filename*. *filename* should not specify a file type, and the source-code file that it names must have been previously processed by the compiler.
>
> If *filename* is a relative filename (the normal case), it is interpreted as being relative to the file in which the declaration appears. Thus if the declaration appears in file '`/usr/cph/foo.scm`', then the compiler looks for a file called '`/usr/cph/`*filename*`.ext`'.
>
> Note: When the compiler finds top-level integrations, it collects them and outputs them into an auxiliary file with extension '`.ext`'. This '`.ext`' file is what the `integrate-external` declaration refers to.

Note that the most common use of this facility, in-line coding of procedure definitions, requires a somewhat complicated use of these declarations. Because this is so common, there is a special form, `define-integrable`, which is like `define` but performs the appropriate declarations. For example:

```
(define-integrable (foo-bar foo bar)
  (vector-ref (vector-ref foo bar) 3))
```

Here is how you do the same thing without this special form: there should be an `integrate-operator` declaration for the procedure's name, and (internal to the procedure's definition) an `integrate` declaration for each of the procedure's parameters, like this:

```
(declare (integrate-operator foo-bar))

(define foo-bar
  (lambda (foo bar)
    (declare (integrate foo bar))
    (vector-ref (vector-ref foo bar) 3)))
```

The reason for this complication is as follows: the `integrate-operator` declaration finds all the references to `foo-bar` and replaces them with the lambda expression from the definition. Then, the `integrate` declarations take effect because the combination in which the reference to `foo-bar` occurred supplies code that is substituted throughout the body of the procedure definition. For example:

```
(foo-bar (car baz) (cdr baz))
```

First use the `integrate-operator` declaration:

```
((lambda (foo bar)
   (declare (integrate foo bar))
   (vector-ref (vector-ref foo bar) 3))
 (car baz)
 (cdr baz))
```

Next use the internal `integrate` declaration:

```
((lambda (foo bar)
   (vector-ref (vector-ref (car baz) (cdr baz)) 3))
 (car baz)
 (cdr baz))
```

Next notice that the variables `foo` and `bar` are not used, and eliminate them:

```
((lambda ()
   (vector-ref (vector-ref (car baz) (cdr baz)) 3)))
```

Finally, remove the `((lambda () ...))` to produce

```
(vector-ref (vector-ref (car baz) (cdr baz)) 3)
```

Useful tip

To see the effect of integration declarations (and of macros) on a source file, pretty-print the '`.bin`' file like this (be prepared for a lot of output).

```
(sf "foo.scm")
(pp (fasload "foo.bin"))
```

4.2.3 Operator Replacement

The `replace-operator` declaration is provided to inform the compiler that certain operators may be replaced by other operators depending on the number of arguments. For example:

Declaration:

```
(declare (replace-operator (map (2 map-2) (3 map-3))))
```

Replacements:

```
(map f x y z) ↦ (map f x y z)
(map f x y)   ↦ (map-3 f x y)
(map f x)     ↦ (map-2 f x)
(map f)       ↦ (map f)
(map)         ↦ (map)
```

Presumably `map-2` and `map-3` are efficient versions of `map` that are written for exactly two and three arguments respectively. All the other cases are not expanded but are handled by the original, general `map` procedure, which is less efficient because it must handle a variable number of arguments.

replace-operator *name ...*   declaration+
> The syntax of this declaration is
>
> ```
> (replace-operator
>   (name
>     (nargs1 value1)
>     (nargs2 value2)
>     ...))
> ```
>
> where
>
> - *name* is a symbol.
> - *nargs1*, *nargs2* etc. are non-negative integers, or one of the following symbols: `any`, `else` or `otherwise`.
> - *value1*, *value2* etc. are simple expressions in one of these forms:
>
>   '*constant*   A constant.
>
>   *variable*   A variable.
>
>   (`primitive` *primitive-name* [*arity*])
>   > The primitive procedure named *primitive-name*. The optional element *arity*, a non-negative integer, specifies the number of arguments that the primitive accepts.
>
>   (`global` *var*)
>   > A global variable.
>
> The meanings of these fields are:
>
> - *name* is the name of the operator to be reduced. If is is not shadowed (for example, by a let) then it may be replaced according to the following rules.
> - If the operator has *nargsN* arguments then it is replaced with a call to *valueN* with the same arguments.

- If the number of arguments is not listed, and one of the *nargsN* is `any`, `else` or `otherwise`, then the operation is replaced with a call to the corresponding *valueN*. Only one of the *nargsN* may be of this form.

- If the number of arguments is not listed and none of the *nargsN* is `any`, `else` or `otherwise`, then the operation is not replaced.

4.2.4 Operator Reduction

The `reduce-operator` declaration is provided to inform the compiler that certain names are n-ary versions of binary operators. Here are some examples:

Declaration:

```
(declare (reduce-operator (cons* cons)))
```

Replacements:

```
(cons* x y z w) ↦ (cons x (cons y (cons z w))),
(cons* x y) ↦ (cons x y)
(cons* x) ↦ x
(cons*)  error   too few arguments
```

Declaration:

```
(declare (reduce-operator (list cons (null-value '() any))))
```

Replacements:

```
(list x y z w) ↦ (cons x (cons y (cons z (cons w '()))))
(list x y) ↦ (cons x (cons y '()))
(list x) ↦ (cons x '())
(list) ↦ '()
```

Declaration:

```
(declare (reduce-operator (- %- (null-value 0 single) (group left))))
```

Replacements:

```
(- x y z w) ↦ (%- (%- (%- x y) z) w)
(- x y) ↦ (%- x y)
(- x) ↦ (%- 0 x)
(-) ↦ 0
```

Declaration:

```
(declare (reduce-operator (+ %+ (null-value 0 none) (group right))))
```

Replacements:

```
(+ x y z w) ↦ (%+ x (%+ y (%+ z w)))
(+ x y) ↦ (%+ x y)
(+ x) ↦ x
(+) ↦ 0
```

Note: This declaration does not cause an appropriate definition of `%+` (in the last example) to appear in your code. It merely informs the compiler that certain optimizations can be performed on calls to `+` by replacing them with calls to `%+`. You should provide a definition of `%+` as well, although it is not required.

Declaration:

```
(declare (reduce-operator (apply (primitive cons)
                                 (group right)
                                 (wrapper (global apply) 1)))))
```

Replacements:

```
(apply f x y z w)
    ↦ ((access apply ()) f (cons x (cons y (cons z w))))
(apply f x y)
    ↦ ((access apply ()) f (cons x y))
(apply f x) ↦ (apply f x)
(apply f) ↦ (apply f)
(apply) ↦ (apply)
```

reduce-operator *name ...*   declaration+

The general format of the declaration is (brackets denote optional elements):

```
(reduce-operator
  (name
    binop
    [(group ordering)]
    [(null-value value null-option)]
    [(singleton unop)]
    [(wrapper wrap [n])]
    [(maximum m)]
  ))
```

where

- *n* and *m* are non-negative integers.

- *name* is a symbol.

- *binop*, *value*, *unop*, and *wrap* are simple expressions in one of these forms:

  '*constant*   A constant.

  variable   A variable.

  (primitive *primitive-name* [*arity*])
          The primitive procedure named *primitive-name*. The optional element *arity* specifies the number of arguments that the primitive accepts.

  (global *var*)
          A global variable.

- *null-option* is either `always`, `any`, `one`, `single`, `none`, or `empty`.

- *ordering* is either `left`, `right`, or `associative`.

The meaning of these fields is:

- *name* is the name of the n-ary operation to be reduced.

- *binop* is the binary operation into which the n-ary operation is to be reduced.

- The `group` option specifies whether *name* associates to the right or left.

- The `null-value` option specifies a value to use in the following cases:

```
none
empty       When no arguments are supplied to name, value is returned.

one
single      When a single argument is provided to name, value becomes the
            second argument to binop.

any
always      binop is used on the "last" argument, and value provides the remain-
            ing argument to binop.
```

In the above options, when *value* is supplied to *binop*, it is supplied on the left if grouping to the left, otherwise it is supplied on the right.

