MIT Scheme User's Manual

Edition 1.52
for Scheme Release 7.4
16 April 1996

by Stephen Adams
Chris Hanson
and the MIT Scheme Team

Introduction

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

This document discusses many operating-system specific features of the MIT Scheme implementation. In order to simplify the discussion, we use abbreviations to refer to some operating systems. When the text uses the term *Unix*, this means any of the Unix systems that we support, including GNU/Linux, HP-UX, Ultrix, NeXT, and SunOS. The term *OS/2* means the IBM OS/2 operating system, version 2.1, 2.11, or 3.0 Warp. We use the term *Windows* to collectively refer to the Microsoft Windows operating systems: Windows 3.1, Windows 95, and Windows NT. We use the term *PC* to refer to any computer running OS/2 or Windows. Thus we consider a PC to be a system with a DOS-like file system, using backslashes for directory separators, drive letters, CR-LF line termination, and (potentially) the hideous 8.3 short filenames.

The primary distribution site for this software is the FTP server '`swiss-ftp.ai.mit.edu`'. Although our software is distributed from other sites and in other media, the complete distribution and the most recent release is always available at our site.

To report bugs, send email to '`bug-cscheme@zurich.ai.mit.edu`'. Please include the output of the `identify-world` procedure (see Section 2.1 [Basics of Starting Scheme], page 15), so we know what version of the system you are using.

1 Installation

This chapter describes how to install MIT Scheme release 7.4. The release is supported under several different operating systems: Unix, OS/2, and Windows. Read the section detailing the installation for the operating system that you are using.

If you are installing on an HP 9000 series 700 or 800, you should also read Section 1.2 [HP-PA Installation], page 5.

1.1 Unix Installation

We will use as an example the installation for GNU/Linux. The installation for other Unix systems is similar. If you are installing on an HP 9000 series 700 or 800, see also Section 1.2 [HP-PA Installation], page 5.

MIT Scheme is distributed as a compressed 'tar' file. The tar file contains a single directory, called '`dist-7.4`', and that directory contains subdirectories '`bin`', '`etc`', and '`lib`'. The '`bin`' subdirectory contains two executable files, '`scheme`' and '`bchscheme`'. The '`etc`' subdirectory contains several files that are used during the installation process. The '`lib`' subdirectory contains several files and subdirectories that Scheme uses while it is executing.

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

- First unpack the distribution:

  ```
  gzip -cd linux.tar.gz | tar xvf -
  ```

  If you do not have the 'gzip' program: we have built executables of this program for each of the machines that we support. Either use the executable that we have already built, or else get the gzip sources (from us or from numerous other sources) and build it yourself.
- Next, go to the directory that you just unpacked:

  ```
  cd dist-7.4
  ```

  The easiest way to install the files is to use the "install.sh" shell script that we've provided in the distribution. You will need to edit that script to tell it where and how you want Scheme installed. If you want to do something more complicated, or if the script can't easily be made to work for your system, use the script as a guide to installation.

- The script assumes there is no pre-existing installation of MIT Scheme on your system. If there is, you may need to edit the script, or delete the existing version (particularly the contents of '`/usr/local/lib/mit-scheme`', see the later note on editing '`libdir`'). If you have trouble deleting the binary because it is in use, try renaming it to '`scheme.busy`' and deleting it later.

- The installation script normally uses the command '`cp -pr`' to copy the files from the distribution directory to their final locations. You can change this by editing the definition of the '`install`' variable in the '`install.sh`' shell script. Some versions of '`cp`' do not support the '`p`' option; using simply '`cp -r`' in this case is acceptable. You can also use the '`mv`' command to move the files and directories into place, but note that most versions of '`mv`' require the source and destination to be on the same physical device.

- The installation script may try to make some symbolic links during the installation. This is normally done using the command '`ln -s`'. If your system doesn't support symbolic links, or if you prefer to copy or hard-link files, change the definition of the '`link`' variable in the shell script.

- For many systems, a standard place to put executable files is the directory '`/usr/local/bin`'. If this is true for your system, we suggest installing the executables there. Otherwise, you probably have some other directory that serves the same purpose; edit '`install.sh`' and change the definition of the shell variable '`bindir`' to be this other directory.

