MIT/GNU Scheme User's Manual

for release 12.0.90
2020-06-12

by Stephen Adams
Chris Hanson
and the MIT Scheme Team

This manual documents the use of MIT/GNU Scheme 12.0.90.

Table of Contents

Introduction

This document describes how to install and use MIT/GNU 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.

This document discusses many operating-system specific features of the MIT/GNU 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, macOS, and the BSD variants.

The primary distribution site for this software is

`https://www.gnu.org/software/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.

The release notes for the current release are at

`https://www.gnu.org/software/mit-scheme/release.html`

To report bugs, use the bug-reporting tool at

`https://savannah.gnu.org/projects/mit-scheme/`

Please include the output of the `identify-world` procedure (see Section 2.1 [Basics of Starting Scheme], page 5), so we know what version of the system you are using.

1 Installation

This chapter describes how to install MIT/GNU Scheme. The release is supported under various unix operating systems. 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. There are several references to *ARCH* below; these refer to the computer architecture that Scheme is compiled for: either '`i386`' '`x86-64`', '`aarch64`', or '`svm1`'.

MIT/GNU Scheme is distributed as a compressed 'tar' file. The tar file contains both source and binary files; the binary files are pre-compiled Scheme code for a particular computer architecture. The source files are C programs that need to be compiled.

Requirements

At a minimum, you will need a C compiler (e.g. '`gcc`') and a '`make`' program, and a "curses" library. For example, here are the packages that must be installed on some popular systems:

- Debian-like systems: `gcc make m4 libncurses-dev`
- CentOS-like systems: `gcc make m4 ncurses-devel`
- macOS systems: Command line developer tools '`xcode-select --install`'

Additionally, if you want support for X11 graphics, you'll need:

- Debian-like systems: `libx11-dev`
- CentOS-like systems: `libX11-devel`
- macOS systems: `XQuartz` (from `https://www.xquartz.org/`)

Steps

In order to install the software, it's necessary to configure and compile the C code, then to install the combined C and Scheme binaries, with the following steps.

1. Unpack the tar file, `mit-scheme-`*`VERSION`*`-`*`ARCH`*`.tar.gz`, into the directory `mit-scheme-`*`VERSION`*. For example,

   ```
   tar xzf mit-scheme-VERSION-i386.tar.gz
   ```

   will create a new directory `mit-scheme-`*`VERSION`*.

2. Move into the `src` subdirectory of the new directory:

   ```
   cd mit-scheme-VERSION/src
   ```

3. Configure the software:

   ```
   ./configure
   ```

   By default, the software will be installed in `/usr/local`, in the subdirectories `bin` and `lib`. If you want it installed somewhere else, for example `/opt/mit-scheme`, pass the `--prefix` option to the configure script, as in *`./configure --prefix=/opt/mit-scheme`*.

The configure script accepts all the normal arguments for such scripts, and additionally accepts some that are specific to MIT/GNU Scheme. To see all the possible arguments and their meanings, run the command `./configure --help`. However, do not specify the following options, which are all preconfigured to the right values; doing so will probably cause the build to fail:

```
--enable-native-code
--enable-host-scheme-test
--enable-cross-compiling
--with-compiler-target
--with-default-target
--with-scheme-build
```

4. Build the software:

   ```
   make
   ```

5. Install the software:

   ```
   make install
   ```

   Depending on configuration options and file-system permissions, you may need superuser privileges to do the installation steps.

6. Build the documentation:

   ```
   cd ../doc
   ./configure
   make
   ```

7. Install the documentation:

   ```
   make install-info install-html install-pdf
   ```

   Depending on configuration options and file-system permissions, you may need superuser privileges to do the installation step.

Plugins

After you have installed Scheme you may want to install several *plugins*. Scheme no longer uses dynamically loaded microcode modules installed with Scheme. The micromodules have been converted into plugins: new subsystems that use the C/FFI to dynamically load the same code. Instead you configure, build, and install additional plugins after installing the core system.

By default, the following plugins are built and installed: `edwin`, `imail`, `x11`, and `x11-screen`. (The latter two only if X11 libraries are installed on your system.) To get all of the functionality previously available in version 9.2 you will need to build and install the remaining plugins included in the `src` subdirectory: `blowfish`, `gdbm`, and `pgsql`. These plugins are all configured, built, and installed in the GNU standard way. See the `README` file in each plugin's source directory for complete details.