- The `singleton` option specifies a function, *unop*, to be invoked on the single argument given. This option supersedes the `null-value` option, which can only take the value `none`.

- The `wrapper` option specifies a function, *wrap*, to be invoked on the result of the outermost call to *binop* after the expansion. If *n* is provided it must be a non-negative integer indicating a number of arguments that are transferred verbatim from the original call to the wrapper. They are passed to the left of the reduction.

- The maximum option specifies that calls with more than *m* arguments should not be reduced.

4.3 Efficiency Tips

How you write your programs can have a large impact on how efficiently the compiled program runs. The most important thing to do, after choosing suitable data structures, is to put the following declaration near the beginning of the file.

```
(declare (usual-integrations))
```

Without this declaration the compiler cannot recognize any of the common operators and compile them efficiently.

The `usual-integrations` declaration is usually sufficient to get good quality compiled code.

If you really need to squeeze more performance out of your code then we hope that you find the following grab-bag of tips, hints and explanations useful.

4.3.1 Coding style

Scheme is a rich language, in which there are usually several ways to say the same thing. A *coding style* is a set of rules that a programmer uses for choosing an expressive form to use in a given situation. Usually these rules are aesthetic, but sometimes there are efficiency issues involved; this section describes a few choices that have non-obvious efficiency consequences.

Better predicates

Consider the following implementation of `map` as might be found in any introductory book on Scheme:

```
(define (map f lst)
  (if (null? lst)
      '()
      (cons (f (car lst)) (map f (cdr lst)))))
```

The problem with this definition is that at the points where `car` and `cdr` are called we still do not know that *lst* is a pair. The compiler must insert a type check, or if type checks are disabled, the program might give wrong results. Since one of the fundamental properties of `map` is that it transforms lists, we should make the relationship between the input pairs and the result pairs more apparent in the code:

```
(define (map f lst)
  (cond ((pair? lst)
         (cons (f (car lst)) (map f (cdr lst))))
        ((null? lst)
         '())
        (else
         (error "Not a proper list:"  lst))))
```

Note also that the `pair?` case comes first because we expect that `map` will be called on lists which have, on average, length greater that one.

Internal procedures

Calls to internal procedures are faster than calls to global procedures. There are two things that make internal procedures faster: First, the procedure call is compiled to a direct jump to a known location, which is more efficient that jumping 'via' a global binding. Second, there is a knock-on effect: since the compiler can see the internal procedure, the compiler can analyze it and possibly produce better code for other expressions in the body of the loop too:

```
(define (map f original-lst)
  (let walk ((lst original-lst))
    (cond ((pair? lst)
           (cons (f (car lst)) (walk (cdr lst))))
          ((null? lst)
           '())
          (else
           (error "Not a proper list:"  original-lst)))))
```

Internal defines

Internal definitions are a useful tool for structuring larger procedures. However, certain internal definitions can thwart compiler optimizations. Consider the following two procedures, where `compute-100` is some unknown procedure that we just know returns '100'.

```
(define (f1)
  (define v 100)
  (lambda () v))

(define (f2)
  (define v (compute-100))
  (lambda () v))
```

The procedure returned by `f1` will always give the same result and the compiler can prove this. The procedure returned by `f2` may return different results, even if `f2` is only called once. Because of this, the compiler has to allocate a memory cell to `v`. How can the procedure return different results?

The fundamental reason is that the continuation may escape during the evaluation of `(compute-100)`, allowing the rest of the body of `f2` to be executed *again*:

```
(define keep)

(define (compute-100)
  (call-with-current-continuation
   (lambda (k)
     (set! keep k)
     100)))

(define p (f2))

(p)                    ⇒ 100
(keep -999)            ⇒ p        re-define v and p
(p)                    ⇒ -999
```

To avoid the inefficiency introduced to handle the general case, the compiler must prove that the continuation cannot possibly escape. The compiler knows that lambda expressions and constants do not let their continuations escape, so order the internal definitions so that definitions of the following forms come first:

```
(define x 'something)
(define x (lambda (...) ...))
(define (f u v) ...)
```

Note: The IEEE Scheme standard permits *only* lambda expressions and constants as the value of internal defines. Furthermore, all internal definitions must appear before any other expressions in the body. Following the standard simultaneously assures portability and avoids the implementation inefficiencies described in this section.

4.3.2 Global variables

Compiled code usually accesses variables in top-level first-class environments via *variable caches*. Each compiled procedure has a set of variable caches for the global variables that it uses. There are three kinds of variable cache - read caches for getting the value of a variable (referencing the variable), write caches for changing the value, and execute caches for calling the procedure assigned to that variable.

Sometimes the variable caches contain special objects, called reference traps, that indicate that the operation cannot proceed normally and must either be completed by the system (in order to keep the caches coherent) or must signal an error. For example, the assignment

```
(set! newline my-better-newline)
```

will cause the system to go to each compiled procedure that calls `newline` and update its execute cache to call the new procedure. Obviously you want to avoid updating hundreds of execute caches in a critical loop. Using `fluid-let` to temporarily redefine a procedure has the same inefficiency (but twice!).

To behave correctly in all situations, each variable reference or assignment must check for the reference traps.

Sometimes you can prove that the variable (a) will always be bound, (b) will never be unassigned, and (c) there will never be any compiled calls to that variable. The compiler can't prove this because it assumes that other independently compiled files might be loaded that invalidate these assumptions. If you know that these conditions hold, the following declarations can speed up and reduce the size of a program that uses global variables.

ignore-reference-traps *variables*   declaration+
> This declaration tells the compiler that it need not check for reference-trap objects when referring to the given *variables*. If any of the *variables* is unbound or unassigned then a variable reference will yield a reference-trap object rather than signaling an error. This declaration is relatively safe: the worst that can happen is that a reference-trap object finds its way into a data structure (e.g. a list) or into interpreted code, in which case it will probably cause some 'unrelated' variable to mysteriously become unbound or unassigned.

ignore-assignment-traps *variables*   declaration+
> This declaration tells the compiler that it need not check for reference-trap objects when assigning to the given *variables*. An assignment to a variable that ignores assignment traps can cause a great deal of trouble. If there is a compiled procedure call anywhere in the system to this variable, the execute caches will not be updated, causing an inconsistency between the value used for the procedure call and the value seen by reading the variable. This mischief is compounded by the fact that the assignment can cause other assignments that were compiled with checks to behave this way too.

The *variables* are specified with expressions from the following set language:

set *name ...*   variable-specification
> All of the explicitly listed names.

all   variable-specification
none   variable-specification
free   variable-specification
bound   variable-specification
assigned   variable-specification
> These expressions name sets of variables. `all` is the set of all variables, `none` is the empty set, `free` is all of the variables bound outside the current block, `bound` is all of the variables bound in the current block and `assigned` is all of the variables for which there exists an assignment (i.e. `set!`).

union *set1 set2*   variable-specification
intersection *set1 set2*   variable-specification
difference *set1 set2*   variable-specification
> For example, to ignore reference traps on all the variables except *x*, *y* and any variable that is assigned to

```
(declare (ignore-reference-traps
          (difference all (union assigned (set x y)))))
```

4.3.3 Fixnum arithmetic

The usual arithmetic operations like + and < are called generic arithmetic operations because they work for all (appropriate) kinds of number.

A *fixnum* is an exact integer that is small enough to fit in a machine word. In MIT Scheme, fixnums are 26 bits on 32-bit machines, and 56 bits on 64-bit machines; it is reasonable to assume that fixnums are at least 24 bits. Fixnums are signed; they are encoded using 2's complement.

All exact integers that are small enough to be encoded as fixnums are always encoded as fixnums — in other words, any exact integer that is not a fixnum is too big to be encoded as such. For this reason, small constants such as 0 or 1 are guaranteed to be fixnums. In addition, the lengths of and valid indexes into strings and vectors are also always fixnums.