- Normally, auxiliary files are stored in the directory '`/usr/local/lib/mit-scheme/`'. If you can, we suggest that you create such a directory and install these files there. Otherwise, create a directory somewhere else and edit the installation script to change the value of the variable '`libdir`' to be this directory.

- If you have GNU Emacs installed on your system, the installation script will attempt to automatically determine values for the script variables '`infodir`', '`emacslisp`', and '`movemail`'. If it guesses incorrectly, you will have to override the guesses by editing the script with appropriate values.

- Once you are satisfied that the '`install.sh`' script is correct, execute it:

  ```
  ./install.sh
  ```

- After '`install.sh`' finishes, you can delete the '`dist-7.4`' directory that you unpacked earlier.

  You should now be able to run MIT Scheme by typing

  ```
  scheme
  ```

  at the command line (if you use the C shell, you may have to type '`rehash`' before '`scheme`' will be recognized).

1.2 HP-PA Installation

If you are using an HP 9000 series 700/800 computer (often called an HP Precision Architecture machine, or HP-PA for short), read this section. You should also read Section 1.1 [Unix Installation], page 3.

Scheme has built-in code that flushes the instruction and data caches of your machine in certain circumstances. This code is sensitive to your computer's model, because each model has different cache hardware.

This distribution contains a database, called '`hppacach.mod`', that describes the cache structure for each model of computer. As of this writing, that database contains entries for the following models:

```
705, 710, 712, 715, 720, 725, 730, 735, 750, 755, 770,
816, 834, 835, 850, 867, 877
```

If you have a model that is not in the database, Scheme will not run; instead it will print an error message informing you of this fact, and stop. If this happens, you must add an entry to the database. This must be done once, at installation, for each new model.

Here is the procedure for updating the database:

- Run the program '`hppacach`' that is included in the distribution. You must give this program the filename of the database file as an argument. Normally the database file name is '`/usr/local/lib/mit-scheme/hppacach.mod`', but if you install Scheme in a non-standard place it will be different. For example, if you install Scheme as '`/usr/foo/mit-scheme/`', the database file is called '`/usr/foo/mit-scheme/hppacach.mod`'.
- Assuming that Scheme is installed in the normal place, update the database by becoming the super-user and executing the following commands on your machine:

  ```
  cd /usr/local/lib/mit-scheme
  ./hppacach -update hppacach.mod
  ```

  The reason that you must be super-user is that '`hppacach`' needs to read the device '`/dev/kmem`' to get the information that it needs. Normally, '`/dev/kmem`' is readable only by the super-user, or by users in group `sys`. Thus, becoming super-user is the easiest way to read this information. An alternative method for doing this is to change the permissions of the '`hppacach`' program so that it is in group `sys`, and to turn on its "set group ID on exec" permission bit, but since this also requires you to be super-user, you might as well just execute the program as the super-user.

Please note that you must execute this program on the computer whose model you wish to add to the database. Also, if you wish to add several models to the database, you must execute the program once on each model. If you have several computers that are all of the same model, you need only update the database once from one of the computers; thereafter all of the other computers of that model will work.

- If you add a new model to the database, we'd appreciate it if you would send us the model information, so that we can update the copy of the database that we distribute to others. This can be done using the 'hppacach' program, as follows:

  ```
  cd /usr/local/lib/mit-scheme
  ./hppacach -print hppacach.mod > model-7xx
  ```

  (If you have previously updated the database, you do not need to be super-user to execute this command.) The resulting file, 'model-7xx' in this example, is the model information for the machine that you executed the command on; it is C code that we can use to update our copy of the database. Send the model information for each new model to us by electronic mail at bug-cscheme@zurich.ai.mit.edu. Thanks!

1.3 OS/2 Installation

This section describes how to install MIT Scheme on a machine running OS/2 2.1 or OS/2 Warp 3.0.

1.3.1 Prerequisites

The Scheme files use about 20 megabytes of disk space when installed. An additional 5 megabytes of disk space is required during installation. After installation, you can reduce the amount of disk space required by deleting some files (see Section 1.3.3 [Image Files for OS/2 Installation], page 8).