Cleanup

After installing Scheme and your desired plugins, you can delete the source directory:

```
cd ../..
rm -rf mit-scheme-VERSION
```

2 Running Scheme

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

2.1 Basics of Starting Scheme

Under unix, MIT/GNU Scheme is invoked by typing

```
mit-scheme
```

at your operating system's command interpreter. In either case, Scheme will load itself and print something like this:

```
Copyright (C) 2019 Massachusetts Institute of Technology
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Image saved on Tuesday May 26, 2020 at 10:23:04 PM
  Release 10.90 || SF || LIAR/x86-64
```

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.

Following this there may be additional names for specific subsystems. '`SF`' refers to the scode optimization program `sf`; '`LIAR/`*ARCH*' is the native-code compiler, where *ARCH* is the native-code architecture it compiles to; '`Edwin`' is the Emacs-like text editor. There are other subsystems you can load that will add themselves to this list.

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 7.
- *Shell scripts*. 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 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`.

  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 (see Section 8.1 [Starting Edwin], page 55).

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 7).

- *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 10.
- *Icons*. 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; any unreachable objects in constant space remain there until the Scheme process is terminated. Constant space is used for objects that are essentially permanent, like procedures in the runtime system. Doing this reduces the expense of garbage collection because these objects are no longer copied.
- 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/GNU Scheme uses a two-space copying garbage collector for reclaiming storage in the heap. The second space, used only during garbage collection, is dynamically allocated as needed.

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/GNU 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; 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 22).

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 9.

Note that MIT/GNU Scheme supports only *long* options, that is, options specified by verbose names, as opposed to *short* options, which are specified by single characters. All options start with two hyphens, for compatibility with GNU coding standards (and most modern programs).

These are the microcode options:

`--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, `all.com`; in these cases the library directories are searched, but not the working directory.