If you know that a value is always a small fixnum, you can substitute the equivalent fixnum operation for the generic operation. However, care should be exercised: if used improperly, these operations can return incorrect answers, or even malformed objects that confuse the garbage collector. For a listing of all fixnum operations, see section "Fixnum Operations" in *MIT Scheme Reference Manual*.

A fruitful area for inserting fixnum operations is in the index operations in tight loops.

4.3.4 Flonum arithmetic

Getting efficient flonum arithmetic is much more complicated and harder than getting efficient fixnum arithmetic.

Flonum consing

One of the main disadvantages of generic arithmetic is that not all kinds of number fit in a machine register. Flonums have to be *boxed* because a 64-bit IEEE floating-point number (the representation that MIT Scheme uses) does not fit in a regular machine word. This is true even on 64-bit architectures because some extra bits are needed to distinguish floating-point numbers from other objects like pairs and strings. Values are boxed by storing them in a small record in the heap. Every floating-point value that you see at the REPL is boxed. Floating-point values are unboxed only for short periods of time when they are in the machine's floating-point unit and actual floating-point operations are being performed.

Numerical calculations that happen to be using floating-point numbers cause many temporary floating-point numbers to be allocated. It is not uncommon for numerical programs to spend over half of their time creating and garbage collecting the boxed flonums.

Consider the following procedure for computing the distance of a point (x,y) from the origin.

```
(define (distance x y)
  (sqrt (+ (* x x) (* y y))))
```

The call (distance 0.3 0.4) returns a new, boxed flonum, 0.5. The calculation also generates three intermediate boxed flonums. This next version works only for flonum inputs, generates only one boxed flonum (the result) and runs eight times faster:

```
(define (flo:distance x y)
  (flo:sqrt (flo:+ (flo:* x x) (flo:* y y))))
```

Note that `flo:` operations are usually effective only within a single arithmetic expression. If the expression contains conditionals or calls to procedures then the values tend to get boxed anyway.

Flonum vectors

Flonum vectors are vectors that contain only floating-point values, in much the same way as a string is a 'vector' containing only character values.

Flonum vectors have the advantages of compact storage (about half that of a conventional vector of flonums) and judicious use of flonum vectors can decrease flonum consing.

The disadvantages are that flonum vectors are incompatible with ordinary vectors, and if not used carefully, can increase flonum consing. Flonum vectors are a pain to use because they require you to make a decision about the representation and stick with it, and it might not be easy to ascertain whether the advantages in one part of the program outweigh the disadvantages in another.

The flonum vector operations are:

flo:vector-cons *n*   procedure+
> Create a flonum vector of length *n*. The contents of the vector are arbitrary and might not be valid floating-point numbers. The contents should not be used until initialized.

flo:vector-ref *flonum-vector index*   procedure+
flo:vector-set! *flonum-vector index value*   procedure+
flo:vector-length *flonum-vector*   procedure+
> These operations are analogous to the ordinary vector operations.

Examples

The following operation causes no flonum consing because the flonum is loaded directly from the flonum vector into a floating-point machine register, added, and stored again. There is no need for a temporary boxed flonum.

```
(flo:vector-set! v 0 (flo:+ (flo:vector-ref v 0) 1.2))
```

In this next example, every time **g** is called, a new boxed flonum has to be created so that a valid Scheme object can be returned. If **g** is called more often than the elements of *v* are changed then an ordinary vector might be more efficient.

```
(define (g i)
  (flo:vector-ref v i))
```

Common pitfalls

Pitfall 1: Make sure that your literals are floating-point constants:

```
(define (f1 a) (flo:+ a 1))
(define (f2 a) (flo:+ a 1.))
```

f1 will most likely cause a hardware error, and certainly give the wrong answer. f2 is correct.

Pitfall 2: It is tempting to insert calls to exact->inexact to coerce values into flonums. This does not always work because complex numbers may be exact or inexact too. Also, the current implementation of exact->inexact is slow.

Pitfall 3: A great deal of care has to be taken with the standard math procedures. For example, when called with a flonum, both sqrt and asin can return a complex number (e.g with argument -1.5).

5 Debugging

Parts of this chapter are adapted from *Don't Panic: A 6.001 User's Guide to the Chipmunk System*, by Arthur A. Gleckler.

Even computer software that has been carefully planned and well written may not always work correctly. Mysterious creatures called *bugs* may creep in and wreak havoc, leaving the programmer to clean up the mess. Some have theorized that a program fails only because its author made a mistake, but experienced computer programmers know that bugs are always to blame. This is why the task of fixing broken computer software is called *debugging*.

It is impossible to prove the correctness of any non-trivial program; hence the Cynic's First Law of Debugging:

> Programs don't become more reliable as they are debugged; the bugs just get harder to find.

Scheme is equipped with a variety of special software for finding and removing bugs. The debugging tools include facilities for tracing a program's use of specified procedures, for examining Scheme environments, and for setting *breakpoints*, places where the program will pause for inspection.

Many bugs are detected when programs try to do something that is impossible, like adding a number to a symbol, or using a variable that does not exist; this type of mistake is called an *error*. Whenever an error occurs, Scheme prints an error message and starts a new REPL. For example, using a nonexistent variable `foo` will cause Scheme to respond

```
1 ]=> foo

;Unbound variable: foo
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of foo.
; (RESTART 2) => Define foo to a given value.
; (RESTART 1) => Return to read-eval-print level 1.

2 error>
```

Sometimes, a bug will never cause an error, but will still cause the program to operate incorrectly. For instance,

```
(prime? 7)   ⇒   #f
```

In this situation, Scheme does not know that the program is misbehaving. The programmer must notice the problem and, if necessary, start the debugging tools manually.

There are several approaches to finding bugs in a Scheme program:

- Inspect the original Scheme program.
- Use the debugging tools to follow your program's progress.
- Edit the program to insert checks and breakpoints.

Only experience can teach how to debug programs, so be sure to experiment with all these approaches while doing your own debugging. Planning ahead is the best way to ward off bugs, but when bugs do appear, be prepared to attack them with all the tools available.

5.1 Subproblems and Reductions

Understanding the concepts of *reduction* and *subproblem* is essential to good use of the debugging tools. The Scheme interpreter evaluates an expression by *reducing* it to a simpler expression. In general, Scheme's evaluation rules designate that evaluation proceeds from one expression to the next by either starting to work on a *subexpression* of the given expression, or by reducing the entire expression to a new (simpler, or reduced) form. Thus, a history of the successive forms processed during the evaluation of an expression will show a sequence of subproblems, where each subproblem may consist of a sequence of reductions.

For example, both `(+ 5 6)` and `(+ 7 9)` are subproblems of the following combination:

```
(* (+ 5 6) (+ 7 9))
```

If `(prime? n)` is true, then `(cons 'prime n)` is a reduction for the following expression:

```
(if (prime? n)
    (cons 'prime n)
    (cons 'not-prime n))
```

This is because the entire subproblem of the `if` expression can be reduced to the problem `(cons 'prime n)`, once we know that `(prime? n)` is true; the `(cons 'not-prime n)` can be ignored, because it will never be needed. On the other hand, if `(prime? n)` were false, then `(cons 'not-prime n)` would be the reduction for the `if` expression.

The *subproblem level* is a number representing how far back in the history of the current computation a particular evaluation is. Consider `factorial`:

```
(define (factorial n)
  (if (< n 2)
      1
      (* n (factorial (- n 1)))))
```

If we stop `factorial` in the middle of evaluating `(- n 1)`, the `(- n 1)` is at subproblem level 0. Following the history of the computation "upwards," `(factorial (- n 1))` is at subproblem level 1, and `(* n (factorial (- n 1)))` is at subproblem level 2. These expressions all have *reduction number* 0. Continuing upwards, the `if` expression has reduction number 1.