MIT Scheme requires a fair amount of RAM to run well. We haven't tried running this on any machine with less than 36 megabytes, so we don't have any hard data on what the smallest practical amount of RAM is. However, for running Scheme under OS/2 Warp, 8 megabytes is probably the least you should consider, and 12 megabytes is probably comfortable. If you want to use the Scheme compiler or the Edwin text editor, you should have at least 16 megabytes of RAM.

MIT Scheme has been tested only on OS/2 Warp 3.0, but it should also run under OS/2 versions 2.1 and 2.11. It was compiled using IBM Visual Age C++ version 3.0 and the OS/2 Toolkit version 3.0.

1.3.2 OS/2 Installation Procedure

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

'readme' This file.

'bin\' The executable programs 'scheme.exe' and 'bchschem.exe'.

'doc\' Documentation files. Two files, the *MIT Scheme User's Manual* and the *MIT Scheme Reference Manual*, are provided in '.inf' format; these files can be read with the OS/2 'view' command.

'etc\' Installation command files. These are scripts that will be used during the installation procedure.

'icons\' A directory containing some useful icons.

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

Perform the following steps to install Scheme:

1. Run the command file 'etc\mkbands.cmd'. This creates the image files 'compiler.com', 'edwin.com', and 'all.com', and deletes the files 'compdel.com' and 'eddel.com'.
2. Move the executable files 'scheme.exe' and 'bchschem.exe' from 'bin\' to any directory that appears in your 'PATH' environment variable. Depending on your needs, you may want to keep only one of these files; each of these files is about 450 kilobytes and chances are you'll only be using one of them. If you keep only 'bchschem.exe' we recommend you rename it to 'scheme.exe'. Of course, you can also keep both programs around if you think you might use them both.

   The only difference between these two programs is in how they handle garbage collection. 'scheme.exe' allocates two memory heaps, and copies objects between the heaps to preserve them. This means that most of the time the other heap is occupying valuable memory but doesn't hold any interesting data. 'bchschem.exe' allocates only one memory heap, creates a disk file during garbage collection, copies objects into the file, then copies them back into memory.[1]

   These programs provide you with some important performance trade-offs. If you have plenty of memory and want the best performance, use 'scheme.exe'. If you don't have enough memory, or if you want to use less memory and will accept slower performance, use 'bchschem.exe'. One way to tell that you don't have enough memory is to run 'scheme.exe' for a while and see if your machine is paging during garbage collection.

[1] See Section 2.3 [Memory Usage], page 16, for more detail about this topic.

You might consider trying to use 'scheme.exe' and letting the operating system's paging handle the lack of RAM. But usually you will find that using 'bchschem.exe' without paging is much faster than using 'scheme.exe' with paging. Of course, if you are using 'bchschem.exe' and you're still paging, there's nothing you can do to win.

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

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

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

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

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

   ```
   scheme -library d:\scheme\lib
   ```

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

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

5. *Optional:* Run the command file 'etc\mkfolder.cmd' to create a folder containing icons to invoke Scheme. You must give this command file a single argument, which is the full path name of the executable file that you will use. For example, if you stored the file 'scheme.exe' as 'c:\scheme\bin\scheme.exe', then you would invoke this command file as follows:

   ```
   etc\mkfolder c:\scheme\bin\scheme.exe
   ```

   This will create a folder containing four icons, called 'Scheme', 'Edwin', 'Liar', and 'All', which correspond to the image files 'runtime.com', 'edwin.com', 'compiler.com', and 'all.com'. If you decide to delete any of the image files, you can delete the corresponding icons as well.

6. *Optional:* The 'icons\' directory contains some Scheme-ish icon files. If you don't like the icons created by the 'etc\mkfolder' command, you can use these icon files to change their appearance.

7. *Optional:* Consider setting some of other environment variables described below.

1.3.3 Image Files

Scheme stores its runtime environment in a binary *image* file, which is directly loaded into the Scheme process. An image file is a snapshot of Scheme's memory taken at a particular time (you can create your own image files using the `disk-save` procedure; see Section 3.3 [World Images], page 38). This distribution comes with four different image files, each of which contains different mixes of programs. These image files are stored in the '`lib\`' directory.