`--heap` *`blocks`*
: Specifies the size of the heap in 1024-word blocks. Overrides any default. 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 Bourne shell, 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:

```
mit-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-`*ARCH*.

`--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 `--band` option. This option is useful only for maintenance and development of the MIT/GNU Scheme runtime system.

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 `${HOME}/.scheme.init` 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.

[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.

`--eval` *expression* `...`
: This option causes Scheme to evaluate the *expression*s following it on the command line, up to but not including the next argument 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 argument that starts with a hyphen. The files are loaded in the `user-initial-environment`. Unless explicitly handled, errors during loading are silently ignored.

`--edit`
: This option causes Edwin to be loaded and started immediately when Scheme is started.

The following options allow arguments to be passed to scripts via the `command-line-arguments` procedure.

`command-line-arguments` [procedure]
: Returns a list of arguments (strings) gathered from the command-line by options like `--args` or `--`.

`--args` *argument* `...`
: This option causes Scheme to append the *argument*s, up to (but not including) the next argument that starts with a hyphen, to the list returned by the `command-line-arguments` procedure.

`--` *argument* `...`
: This option causes Scheme to append the rest of the command-line arguments (even those starting with a hyphen) to the list returned by the `command-line-arguments` procedure.

2.5 Custom Command-line Options

MIT/GNU 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 21). 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 containing at least one character; do not include the leading hyphens.

`simple-command-line-parser` *keyword thunk* [*help*] [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. *Help*, when provided, should be a string describing the option in the `--help` output.

`argument-command-line-parser` *keyword multiple? procedure* [*help*] [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. *Help*, when provided, should be a string describing the option. It is included in the `--help` output. When not provided, `--help` will say something lame about your command line option.

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 either `#f` or a thunk to invoke after the whole command line has been parsed (and the init file loaded). Thus *procedure* has the option of executing the appropriate action at parsing time, or delaying it until after the parsing is complete. The execution of the procedures (or their associated delayed actions) is strictly left-to-right, with the init file loaded between the end of parsing and the delayed actions.

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/GNU Scheme that they affect.

Environment variables that affect the microcode must be defined before you start Scheme; 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 program called `mit-scheme-`*`ARCH-VERSION`*. The values they specify are overridden by the corresponding command-line options, if given.

`MITSCHEME_BAND`

The initial band to be loaded. The default value is `all.com`.

`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-`*`ARCH-VERSION`*.

`MITSCHEME_CONSTANT`
: The size of constant space, in 1024-word blocks; overridden by `--constant`. The default value is computed to be the correct size for the band being loaded.

`MITSCHEME_HEAP_SIZE`
: The size of the heap, in 1024-word blocks; overridden by `--heap`. The default value depends on the architecture: for 32-bit machines the default is '`3072`', and for 64-bit machines the default is '`16384`'.

`MITSCHEME_STACK_SIZE`
: The size of the stack, in 1024-word blocks; overridden by `--stack`. The default value is '`1024`'.

2.6.2 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, for example `/home/joe`. 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: `/var/tmp`, `/usr/tmp`, `/tmp`.

`MITSCHEME_INF_DIRECTORY`
: Directory containing the debugging information files for the Scheme system. Should contain subdirectories corresponding to the subdirectories in the source tree. By default, the information is searched for on the library path.

`MITSCHEME_LOAD_OPTIONS`
: Specifies the location of the options database file used by the `load-option` procedure. The default is `optiondb.scm` on the library path.

2.6.3 Environment Variables for Edwin

These environment variables are referred to by Edwin.

`EDWIN_BINARY_DIRECTORY`
: Directory where Edwin expects to find files providing autoloaded facilities. The default is `edwin` on the library path.

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

`EDWIN_ETC_DIRECTORY`
: Directory where Edwin expects to find utility programs and documentation strings. The default is `edwin` on the library path.

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

`SHELL` Filename of the shell program to use in shell buffers and when executing shell commands. Used to initialize the `shell-path-name` editor variable. The default is `/bin/sh` on unix systems.

`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` Used when Edwin runs under unix on a terminal. Number of text lines on the screen, for systems that don't support '`TIOCGWINSZ`'.

`COLUMNS` Used when Edwin runs under unix on a terminal. Number of text columns on the screen, for systems that don't support '`TIOCGWINSZ`'.

2.7 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 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 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.

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 they are `C-g` and `C-c`. 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.

`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` 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` 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` 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.

`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 are some procedures that manage 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.

`ve` *environment* [procedure]
: Starts a sub-REPL with it's environment set to *environment* (`ve` stands for "Visit 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.

3.1.5 REPL Escapes

Normally the REPL evaluates an expression and prints the value it returns. The REPL also supports a set of special *escapes* that bypass the normal evaluation. There are two kinds of escapes:

`,(`*command arg ...*`)`
`,`*command*
: tells the REPL to perform a special action. The symbol *command* specifies the action to perform; the *arg* elements are command specific. A command that can be used with no *arg* elements can be abbreviated by dropping the parentheses. Additionally, *command* can be shortened to any unique prefix, such as `po` for `pop`. Note that *command* is not evaluated. An *arg* is not evaluated, unless it starts with a comma, in which case it is evaluated in the current REPL environment.

`,,`*expression*
: evaluates *expression* in `user-initial-environment` instead of the current REPL environment. This is especially useful when working with library environments, where many of the usual definitions, for example `debug`, are not available.

The rest of this section documents the commands that can be used with the first form of escape. The most important command is `help`:

`help` [*name*] [REPL command]
: Prints each of the available commands along with a summary of what they do. If *name* is given, show only commands that match *name*.

```
,(help p)
⊣ ;,pop
⊣ ;    Pops an environment off the stack and moves the REPL there.
⊣ ;,push
⊣ ;,(push env)
⊣ ;    Push the REPL env on the env stack and move the REPL to a new env.
⊣ ;
⊣ ;    If ENV is provided, it is converted to an environment in the usual
⊣ ;    way.  The the current REPL env is pushed on the env stack and the REPL
⊣ ;    is moved to ENV.
⊣ ;
⊣ ;    If ENV is not provided, the current REPL env is exchanged with the top
⊣ ;    of the env stack.
```

A number of the commands manipulate the REPL's environment in various ways. These involve the following parts:

- The current REPL environment is the environment that's used to evaluate expressions.
- The *environment stack* contains additional environments that are saved for future use. This stack is modified by the `push`, `pop`, `bury`, and `ge` commands.
- A set of *named environments* that have been given symbolic names. This set is modified by `name` and `unname`.

`envs` [*env-name*] [REPL command]

Prints a summary of the environments. If *env-name* is given, prints only the named environments matching *env-name*.

For example, here is the output when the system is started:

```
,envs
⊣ ;here: (user) #[environment 12]
⊣ ;The env stack is empty
⊣ ;no named envs
```

Where `;here:` marks the current REPL environment.

Several commands take an *env* argument, specifying an environment. This argument can have several forms:

a symbol Refers to a named environment.

a library name

Refers to the environment of a loaded library. For example, '`(scheme base)`'.

a package name

Refers to the environment of a loaded MIT/GNU Scheme package. For example, '`(runtime)`'.

`,`*expression*

Evaluates *expression* in the current environment; its value must be an environment object.

`push` [*env*] [REPL command]

Pushes the current REPL environment on the environment stack, then moves the REPL to a new environment. If *env* is not given, then this swaps the current REPL

environment and the environment on the top of the stack. Otherwise *env* specifies the new environment in the usual way.

If the command completes successfully, it prints the current REPL environment and the environment stack:

```
,(push (srfi 133))
⊣ ;here: #[environment 28]
⊣ ;stack:
⊣ ;    0: (user) #[environment 12]
```

We can swap the two environments:

```
,push
⊣ ;Package: (user)
⊣ ;here: (user) #[environment 12]
⊣ ;stack:
⊣ ;    0: #[environment 28]
```

`pop` [REPL command]

Pops off the top of the environment stack and moves the current REPL environment there.

```
,pop
⊣ ;Package: (user)
⊣ ;here: (user) #[environment 12]
⊣ ;The env stack is empty
```

`bury` [REPL command]

Saves the current REPL environment at the bottom of the stack, then pops off the top of the environment stack and moves the current REPL environment there.

```
,(push (runtime))
⊣ ;Package: (runtime)
⊣ ;here: (runtime) #[environment 30]
⊣ ;stack:
⊣ ;    0: #[environment 28]
⊣ ;    1: (user) #[environment 12]

,bury
⊣ ;here: #[environment 28]
⊣ ;stack:
⊣ ;    0: (user) #[environment 12]
⊣ ;    1: (runtime) #[environment 30]
```

`ge` [*env*] [REPL command]

Sets the current REPL environment to the specified environment without affecting the environment stack. If *env* is not given, a newly created top-level environment is used.

This is basically the same as the `ge` procedure.

`ve` [*env*] [REPL command]
: Creates a new child REPL, setting its current environment to the specified one. If *env* is not given, a newly created top-level environment is used.

  This is basically the same as the `ve` procedure.

`name` *env-name* [REPL command]
: Gives the current REPL environment a name *env-name* and adds it to the set of named environments. The argument *env-name* must be a symbol.

```
,(name foobar)
⊣ ;env named foobar has been assigned

,envs
⊣ ;here: foobar #[environment 28]
⊣ ;stack:
⊣ ;    0: (user) #[environment 12]
⊣ ;    1: (runtime) #[environment 30]
⊣ ;named envs
⊣ ;    foobar #[environment 28]
```

`unname` [*env-name*] [REPL command]
: Removes the environment with name *env-name* from the set of named environments. If *env-name* is not given, removes all named environments.

```
,(unname foobar)
⊣ ;env named foobar has been unassigned

,envs
⊣ ;here: #[environment 28]
⊣ ;stack:
⊣ ;    0: (user) #[environment 12]
⊣ ;    1: (runtime) #[environment 30]
⊣ ;no named envs
```

This group of commands manages nested REPL instances.

`down` [REPL command]
: Creates a new child REPL with the same current environment as this one.

`import` *import-set* ... [REPL command]
: Imports the given *import-set*s into the current REPL environment. The syntax is described in R7RS section 5.2.

`up` [REPL command]
: Pops up one level to the parent REPL.

  This is equivalent to calling `cmdl-interrupt/abort-previous`.

`top-level` [REPL command]
: Pops up to the top-level REPL.

  This is equivalent to calling `cmdl-interrupt/abort-top-level`.

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 22). 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 names ending in `.scm`, and files of binary SCode have names ending in `.bin`. Native-code binaries have names ending in `.com`. R7RS library files conventionally end in `.sld`, `.binld`, and `.comld` respectively.

If no file-name suffix is specified, `load` will choose a file by trying different suffixes, preferring in order native-code binaries, SCode binaries, and source files.

All file names 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/GNU Scheme Reference Manual*.

`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/GNU Scheme Reference Manual*.

`gdbm` Support to access `gdbm` databases. Undocumented; see the source files `runtime/gdbm.scm` and `microcode/prgdbm.c`.

`ordered-vector`
Support to search and do completion on vectors of ordered elements. Undocumented; see the source file `runtime/ordvec.scm`.

`regular-expression`
Support to search and match strings for regular expressions. See Section "Regular Expressions" in *MIT/GNU 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/GNU 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 `all.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 6, for a discussion of how MIT/GNU 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.) **Caution:** 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. See Section 2.3 [Memory Usage], page 6.

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 minimal 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:   2302316 words =   2248 blocks +  364 words
constant free:         128 words =      0 blocks +  128 words
heap in use:       1747805 words =   1706 blocks +  861 words
heap free:        49682723 words =  48518 blocks +  291 words
```