Moving backwards in the history of a computation, subproblem levels and reduction numbers increase, starting from zero at the expression currently being evaluated. Reduction numbers increase until the next subproblem, where they start over at zero. The best way to get a feel for subproblem levels and reduction numbers is to experiment with the debugging tools, especially `debug`.

5.2 The Command-Line Debugger

There are two debuggers available with MIT Scheme. One of them runs under Edwin, and is described in that section of this document (see Section 7.6 [Edwin Debugger], page 62). The other is command-line oriented, does not require Edwin, and is described here.

The *command-line debugger*, called `debug`, is the tool you should use when Scheme signals an error and you want to find out what caused the error. When Scheme signals an error, it records all the information necessary to continue running the Scheme program that

caused the error; the debugger provides you with the means to inspect this information. For this reason, the debugger is sometimes called a *continuation browser.*

Here is the transcript of a typical Scheme session, showing a user evaluating the expression `(fib 10)`, Scheme responding with an unbound variable error for the variable `fob`, and the user starting the debugger:

```
1 ]=> (fib 10)

;Unbound variable: fob
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of fob.
; (RESTART 2) => Define fob to a given value.
; (RESTART 1) => Return to read-eval-print level 1.

2 error> (debug)

There are 6 subproblems on the stack.

Subproblem level: 0 (this is the lowest subproblem level)
Expression (from stack):
    fob
Environment created by the procedure: FIB
 applied to: (10)
The execution history for this subproblem contains 1 reduction.
You are now in the debugger.  Type q to quit, ? for commands.

3 debug>
```

This tells us that the error occurred while trying to evaluate the expression '`fob`' while running '`(fib 10)`'. It also tells us this is subproblem level 0, the first of 6 subproblems that are available for us to examine. The expression shown is marked '`(from stack)`', which tells us that this expression was reconstructed from the interpreter's internal data structures. Another source of information is the *execution history*, which keeps a record of expressions evaluated by the interpreter. The debugger informs us that the execution history has recorded some information for this subproblem, specifically a description of one reduction.

What follows is a description of the commands available in the debugger. To understand how the debugger works, you need to understand that the debugger has an implicit state that is examined and modified by commands. The state consists of three pieces of information: a *subproblem*, a *reduction*, and an *environment frame*. Each of these parts of the implicit state is said to be *selected*; thus one refers to the *selected subproblem*, and so forth. The debugger provides commands that examine the selected state, and allow you to select different states.

Here are the debugger commands. Each of these commands consists of a single letter, which is to be typed by itself at the debugger prompt. It is not necessary to type RET after these commands.

Traversing subproblems

> The debugger has several commands for traversing the structure of the continuation. It is useful to think of the continuation as a two-dimensional structure:

a backbone consisting of subproblems, and associated ribs consisting of reductions. The bottom of the backbone is the most recent point in time; that is where the debugger is positioned when it starts. Each subproblem is numbered, with `0` representing the most recent time point, and ascending integers numbering older time points. The `u` command moves up to older points in time, and the `d` command moves down to newer points in time. The `g` command allows you to select a subproblem by number, and the `h` command will show you a brief summary of all of the subproblems.

Traversing reductions

If the subproblem description says that 'The execution history for this subproblem contains N reductions', then there is a "rib" of reductions for this subproblem. You can see a summary of the reductions for this subproblem using the `r` command. You can move to the next reduction using the `b` command; this moves you to the next older reduction. The `f` command moves in the opposite direction, to newer reductions. If you are at the oldest reduction for a given subproblem and use the `b` command, you will move to the next older subproblem. Likewise, if you are at the newest reduction and use `f`, you'll move to the next newer subproblem.

Examining subproblems and reductions

The following commands will show you additional information about the currently selected subproblem or reduction. The `t` command will reprint the standard description (in case it has scrolled off the screen). The `l` command will pretty-print (using `pp`) the subproblem's expression.

Traversing environments

Nearly all subproblems and all reductions have associated environments. Selecting a subproblem or reduction also selects the associated environment. However, environments are structured as a sequence of frames, where each frame corresponds to a block of environment variables, as bound by `lambda` or `let`. These frames collectively represent the block structure of a given environment.

Once an environment frame is selected by the debugger, it is possible to select the parent frame of that frame (in other words, the enclosing block) using the `p` command. You can subsequently return to the original child frame using the `s` command. The `s` command works because the `p` command keeps track of the frames that you step through as you move up the environment hierarchy; the `s` command just retraces the path of saved frames. Note that selecting a frame using `p` or `s` will print the bindings of the newly selected frame.

Examining environments

The following commands allow you to examine the contents of the selected frame. The `c` command prints the bindings of the current frame. The `a` command prints the bindings of the current frame and each of its ancestor frames. The `e` command enters a read-eval-print loop in the selected environment frame; expressions typed at that REPL will be evaluated in the selected environment. To exit the REPL and return to the debugger, evaluate `(abort->previous)` or use `restart`. The `v` command prompts for a single expression and evaluates it in the selected environment. The `w` command invokes the environment in-

spector (`where`); quitting the environment inspector returns to the debugger. Finally, the `o` command pretty-prints the procedure that was called to create the selected environment frame.

Continuing the computation

There are three commands that can be used to restart the computation that you are examining. The first is the `k` command, which shows the currently active restarts, prompts you to select one, and passes control to the it. It is very similar to evaluating '(`restart`)'.

The other two commands allow you to invoke internal continuations. This should not be done lightly; invoking an internal continuation can violate assumptions that the programmer made and cause unexpected results. Each of these commands works in the same way: it prompts you for an expression, which is evaluated in the selected environment to produce a value. The appropriate internal continuation is then invoked with that value as its sole argument. The two commands differ only in which internal continuation is to be invoked.

The `j` command invokes the continuation associated with the selected subproblem. What this means is as follows: when the description of a subproblem is printed, it consists of two parts, and "expression" and a "subproblem being executed". The latter is usually marked in the former by the specific character sequence '`###`'. The internal continuation of the subproblem is the code that is waiting for the "subproblem being executed" to return a value. So, in effect, you are telling the program what the "subproblem being executed" will evaluate to, and bypassing further execution of that code.

The `z` command is slightly different. It instead invokes the continuation that is waiting for the outer "expression" to finish. In other words, it is the same as invoking the `j` command in the next frame up. So you can think of this as an abbreviation for the `u` command followed by the `j` command.

Wizard commands

The `m`, `x`, and `y` commands are for Scheme wizards. They are used to debug the MIT Scheme implementation. If you want to find out what they do, read the source code.

Miscellaneous commands

The `i` command will reprint the error message for the error that was in effect immediately before the debugger started. The `q` command quits the debugger, returning to the caller. And the `?` command prints a brief summary of the debugger's commands.

5.3 Debugging Aids

This section describes additional commands that are useful for debugging.

bkpt *datum argument . . .*   procedure+

Sets a breakpoint. When the breakpoint is encountered, *datum* and the *argument*s are typed (just as for `error`) and a read-eval-print loop is entered. The environment of the read-eval-print loop is derived by examining the continuation of the call to

bkpt; if the call appears in a non-tail-recursive position, the environment will be that of the call site. To exit from the breakpoint and proceed with the interrupted process, call the procedure `continue`. Sample usage:

```
1 ]=> (begin (write-line 'foo)
             (bkpt 'test-2 'test-3)
             (write-line 'bar)
             'done)

foo
 test-2 test-3
;To continue, call RESTART with an option number:
; (RESTART 2) => Return from BKPT.
; (RESTART 1) => Return to read-eval-print level 1.

2 bkpt> (+ 3 3)

;Value: 6

2 bkpt> (continue)

bar
;Value: done
```

pp *object* [*output-port* [*as-code?*]]   procedure+

The `pp` procedure is described in section "Output Procedures" in *MIT Scheme Reference Manual*. However, since this is a very useful debugging tool, we also mention it here. `pp` provides two very useful functions:

1. `pp` will print the source code of a given procedure. Often, when debugging, you will have a procedure object but will not know exactly what procedure it is. Printing the procedure using `pp` will show you the source code, which greatly aids identification.