1. '`runtime.com`' is the basic runtime environment with no other programs loaded. This is the smallest image file. All other image files are supersets of this file. This is the default image file that is used when Scheme starts up, unless it is overridden by a command-line option.
2. '`compiler.com`' contains the basic runtime environment and the Liar native-code compiler. This is the image file that is loaded when Scheme is started with the '`-compiler`' command-line option.
3. '`edwin.com`' contains the basic runtime environment and the Edwin text editor. This is the image file that is loaded when Scheme is started with the '`-edwin`' command-line option.
4. '`all.com`' contains the basic runtime environment, the Liar compiler, and the Edwin editor. This is the largest of the image files. There is no command-line option corresponding to this image file, but the following options can be used to load this file:

   ```
   -large -constant 1600 -band all.com
   ```

You can delete any of these files that you aren't going to use (these image files take up a *lot* of disk space!). However, keep in mind that you need at least one image file to run Scheme at all.

1.3.4 Documentation

Documentation for Scheme is included with this release, in the directory '`doc\`'. It consists of the following files:

- '`user.inf`' is the *MIT Scheme User's Manual*, which describes how to run and use Scheme. It describes all of the environment variables and command-line options, how to run the compiler and editor, etc.
- '`scheme.inf`' is the *MIT Scheme Reference Manual*, which documents the programming language and the runtime library. This is a large and detailed document for writing programs.
- '`console.txt`' describes the procedures that can be used to control the appearance of the console window (i.e. Scheme's main window).

Note: the User's Manual and Reference Manual are available in other forms, specifically: as PostScript files and as GNU Info files. These alternate forms of documentation may be obtained from our FTP site, '`swiss-ftp.ai.mit.edu`'.

1.3.5 Environment Variables

This section documents several of the environment variables that Scheme uses, and gives an example for each one. These are the environment variables that must usually be defined when installing Scheme under OS/2. For complete documentation of all of the environment variables used by Scheme, see Section 2.5 [Environment Variables], page 22.

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

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

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

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

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

`TEMP`
: tells Scheme the name of a directory where it can store temporary files.

```
SET TEMP=C:\TMP
```

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

```
SET HOME=C:\CPH
```

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

```
SET USER=cph
```

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

```
SET SHELL=C:\LOCAL\PROG\4OS2251\4OS2.EXE
```

1.4 Windows Installation

This section describes how to install MIT Scheme on Windows 3.1, Windows 95 and Windows NT. Note that we have tested MIT Scheme only on Windows NT 3.10, Windows 3.1 and Windows

95. It should run under Windows NT 3.51. It was compiled using the Microsoft's MSTOOLS compiler that comes with Windows NT 3.10.

1.4.1 Prerequisites

The Scheme files use about 20 megabytes of disk space when installed. An additional 5 megabytes of disk space is required during installation. After installation, you can reduce the amount of disk space required by deleting some files (see "Image Files" below).

MIT Scheme requires a fair amount of RAM to run well. We haven't tried running this on any machine with less than 32 megabytes, so we don't have any hard data on what the smallest practical amount of RAM is. However, for running Scheme under Windows 3.1, 8 megabytes is probably the least you should consider, and 12 megabytes is probably comfortable. If you want to use the Scheme compiler or the Edwin text editor, you should have at least 16 megabytes of RAM.

MIT Scheme is a 32-bit program. To run it on Windows 3.1 you need to install the Win32s compatibility package from Microsoft. You might already have Win32s on your machine if another 32-bit software package installed it. Win32s is provided in the self-extracting ZIP file '`pw1118.exe`'. Create a temporary directory. Copy '`pw1118.exe`' into the directory, change into the directory and run '`pw1118`'. This will create three files. Follow the instructions in '`readme.txt`'.

1.4.2 Windows Installation Procedure

Create a directory where you would like Scheme to be installed. We suggest '`c:\scheme`'. From now on we will call this directory the *Scheme root directory*. Change to this directory and unpack '`bin.zip`':

```
unzip bin.zip
```

You will have the following files and directories:

'`readme`' These installation instructions.