`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

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`. It will also produce `foo.bin`, `foo.bci`, and possibly `foo.ext`. The corresponding names for R7RS libraries are `foo.sld`, `foo.comld`, `foo.binld`, and `foo.bcild` (libraries never generate `.ext` files). If you later evaluate

```
(load "foo")
```

`foo.com` (or `foo.comld`) will be loaded.

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.

`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 the suffix `.bin`. If you do not specify a suffix 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

This section doesn't apply to R7RS source or library files, since their environments are completely specified by `import` statements.

Other source files usually contain 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 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) f (cons x (cons y (cons z w))))
(apply f x y)
   ↦ ((access apply #f) 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 remaining 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) ...)
```

4.3.2 Top-level 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 top-level 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!), which is a great reason to use `parameterize` instead.

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 Type and range checking

The compiler inserts type (and range) checks so that e.g. applying `vector-ref` to a string (or an invalid index) signals an error. Without these checks, an in-lined `vector-ref` application will return garbage — an object with random type and address. At best, accessing any part of that object will produce an invalid address trap. At worst, the garbage collector is confused and your world is destroyed.

The compiler punts type and range checks when it can prove they are not necessary. Using "Better Predicates" helps (see Section 4.3.1 [Coding style], page 32), but many checks will remain. If you know a data structure will be read-only, a certain size, etc. many of the remaining checks can prove unnecessary. To make these decisions for yourself, you can turn off the compiler's implicit checks. The following procedure definition ensures minimum flonum consing (i.e. none, see Section 4.3.5 [Flonum arithmetic], page 36) and maximum speed. It's safe use is entirely up to you.