2. `pp` will print the fields of a record structure. If you have a compound object pointer, print it using `pp` to see the component fields, like this:

```
(pp (->pathname "~"))
⊣ #[pathname 14 "/usr/home/cph"]
⊣ (host #[host 15])
⊣ (device unspecific)
⊣ (directory (absolute "usr" "home"))
⊣ (name "cph")
⊣ (type ())
⊣ (version unspecific)
```

When combined with use of the `#@` syntax, `pp` provides the functionality of a simple object inspector. For example, let's look at the fields of the host object from the above example:

```
(pp #@15)
⊣ #[host 15]
⊣ (type-index 0)
⊣ (name ())
```

pa *procedure*   procedure+

> `pa` prints the arguments of *procedure*. This can be used to remind yourself, for
> example, of the correct order of the arguments to a procedure.
>
>   for-all?
>   ⇒ #[compiled-procedure 40 ("boole" #x6) #xC #x20ECB0]
>
>   (pa for-all?)
>   ⊣ (items predicate)
>
>   (pp for-all?)
>   ⊣(named-lambda (for-all? items predicate)
>   ⊣   (let loop ((items items))
>   ⊣   (or (null? items)
>   ⊣   (and (predicate (car items))
>   ⊣   (loop (cdr items))))))

where [*obj*]   procedure+

> The procedure `where` enters the environment examination system. This allows en-
> vironments and variable bindings to be examined and modified. `where` accepts one-
> letter commands. The commands can be found by typing *?* to the 'where>' prompt.
> The optional argument, *obj*, is an object with an associated environment: an environ-
> ment, a procedure, or a promise. If *obj* is omitted, the environment examined is the
> read-eval-print environment from which `where` was called (or an error or breakpoint
> environment if called from the debugger). If a procedure is supplied, `where` lets the
> user examine the closing environment of the procedure. This is useful for debugging
> procedure arguments and values.

apropos *string* [*environment* [*search-parents?*]]   procedure+

> Search an environment for bound names containing *string* and print out the matching
> bound names. If *environment* is specified, it must be an environment or package name,
> and it defaults to the current REPL environment. The flag *search-parents?* specifies
> whether the environment's parents should be included in the search. The default is
> `#f` if *environment* is specified, and `#t` if *environment* is not specified.
>
>   (apropos "search")
>   ⊣ #[package 47 (user)]
>   ⊣ #[package 48 ()]
>   ⊣ list-search-negative
>   ⊣ list-search-positive
>   ⊣ nt-fs-flag/case-sensitive-search
>   ⊣ re-string-search-backward
>   ⊣ re-string-search-forward
>   ⊣ re-substring-search-backward
>   ⊣ re-substring-search-forward
>   ⊣ search-ordered-subvector
>   ⊣ search-ordered-vector
>   ⊣ search-protection-list
>   ⊣ string-search-all
>   ⊣ string-search-backward

```
⊣ string-search-forward
⊣ substring-search-all
⊣ substring-search-backward
⊣ substring-search-forward
⊣ vector-binary-search
```

5.4 Advising Procedures

Giving advice to procedures is a powerful debugging technique. `trace` and `break` are useful examples of advice-giving procedures. Note that the advice system only works for interpreted procedures.

trace-entry *procedure*   procedure+

Causes an informative message to be printed whenever *procedure* is entered. The message is of the form

```
[Entering #[compound-procedure 1 foo]
    Args: val1
          val2
          ...]
```

where *val1*, *val2* etc. are the evaluated arguments supplied to the procedure.

```
(trace-entry fib)
(fib 3)
⊣ [Entering #[compound-procedure 19 fib]
⊣     Args: 3]
⊣ [Entering #[compound-procedure 19 fib]
⊣     Args: 1]
⊣ [Entering #[compound-procedure 19 fib]
⊣     Args: 2]
⇒ 3
```

trace-exit *procedure*   procedure+

Causes an informative message to be printed when *procedure* terminates. The message contains the procedure, its argument values, and the value returned by the procedure.

```
(trace-exit fib)
(fib 3)
⊣ [1
⊣        <== #[compound-procedure 19 fib]
⊣     Args: 1]
⊣ [2
⊣        <== #[compound-procedure 19 fib]
⊣     Args: 2]
⊣ [3
⊣        <== #[compound-procedure 19 fib]
⊣     Args: 3]
⇒ 3
```

trace-both *procedure*   procedure+
trace *procedure*   procedure+
>   Equivalent to calling both `trace-entry` and `trace-exit` on *procedure*. `trace` is the
>   same as `trace-both`.
>
> ```
> (trace-both fib)
> (fib 3)
> ⊣ [Entering #[compound-procedure 19 fib]
> ⊣     Args: 3]
> ⊣ [Entering #[compound-procedure 19 fib]
> ⊣     Args: 1]
> ⊣ [1
> ⊣       <== #[compound-procedure 19 fib]
> ⊣     Args: 1]
> ⊣ [Entering #[compound-procedure 19 fib]
> ⊣     Args: 2]
> ⊣ [2
> ⊣       <== #[compound-procedure 19 fib]
> ⊣     Args: 2]
> ⊣ [3
> ⊣       <== #[compound-procedure 19 fib]
> ⊣     Args: 3]
> ⇒ 3
> ```

untrace-entry [*procedure*]   procedure+
>   Stops tracing the entry of *procedure*. If *procedure* is not given, the default is to stop
>   tracing the entry of all entry-traced procedures.

untrace-exit [*procedure*]   procedure+
>   Stops tracing the exit of *procedure*. If *procedure* is not given, the default is all
>   exit-traced procedures.

untrace [*procedure*]   procedure+
>   Stops tracing both the entry to and the exit from *procedure*. If *procedure* is not
>   given, the default is all traced procedures.

break-entry *procedure*   procedure+
>   Like `trace-entry` with the additional effect that a breakpoint is entered when *pro-
>   cedure* is invoked. Both *procedure* and its arguments can be accessed by calling the
>   procedures `*proc*` and `*args*`, respectively. Use `restart` or `continue` to continue
>   from a breakpoint.

break-exit *procedure*   procedure+
>   Like `trace-exit`, except that a breakpoint is entered just prior to leaving *procedure*.
>   *Procedure*, its arguments, and the result can be accessed by calling the procedures
>   `*proc*`, `*args*`, and `*result*`, respectively. Use `restart` or `continue` to continue
>   from a breakpoint.

break-both *procedure*   procedure+
break *procedure*   procedure+
    Sets a breakpoint at the beginning and end of *procedure*. This is `break-entry` and
    `break-exit` combined.

unbreak [*procedure*]   procedure+
    Discontinues the entering of a breakpoint on the entry to and exit from *procedure*. If
    procedure is not given, the default is all breakpointed procedures.

unbreak-entry [*procedure*]   procedure+
    Discontinues the entering of a breakpoint on the entry to *procedure*. If *procedure* is
    not given, the default is all entry-breakpointed procedures.

unbreak-exit [*procedure*]   procedure+
    Discontinues the entering of a breakpoint on the exit from *procedure*. If *procedure* is
    not given, the default is all exit-breakpointed procedures.

The following three procedures are valid only within the dynamic extent of a breakpoint.
In other words, don't call them unless you are stopped inside a breakpoint.

proc   procedure+
    Returns the procedure in which the breakpoint has stopped.

args   procedure+
    Returns the arguments to the procedure in which the breakpoint has stopped. The
    arguments are returned as a newly allocated list.

result   procedure+
    Returns the result yielded by the procedure in which the breakpoint has stopped.
    This is valid only when in an exit breakpoint.