'`bin\`' The executable programs '`scheme.exe`' and '`bchschem.exe`', and the libraries '`scheme16.dll`', '`scheme31.dll`' and '`scheme32.dll`'.

'doc\' Documentation files. Three files, the *MIT Scheme Users' Manual*, the *MIT Scheme Reference Manual*, and the *Revised^4 Report on the Algorithmic Language Scheme*, are provided in Windows Help ('.hlp') format.

'etc\' Installation command files.

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

If you do not get this directory structure you may be using a version of 'unzip' that does not understand directory trees. In this case use the version of 'unzip' that comes with MIT Scheme, in the same directory as the '.zip' files.

Perform the following steps to install Scheme:

1. *Build the bands.* Make sure that you are in the Scheme root directory (you should be there already). Run the following command files

   ```
   etc\mk_comp.bat
   etc\mk_edwin.bat
   etc\mk_all.bat
   ```

   This will create the image files 'lib\compiler.com', 'lib\edwin.com', and 'lib\all.com'. You may now delete the files 'lib\compdel.com' and 'lib\eddel.com'.

   If you are running Windows 3.1 you cannot run Scheme from the command line. You need to make a folder containing icons to run Scheme, as described below, and then run the following icons in turn: 'Build COMPILER.COM band', 'Build EDWIN.COM band' and 'Build ALL.COM band'.

2. You may move the executable files 'scheme.exe' and 'bchschem.exe' and the three 'scheme*nn*.dll' files from 'bin\' to any directory that appears in your 'PATH', though we suggest that you leave them put.

   Depending on your needs, you may want to keep only one of the 'scheme.exe' or 'bchschem.exe' files; each of these files is about 550 kilobytes and chances are you'll only be using one of them. If you keep only 'bchschem.exe' we recommend you rename it to 'scheme.exe'. Of course, you can also keep both programs around if you think you might use them both.

   The only difference between these two programs is in how they handle garbage collection. 'scheme.exe' allocates two memory heaps, and copies objects between the heaps to preserve them. This means that most of the time the other heap is occupying valuable memory but doesn't hold any interesting data. 'bchschem.exe' allocates only one memory heap, creates a disk file during garbage collection, copies objects into the file, then copies them back into memory.[2]

[2] See Section 2.3 [Memory Usage], page 16, for more detail about this topic.

These programs provide you with some important performance trade-offs. If you have plenty of memory and want the best performance, use '`scheme.exe`'. If you don't have enough memory, or if you want to use less memory and will accept slower performance, use '`bchschem.exe`'. One way to tell that you don't have enough memory is to run '`scheme.exe`' for a while and see if your machine is paging during garbage collection.

You might consider trying to use '`scheme.exe`' and letting the operating system's paging handle the lack of RAM. But usually you will find that using '`bchschem.exe`' without paging is much faster than using '`scheme.exe`' with paging. Of course, if you are using '`bchschem.exe`' and you're still paging, there's nothing you can do to win.

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

   For example, if you decide to store the directory as '`d:\schdata`', you would add the following to '`autoexec.bat`':

   ```
   SET MITSCHEME_LIBRARY_PATH=D:\SCHDATA
   ```

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

   ```
   scheme -library d:\schdata
   ```

   If you always supply a '`-library`' option, it is not necessary to have the environment variable defined. For example, instead of editing '`autoexec.bat`', you might create a '`.bat`' file or a shortcut (icon) to invoke Scheme and pass it the '`-library`' option automatically.

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

5. Create a folder containing shortcuts (icons) to invoke Scheme. Run Scheme and load the file '`etc\pmgrp.scm`'. On Windows 95 and windows NT you can run scheme from the command line:

   ```
   bin\scheme -library lib -load etc\pmgrp
   ```

   On Windows 3.1 you have to use the '`File | Run..`' menu option of the Program Manager or File Manager and enter the full paths:

   ```
   c:\scheme\bin\scheme -library c:\scheme\lib -load c:\scheme\etc\pmgrp.scm
   ```

   This will create a folder called '`MIT Scheme 7.4.2`' containing icons for running Scheme and for reading the help files. The icons called '`Scheme`', '`Edwin`', '`Liar`', and '`All`', correspond to running Scheme with the image files '`runtime.com`', '`edwin.com`', '`compiler.com`', and '`all.com`'. If you decide to delete any of the image files, you can delete the corresponding icons as well.