```
(declare (usual-integrations) (integrate-operator %increment!))
(define (%increment! v i)
  (declare (no-type-checks) (no-range-checks))
  (flo:vector-set! v i (flo:+ (flo:vector-ref v i) 1.)))
```

Here are the relevant declarations:

`no-type-checks` [declaration]

In-lined primitives within the block will not check their arguments' types.

`no-range-checks` [declaration]

In-lined primitives within the block will not check that indices are valid.

4.3.4 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/GNU Scheme, fixnums are 26 bits on 32-bit machines, and 58 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/GNU Scheme Reference Manual*.

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

4.3.5 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/GNU Scheme uses) does not fit in a regular machine word on 32-bit machines. Boxing is also done on 64-bit machines 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 on 32-bit machines and about one-third on 64-bit machines) 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 `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 `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/GNU Scheme. One of them runs under Edwin, and is described in that section of this document (see Section 8.6 [Edwin Debugger], page 58). 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 inspector (`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/GNU 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/GNU 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 environments 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 environment, 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 12 (user)]
⊣ #[package 13 ()]
⊣ alist-table-search
⊣ re-string-search-backward
⊣ re-string-search-forward
```

```
⊣ re-substring-search-backward
⊣ re-substring-search-forward
⊣ regexp-search
⊣ regexp-search-all
⊣ regsexp-search-string-forward
⊣ search-gc-finalizer
⊣ search-ordered-subvector
⊣ search-ordered-vector
⊣ string-search-all
⊣ string-search-backward
⊣ string-search-forward
⊣ substring-search-all
⊣ substring-search-backward
⊣ substring-search-forward
⊣ vector-binary-search
⊣ weak-alist-table-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 *procedure* 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 procedure'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 Profiling

MIT/GNU Scheme provides a simple-minded statistical profiler called the stack sampler, which periodically interrupts the program and records the return addresses that it finds on the stack. For each return address, the stack sampler records two counts: the number of times that return address was at the top of the stack, called the *sampled* count, and the number of times that return address was somewhere else in the stack, called the *waiting* count. The topmost 'return address' may correspond with a procedure rather than a continuation, if the procedure was about to be called in a tail position when the stack sampler interrupted.

If a return address has a high sampled count, it is in a specific part of the program that may warrant micro-optimization. If a return address has a high waiting count, then the program spends a long time computing the expression for whose value continuations corresponding with the return address are waiting, and the expression may warrant a better algorithm.

The stack sampling is very coarse-grained, because the program is interrupted only at safe points, which are a subset of procedure calls and returns. Another approach to profiling is to record the address of the machine instruction the program is about to have the machine execute, and then to find where in the program that instruction is. This could provide finer-grained sampled counts, but it requires a lower-level implementation, and cannot provide waiting counts, because the stack may not be in a consistent, inspectable state except at safe points.

Finally, the stack sampler gives information only about compiled code; it ignores continuations on the stack for interpreted code. This is because continuations for compiled code are easier to deal with, and in any case, if a program is too slow when interpreted, the first step is to compile it, not to profile it.

`with-stack-sampling` *interval procedure* [procedure]
: Applies *procedure* to zero arguments. During the dynamic extent of the call to *procedure*, the stack sampler interrupts the program at intervals of approximately *interval* milliseconds. When *procedure* returns, `with-stack-sampling` displays the waiting and sampled counts it gathered.

  More precisely, after each sample, the stack sampler will not sample again before *interval* milliseconds of real time (in the sense of `real-time-clock`; see Section "Machine Time" in *MIT/GNU Scheme Reference Manual*) have elapsed, although more time may elapse before the next sample.

`stack-sampler:show-expressions?` [variable]
: If true, the output of `with-stack-sampling` shows the subexpression corresponding with each return address, and the expression enclosing it. If false, the output is shorter (one line per return address), and just shows the names of procedures forming the environment hierarchy corresponding with the return address.

`stack-sampler:debug-internal-errors?` [variable]
: If false, the default, errors signalled while recording a sample are ignored. Set this to true only if you are debugging the internals of the system.

7 GNU Emacs Interface

GNU Emacs is distributed with a loadable package `xscheme`, which facilitates running Scheme as a subprocess of Emacs.

To invoke Scheme from Emacs, load the `xscheme` package (for example by '`(require 'xscheme)`', 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 the `xscheme` package is loaded into Emacs, `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 in a terminal. (`xscheme-send-control-g-interrupt`)

`C-c C-x` Like typing `C-c C-x` when running Scheme in a terminal. (`xscheme-send-control-x-interrupt`)

`C-c C-u` Like typing `C-c C-u` when running Scheme in a terminal. (`xscheme-send-control-u-interrupt`)

`C-c C-b` Like typing `C-c C-b` when running Scheme in a terminal. (`xscheme-send-breakpoint-interrupt`)

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

8 Edwin

This chapter describes how to start Edwin, the MIT/GNU Scheme text editor. Edwin is a clone of GNU Emacs version 18—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/GNU Scheme, while GNU Emacs extensions are written in Emacs Lisp. This manual does not discuss customization of Edwin.

8.1 Starting Edwin

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

```
mit-scheme --edit
```

Alternatively, you can load Edwin by 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) 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 [Customizing Scheme], page 5).

`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 geometry 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 terminology, 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.

8.2 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 corresponds 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 command interpreter, and the Scheme process is terminated. This command corresponds to the *C-x C-c* command of GNU Emacs.

8.3 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/GNU 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*.

8.4 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*.

8.5 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.

8.6 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.

8.7 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 `foo.scm` will be saved as `#foo.scm#`.

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 implementors 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 contents. 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 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful 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 non-commercially. 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, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

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. (Thus, 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. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

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 Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document 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, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. 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, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include 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, PostScript or PDF 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.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

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 (or copies in media that commonly have printed covers) 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 computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. 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 fewer than five), unless they release you from this requirement.

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", Preserve 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. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, 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 to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

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, and that you preserve all their Warranty Disclaimers.

   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, is called

an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which 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 half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket 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, and all the license notices in the Document, and any Warrany Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

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 `https://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.

A.1 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 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

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.

Appendix B Environment-variable Index

Appendix C Option Index

Appendix D Variable Index

P

Q

R

S

T

U

V

W

X

Appendix E Concept Index

P

R

S

T

U

V

W