The following procedures install *advice* procedures that are called when the advised
procedure is entered or exited. An entry-advice procedure must accept three arguments:
the advised procedure, a list of the advised procedure's arguments, and the advised proce-
dure's application environment (that is, the environment in which the procedure's formal
parameters are bound). An exit-advice procedure must accept four arguments: the advised
procedure, a list of the advised procedure's arguments, the result yielded by the advised
procedure, and the advised procedure's application environment.

Note that the trace and breakpoint procedures described above are all implemented by
means of the more general advice procedures, so removing advice from an advised procedure
will also remove traces and breakpoints.

advise-entry *procedure advice*   procedure+
    Advice must be an entry-advice procedure. *Advice* is attached to *procedure*, so that
    whenever *procedure* is entered, *advice* is called.

advise-exit *procedure advice*   procedure+
> *Advice* must be an exit-advice procedure. *Advice* is attached to *procedure*, so that whenever *procedure* returns, *advice* is called.

advice *procedure*   procedure+
> Returns the advice procedures, if any, that are attached to *procedure*. This is returned as a list of two lists: the first list is all of the entry-advice procedures attached to *procedure*, and the second is all of the exit-advice procedures.

unadvise-entry [*procedure*]   procedure+
> Removes all entry-advice procedures from *procedure*. If *procedure* is not given, the default is all entry-advised procedures.

unadvise-exit [*procedure*]   procedure+
> Removes exit-advice procedures from *procedure*. If *procedure* is not given, the default is all exit-advised procedures.

unadvise [*procedure*]   procedure+
> Removes all advice procedures from *procedure*. This is a combination of `unadvise-entry` and `unadvise-exit`. If *procedure* is not given, the default is all advised procedures.

6 GNU Emacs Interface

There is an interface library, called '`xscheme`', distributed with MIT Scheme and GNU Emacs, which facilitates running Scheme as a subprocess of Emacs. If you wish to use this interface, please install the version of '`xscheme.el`' that comes with MIT Scheme, as it is guaranteed to be correct for your version of Scheme.

This interface works under unix only, because it requires unix signals for its operation. Porting it to either OS/2 or Windows would require reimplementing the interface to eliminate the use of signals. We have no plans to do this.

To invoke Scheme from Emacs, load the '`xscheme`' library, then use *M-x run-scheme*. You may give `run-scheme` a prefix argument, in which case it will allow you to edit the command line that is used to invoke Scheme. *Do not* remove the `-emacs` option!

Note carefully: In Emacs 19 and later, the `run-scheme` command exists, but is different from the one described here! In order to get this interface, you must load the '`xscheme`' library before executing `run-scheme`.

Scheme will be started up as a subprocess in a buffer called '`*scheme*`'. This buffer will be in `scheme-interaction-mode` and all output from the Scheme process will go there. The mode line for the '`*scheme*`' buffer will have this form:

```
--**-*scheme*: 1 [Evaluator]              (Scheme Interaction: input)------
```
The first field, showing '`1`' in this example, is the level number.

The second field, showing '`[Evaluator]`' in this example, describes the type of REPL that is running. Other values include:

```
[Debugger]
[Where]
```
The *mode* after '`Scheme Interaction`' is one of:

'`input`'   Scheme is waiting for input.

'`run`'   Scheme is running an evaluation.

'`gc`'   Scheme is garbage collecting.

When '`xscheme`' is loaded, `scheme-mode` is extended to include commands for evaluating expressions (do *C-h m* in any `scheme-mode` buffer for the most up-to-date information):

M-o   Evaluates the current buffer (`xscheme-send-buffer`).

M-z   Evaluates the current definition (`xscheme-send-definition`). This is also bound to *C-M-x*.

C-M-z   Evaluates the current region (`xscheme-send-region`).

C-x C-e   Evaluates the expression to the left of point (`xscheme-send-previous-expression`). This is also bound to *M-*RET.

C-c C-s   Selects the '`*scheme*`' buffer and places you at its end (`xscheme-select-process-buffer`).

C-c C-y   Yanks the most recently evaluated expression, placing it at point (`xscheme-yank-previous-send`). This works only in the '`*scheme*`' buffer.

The following commands provide interrupt capability:

`C-c C-c`   Like typing `C-g` when running Scheme without Emacs. (`xscheme-send-control-g-interrupt`)

`C-c C-x`   Like typing `C-c C-x` when running Scheme without Emacs. (`xscheme-send-control-x-interrupt`)

`C-c C-u`   Like typing `C-c C-u` when running Scheme without Emacs. (`xscheme-send-control-u-interrupt`)

`C-c C-b`   Like typing `C-c C-b` when running Scheme without Emacs. (`xscheme-send-breakpoint-interrupt`)

`C-c C-p`   Like evaluating (`continue`). (`xscheme-send-proceed`)

7 Edwin

This chapter describes how to start Edwin, the MIT Scheme text editor. Edwin is very
similar to GNU Emacs — you should refer to the GNU Emacs manual for information
about Edwin's commands and key bindings — except that Edwin's extension language is
MIT Scheme, while GNU Emacs extensions are written in Emacs Lisp. This manual does
not discuss customization of Edwin.

7.1 Starting Edwin

To use Edwin, start Scheme with the following command-line options:

```
scheme -edwin -edit
```

Alternatively, you can load Edwin by giving the `-edwin` command-line option and then
calling the procedure `edit`:

edit   procedure+
edwin   procedure+
> Enter the Edwin text editor. If entering for the first time, the editor is initialized
> (by calling `create-editor` with no arguments). Otherwise, the previously-initialized
> editor is reentered.
>
> The procedure `edwin` is an alias for `edit`.

inhibit-editor-init-file?   variable+
> When Edwin is first initialized, it loads your init file (called '`~/.edwin`' under unix,
> '`edwin.ini`' on PCs) if you have one. If the Scheme variable `inhibit-editor-init-`
> `file?` is true, however, your init file will not be loaded even if it exists. By default,
> this variable is false.
>
> Note that you can set this variable in your Scheme init file (see Section 2.2 [Customiz-
> ing Scheme], page 9).

create-editor *arg* ...   procedure+
> Initializes Edwin, or reinitializes it if already initialized. `create-editor` is normally
> invoked automatically by `edit`.
>
> If no *arg*s are given, the value of `create-editor-args` is used instead. In other
> words, the following are equivalent:
>
> ```
> (create-editor)
> (apply create-editor create-editor-args)
> ```
>
> On the other hand, if *arg*s are given, they are used to update `create-editor-args`,
> making the following equivalent:
>
> ```
> (apply create-editor args)
> (begin (set! create-editor-args args) (create-editor))
> ```

create-editor-args   variable+
> This variable controls the initialization of Edwin. The following values are defined:

(#f)   This is the default. Creates a window of some default size, and uses that
                window as Edwin's main window. Under unix, if X11 is not available
                or if the DISPLAY environment variable is undefined, Edwin will run on
                Scheme's console.

(x)   Unix only. Creates an X window and uses it as Edwin's main window.
                This requires the DISPLAY environment variable to have been set to the
                appropriate value before Scheme was started.

(x *geometry*)
                Unix only. Like (x) except that *geometry* specifies the window's geome-
                try in the usual way. *Geometry* must be a character string whose contents
                is an X geometry specification.

(console)
                Unix only. Causes Edwin to run on Scheme's console, or in unix termi-
                nology, the standard input and output. If the console is not a terminal
                device, or is not powerful enough to run Edwin, an error will be signalled
                at initialization time.

(pm)   OS/2 only. Creates a Presentation Manager window and uses it as Ed-
                win's main window.

(win32)   Windows only. Creates a window and uses it as Edwin's main window.

7.2 Leaving Edwin

Once Edwin has been entered, it can be exited in the following ways:

C-x z   Stop Edwin and return to Scheme (suspend-edwin). The call to the procedure
                edit that entered Edwin returns normally. A subsequent call to edit will
                resume Edwin where it was stopped.