   There is a bug in Windows 95 that prevents the Edwin shortcut from being created correctly. To get around this problem, we have created the shortcut with an incorrect command line. You have to manually edit the shortcut to change the curly braces on the command line into parentheses. The command line reads

```
... -eval {edit}
```

but should read

```
... -eval (edit)
```

6. *Optional:* Consider setting some of other environment variables described below.
7. The two files '`w32rtbci.zip`' and '`w32edbci.zip`' contain debugging information for the Scheme runtime and Edwin respectively. The total space required for all of the debugging information is about 8 megabytes. The runtime debugging information is the more useful as it allows the debugger to give you source code level information on all of the standard procedures like '`map`' and '`number->string`'.

   Debugging information can be installed by uncompressing the '`w32rtbci.zip`' and '`w32edbci.zip`' files in the '`SRC`' subdirectory of the library directory.

   ```
   cd c:\scheme\lib
   mkdir SRC
   cd SRC
   unzip w32rtbci
   unzip w32rtbci
   ```

   Debugging information files can be installed in another place.
   In this case the the '`MITSCHEME_INF_DIRECTORY`' environment variable should be set in '`autoexec.bat`'.

   To test that you have installed the debugging information correctly, try to pretty-print a compiled procedure, for example,

   ```
   (pp pp)
   ```

   When the debugging information is installed correctly, Scheme will be able to find the source code for the `pp` procedure, and display it. Otherwise it will use the short opaque form, something like this:

   ```
   #[compiled-procedure 13 ("pp" #x2) #xF #x646BF7]
   ```

2 Running Scheme

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

2.1 Basics of Starting Scheme

Usually, MIT Scheme is invoked by typing

```
scheme
```

at your operating system's command interpreter. (Under Windows 3.1 you must use the Program Manager's '`File | Run..`' command, or an icon.) Scheme will load itself, clear the screen, and print something like this:

```
Scheme saved on Monday March 25, 1996 at 2:15:03 PM
  Release 7.4.2
  Microcode 11.151
  Runtime 14.168
```

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. An '`(alpha)`' or '`(beta)`' following the release number indicates that this is a alpha- or beta-test release. '`Microcode`' is the version number for the part of the system that is written in C. '`Runtime`' is the version number for the part of the system that is written in Scheme.

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

If the compiler is supported for your machine, you can invoke it by giving Scheme the '`-compiler`' option:

```
scheme -compiler
```

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

2.2 Customizing Scheme

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

- *Command-line options.* Many parameters, like memory usage and the location of libraries, may be varied by command-line options. See Section 2.4 [Command-Line Options], page 18.
- *Command scripts or batch files.* You might like to write scripts that invoke Scheme with your favorite command-line options. For example, you might not have enough memory to run Edwin or the compiler with its default memory parameters (it will print something like "Not enough memory for this configuration" and halt when started), so you can write a shell script (Unix), '`.bat`' file (Windows), or '`.cmd`' file (OS/2) that will invoke Scheme with the appropriate '`-heap`' and other parameters.
- Scheme supports *init files*: an init file is a file containing Scheme code that is loaded when Scheme is started, immediately after the identification banner, and before the input prompt is printed. This file is stored in your home directory, which is normally specified by the '`HOME`' environment variable. Under Unix, the file is called '`.scheme.init`'; on the PC it is called '`scheme.ini`'.

  In addition, when Edwin starts up, it loads a separate init file from your home directory into the Edwin environment. This file is called '`.edwin`' under Unix, and '`edwin.ini`' on the PC.

  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 18).
- *Environment variables.* Most microcode parameters, and some runtime system and Edwin parameters, can be specified by means of environment variables. See Section 2.5 [Environment Variables], page 22.
- *Icons.* Under OS/2 and Windows it is possible to create icons that invoke Scheme with different parameters.

2.3 Memory Usage

Some of the parameters that can be customized determine how much memory Scheme uses and how that memory is used. This section describes how Scheme's memory is organized and used;

subsequent sections describe command-line options and environment variables that you can use to customize this usage for your needs.