C-x c   Offer to save any modified buffers, then kill Edwin, returning to Scheme (save-
                buffers-kill-edwin). This is like the suspend-edwin command, except that
                a subsequent call to edit will reinitialize the editor.

C-x C-z   Stop Edwin and suspend Scheme, returning control to the operating system's
                command interpreter (suspend-scheme). When Scheme is resumed (using
                the command interpreter's job-control commands), Edwin is automatically
                restarted where it was stopped. This command is identical to the C-x C-z
                command of GNU Emacs.

C-x C-c   Offer to save any modified buffers, then kill both Edwin and Scheme (save-
                buffers-kill-scheme). Control is returned to the operating system's com-
                mand interpreter, and the Scheme process is terminated. This command is
                identical to the C-x C-c command of GNU Emacs.

7.3 Scheme Mode

As you might expect, Edwin has special support for editing and evaluating Scheme
code. This section describes *Scheme Mode*, the appropriate mode for editing MIT Scheme
programs.

Scheme mode is normally entered automatically by visiting a file whose file name ends in '`.scm`'. You can also mark a file as Scheme code by placing the string '`-*-Scheme-*-`' on the first line of the file. Finally, you can put any buffer in Scheme mode by executing the command `M-x scheme-mode`.

Scheme mode is similar to the Emacs modes that edit Lisp code. So, for example, `C-i` indents the current line, and `C-M-q` indents the expression to the right of point. The close parenthesis will temporarily flash the matching open parenthesis. Most Scheme constructs requiring special indentation are recognized by Scheme mode, for example, `begin`, `do`, and `let`.

Scheme mode also provides support that is specific to Scheme programs, much as Emacs-Lisp mode does in Emacs. Completion of global variable names is provided: type the first few characters of a variable, then type `C-M-i`, and Edwin will attempt to complete the variable name using the current set of bound variables. If `C-M-i` is given a prefix argument, it will complete the name using the current set of interned symbols (which includes the bound variables as a subset).

The `M-A` command (note the *uppercase A*) will show the parameters of a procedure when point is inside a procedure call. For example, type the string '`(quotient`', then press `M-A`, and the command will echo '`(n d)`' in the echo area. With a prefix argument, `M-A` will insert the parameter names in the buffer at point, so in this example, the buffer would contain '`(quotient n d`' after running `C-u M-A`.

7.4 Evaluation

Scheme mode also provides commands for evaluating Scheme expressions. The simplest evaluation command is `C-x C-e`, which evaluates the expression to the left of point. (This key is bound in all buffers, even if they don't contain Scheme code.) The command `M-z` evaluates the definition that point is in (a definition is an expression starting with a left parenthesis in the leftmost column). The command `M-:` prompts for an expression in the minibuffer, evaluates it, and prints the value in the echo area.

Other commands that evaluate larger amounts of code are `C-M-z`, which evaluates all of the expressions in the region, and `M-o`, which evaluates the entire buffer. Both of these commands are potentially dangerous in that they will evaluate anything that appears to be an expression, even if it isn't intended to be.

Normally, these commands evaluate expressions by sending them to a REPL buffer, which performs the evaluations in a separate thread. This has two advantages: it allows you to continue editing while the evaluation is happening, and it keeps a record of each evaluation and its printed output. If you wish to stop a running evaluation and to erase any pending expressions, use the `C-c C-c` command from any Scheme buffer. (Note that by default, Edwin starts up with one REPL buffer, called '`*scheme*`'.)

If you would prefer to have Scheme mode evaluation commands evaluate directly, rather than sending expressions to the REPL buffer, set the Edwin variable `evaluate-in-inferior-repl` to `#f`. In this case, you will not be able to use Edwin while evaluation is occurring; any output from the evaluation will be shown in a pop-up buffer when the evaluation finishes; and you abort the evaluation using `C-g`.

7.5 REPL Mode

Edwin provides a special mechanism for interacting with Scheme read-eval-print loops: REPL buffers. A REPL buffer is associated with a Scheme REPL running in a separate thread of execution; because of this, expressions may be evaluated in this buffer while you simultaneously do other things with the editor. A REPL buffer captures all printed output from an evaluated expression, as well as supporting interactive programs such as `debug`. For these and other reasons, REPL buffers are the preferred means for interacting with the Scheme interpreter.

When Edwin starts, it has one buffer: a REPL buffer called '`*scheme*`'. The command `M-x repl` selects this buffer, if it exists; otherwise it creates a new REPL buffer. If you want two REPL buffers, just rename the '`*scheme*`' buffer to something else and run `M-x repl` again.

REPL buffers support all the same evaluation commands that Scheme mode does; in fact, REPL buffers use a special mode called REPL mode that inherits from Scheme mode. Thus, any key bindings defined in Scheme mode are also defined in REPL mode. One exception to this is the `M-o` command, which is deliberately undefined in REPL mode; otherwise it would be too easy to re-evaluate all the previously evaluated expressions in the REPL buffer.

In addition to evaluation commands, REPL mode provides a handful of special commands for controlling the REPL itself. The commands `C-c C-x` and `C-c C-u` abort the current evaluation, returning to the current or previous REPL levels, respectively. The command `C-c C-b` interrupts the current evaluation, entering a breakpoint.

Each REPL buffer maintains a history of the expressions that were typed into it. Several commands allow you to access the contents of this history. The command `M-p` moves backwards through the history, inserting previously evaluated expressions at point. Likewise, `M-n` moves forward through the history. The commands `C-c C-r` and `C-c C-s` search backward and forward through the history for a particular string. The command `C-c C-o` deletes any output from the previous evaluation; use this command with care since it cannot be undone. The command `C-c C-l` finds the most recent expression in the buffer and moves point there.

When an expression that you evaluate signals an error, the REPL buffer notices this and offers to run the debugger for you. Answer this question with a 'y' or 'n' response. You can start the debugger whenever the REPL buffer is listening by executing the `C-c C-d` command. In either case, this starts the Edwin debugger, which pops up a new window containing the debugger. Your REPL buffer remains in the error state, allowing you to examine it further if you wish.

7.6 The Edwin Debugger

The Edwin debugger is similar to the command-line debugger, except that it takes advantage of multiple windows and Edwin's command structure to provide a more intuitive interface. The debugger operates as a browser, much like Dired, presenting you with an overview of the subproblem structure, and allowing you to examine parts of that structure in more detail by selecting the parts. When started, the debugger creates a buffer '`*debug*`' showing the subproblem structure, and selects the first line.

Each line beginning with 'S' represents either a subproblem or stack frame. A subproblem line may be followed by one or more indented lines (beginning with the letter 'R') which represent reductions associated with that subproblem. The subproblems are indexed with the natural numbers. To obtain a more complete description of a subproblem or reduction, click the mouse on the desired line or move the cursor to the line using the arrow keys (or `C-n` and `C-p`). The description buffer will display the additional information.

The description buffer contains three major regions. The first region contains a pretty-printed version of the current expression. The current subproblem within the expression is highlighted. The second region contains a representation of the frames of the environment of the current expression. The bindings of each frame are listed below the frame header. If there are no bindings in the frame, none will be listed. The frame of the current expression is preceded with '==>'.

The bottom of the description buffer contains a third region for evaluating expressions in the environment of the selected subproblem or reduction. This is the only portion of the buffer where editing is possible. This region can be used to find the values of variables in different environments, or even to modify variable values or data structures (note that variables in compiled code cannot usually be modified).

Typing `e` creates a new buffer in which you may browse through the current environment. In this new buffer, you can use the mouse, the arrows, or `C-n` and `C-p` to select lines and view different environments. The environments listed are the same as those in the description buffer. If the selected environment structure is too large to display (i.e. if the number of bindings in the environment exceeds the value of the editor variable `environment-package-limit`) a message to that effect is displayed. To display the environment in this case, use `M-x set-variable` to set `environment-package-limit` to `#f`. At the bottom of the new buffer is a region for evaluating expressions, similar to that of the description buffer.

The appearance of environment displays is controlled by the editor variables `debugger-show-inner-frame-topmost?` and `debugger-compact-display?` which affect the ordering of environment frames and the line spacing respectively.

Type `q` to quit the debugger, killing its primary buffer, any others that it has created, and the window that was popped up to show the debugger.

Note: The description buffers created by the debugger are given names beginning with spaces so that they do not appear in the buffer list; these buffers are automatically deleted when you quit the debugger. If you wish to keep one of these buffers, simply rename it using `M-x rename-buffer`: once it has been renamed, it will not be automatically deleted.

7.7 Last Resorts

When Scheme exits abnormally it tries to save any unsaved Edwin buffers. The buffers are saved in an auto-save file in case the original is more valuable than the unsaved version. You can use the editor command `M-x recover-file` to recover the auto-saved version. The name used to specify an auto-save file is operating-system dependent: under unix, and on PC file systems with long file names, 'foo.scm' will be saved as '#foo.scm#'; on PC file systems with short file names, it will be saved as 'foo.sav'.

The following Scheme procedures are useful for recovering from bugs in Edwin's implementation. All of them are designed for use when Edwin is *not* running — they should

not be used when Edwin is running. These procedures are designed to help Edwin's imple-
mentors deal with bugs during the implementation of the editor; they are not intended for
casual use, but as a means of recovering from bugs that would otherwise require reloading
the editor's world image from the disk.

save-editor-files   procedure+
> Examines Edwin, offering to save any unsaved buffers. This is useful if some bug
> caused Edwin to die while there were unsaved buffers, and you want to save the
> information without restarting the editor.

reset-editor   procedure+
> Resets Edwin, causing it to be reinitialized the next time that `edit` is called. If you
> encounter a fatal bug in Edwin, a good way to recover is to first call `save-editor-`
> `files`, and then to call `reset-editor`. That should completely reset the editor to
> its initial state.

reset-editor-windows   procedure+
> Resets Edwin's display structures, without affecting any of the buffers or their con-
> tents. This is useful if a bug in the display code causes Edwin's internal display data
> structures to get into an inconsistent state that prevents Edwin from running.

Appendix A Release Notes

The previous full release of MIT Scheme was version 7.5.17. This section describes changes that have occurred since that time. For more detailed information, see the 'ChangeLog' files in the source code.

Note that MIT Scheme still conforms to the *Revised^4 Report on the Algorithmic Language Scheme*, but not to the *Revised^5 Report on The Algorithmic Language Scheme*.

A.1 Recent Changes

These are the changes since release 7.6.0:

- Release 7.7.1 fixes several bugs in IMAIL; fixes a bug that prevented the use of server sockets on Windows systems; and fixes a bug that caused the debugger to generate errors in common circumstances.

- Release 7.7.0 provides "hygienic" macro support, as defined in *R^4RS* and *R^5RS*. This is a complete rewrite of the syntax engine, so any program that uses macros should be rewritten to use the new engine. A subset of the old macro-definition syntax is still supported, but this will eventually be removed. Note that the new syntax engine has no effect on the compiled-code format; most binaries compiled by release 7.6.x should continue to work.

  There are several user-visible consequences to this change:

  - These syntactic keywords have been eliminated: `define-macro`, `in-package`, `macro`, `make-environment`, `scode-quote`, `unassigned?`, and `using-syntax`.

  - The syntactic keyword `the-environment` has been restricted to use in top-level environments. It is no longer allowed in the body of any binding form (e.g. `lambda`, `let`).

  - Syntactic keywords are now stored in environments, rather than in a separate syntax-table structure. The environment abstraction has been enhanced to support this, as well as to make it more general. The changes are documented in the reference manual.

  - The syntax-table abstraction has been eliminated, and most procedures and arguments involving syntax tables have been removed. One exception is the `load` procedure, which still accepts a syntax-table argument, but ignores it.

  Although the 7.6.1 release had a workaround for problems with certain AMD Athlon processors, the workaround was ineffective on machines running Windows operating systems (and possibly OS/2 systems as well). This version fixes that problem.

  The hash-table abstraction is now always loaded. It's not necessary to call `load-option` prior to use of hash tables. For upwards compatibility, calling '(`load-option 'hash-table`)' is still permitted but does nothing.

- Release 7.6.1 provides a workaround for a bug on some AMD Athlon processors (specifically, models 1, 3, and 4). This bug causes those processors to incorrectly execute certain kinds of self-modifying code, including that used by MIT Scheme.

  Release 7.6.1 also fixes a problem that caused the *PARSER and XML subsystems not to load correctly on Windows and OS/2 systems.

A.2 Older Changes

These are the changes that occurred between releases 7.6.0 and 7.5.17:

- The variable-reference machinery has been completely redesigned. The new design provides two new features: (1) a procedure `link-variables` that can link two variable bindings with different names (previously `environment-link-name` could only link bindings with the same name); (2) a new procedure `unbind-variable` that can remove a variable binding. Both of these operations work only on "top-level" environments; they don't work on environments created by invoking compiled procedures.

  This is an incompatible change to the compiled-code representation. All programs must be recompiled.

- The "packaging" subsystem has been redesigned to make it more flexible. The new design takes advantage of the improved variable-linking mechanism to allow packages to export bindings using names different from their internal names. Also, each package now maintains an in-memory data structure that should (in principal) allow it to be uninstalled or reinstalled. This subsystem is still evolving; expect further changes in the future.

  This subsystem is now included in the released binaries; it can be loaded by '`(load-option 'cref)`'.

- Compiled-code debugging information is now keyed to prevent using the wrong debugging info when examining compiled code. This means that in-memory data structures have an identifying key that is matched against a corresponding key in the debugging-info file. The debugging info is used only when the keys match.

- MIT Scheme now has a non-validating XML parser that is mostly conformant (except that it doesn't support UTF-16).

- There is very limited support for UTF-8. At present this is only used by the XML parser.

- All string and character procedures now operate on ISO-8859-1 rather than US-ASCII as the base character set. This isn't real internationalization support, but at least it broadens the character support slightly. This involved changing a number of procedures that were formerly written in C and are now written in Scheme, which may have a performance impact on some uses.

- The procedure `length` now detects circularities in its argument and signals an error.

- A new `rexp` facility provides a more readable syntax for specifying regular expressions. This syntax can be compiled into the usual syntax.

- There is a new *parser buffer* abstraction that supports the building of parsers with backtracking. An associated high-level parser description language allows parsers to be described in a BNF-like language that is compiled into efficient Scheme code.

- A new procedure `flo:finite?` returns '`#f`' for infinite or '`NaN`' flonums and '`#t`' for all others.

- The signal-handling code for GNU/Linux systems has been changed to take advantage of additional information identifying the types of arithmetic exceptions. This allows more accurate error messages to be generated.

- The following procedure names were changed:

  Old name   New name

```
list-transform-positive              keep-matching-items
list-transform-negative              delete-matching-items
list-search-positive                 find-matching-item
```

and the following new procedures were implemented: `keep-matching-items!` and `delete-matching-items!`.

- IMAIL now supports commands to sort folders according to various criteria. The design is similar to that of Rmail.

- IMAIL has been re-engineered to improve performance on very large folders. There is now a disk cache for each IMAP folder, to eliminate unnecessary network delays. Also, the summary buffer previously used a quadratic algorithm to associate lines in the summary with messages in the folder; it now uses a linear algorithm.

- A new IMAIL command `imail-file-message` appends the text of a message to a file more or less as it appears in the buffer. This is intended to allow collecting messages into a file in a human-readable format.

- IMAIL used to determine the type of file folders by their file names; now it examines the file contents instead. When creating a new file folder, it prompts for the type.

GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long

as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

   D. Preserve all the copyright notices of the Document.

   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

   F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

   G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

   H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may

include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document 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.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation 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. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ''GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being list"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

Table of Contents