Scheme uses four kinds of memory:

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

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

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

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

```
stack + (constant + 2*heap) + extra
```

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

```
stack + (constant + heap) + extra
```

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

The Scheme expression '`(print-gc-statistics)`' shows how much heap and constant space is available (see Section 3.4 [Garbage Collection], page 39).

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. (At present, any arguments other than these options will generate a warning message when Scheme starts. In the future, there will be an advertised mechanism by which the extra arguments can be handled by user code.)

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

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

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

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

```
MITSCHEME_LARGE_HEAP
MITSCHEME_LARGE_CONSTANT
MITSCHEME_LARGE_STACK
```

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

```
MITSCHEME_SMALL_HEAP
MITSCHEME_SMALL_CONSTANT
MITSCHEME_SMALL_STACK
```

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

Note: the Scheme expression '`(print-gc-statistics)`' shows how much heap and constant space is available and in use (see Section 3.4 [Garbage Collection], page 39).

`-heap` *blocks*

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

`-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 the C shell in Unix, the following will run Scheme as a background job, redirecting its input and output to files, and preventing it from being killed by keyboard interrupts or by logging out:

```
scheme < /usr/cph/foo.in >& /usr/cph/foo.out &
```

This option is ignored under non-Unix operating systems.

`-nocore` Specifies that Scheme should not generate a core dump under any circumstances. If this option is not given, and Scheme terminates abnormally, you will be prompted to decide whether a core dump should be generated.

This option is ignored under non-Unix operating systems.

`-library` *path*
Sets the library search path to *path*. This is a list of directories that is searched to find various library files, such as bands. If this option is not given, the value of the environment variable '`MITSCHEME_LIBRARY_PATH`' is used; if that isn't defined, the default is used.

On Unix, the elements of the list are separated by colons, and the default value is '`/usr/local/lib/mit-scheme`'. On PCs, the elements of the list are separated by semicolons, and the default value is '`c:\scheme`'.

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

'`-utab`' is an alternate name for the '`-utabmd`' option; at most one of these options may be given.

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

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

`-gc-directory` *directory*
Specifies that *directory* should be used to create files for garbage collection. If the option is not given, the value of environment variable '`MITSCHEME_GC_DIRECTORY`' is used instead, and if that is not defined, '`/tmp`' is used.

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

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

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

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

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

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

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

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

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

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

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

`-no-init-file`
: This option causes Scheme to ignore the '`~/.scheme.init`' or '`scheme.ini`' file, normally loaded automatically when Scheme starts (if it exists).

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

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

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

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

2.5 Environment Variables

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

---

[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.

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

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

2.5.1 Environment Variables for the Microcode

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

'`MITSCHEME_BAND`' (default: '`runtime.com`' on the library path)
: The initial band to be loaded. Overridden by '`-band`', '`-compiler`', or '`-edwin`'.

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

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

`MITSCHEME_LARGE_CONSTANT` (default: '`1000`')
: The size of constant space, in 1024-word blocks, if the '`-large`', '`-compiler`', or '`-edwin`' options are given. Overridden by '`-constant`'. Note: default depends on machine; on most machines it is larger than '`1000`'.

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

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

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

`MITSCHEME_SMALL_CONSTANT` (default: '`450`')
: The size of constant space, in 1024-word blocks, if the size options are not given. Overridden by '`-constant`', '`-large`', '`-compiler`', or '`-edwin`'. Note: default depends on machine; on most machines it is larger than '`450`'.

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

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

`MITSCHEME_UTABMD_FILE` (default: 'utabmd.bin' in the library path)
: The file containing the microcode tables. Overridden by '-utabmd' and '-utab'. It is only necessary when re-building 'runtime.com'.

2.5.2 Environment Variables for 'bchscheme'

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

`MITSCHEME_GC_DIRECTORY` (default: '/tmp')
: The directory in which GC files are written. Overridden by '-gc-directory'.

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

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

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

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

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

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

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

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

2.5.3 Environment Variables for the Runtime System

These environment variables are referred to by the runtime system.

`HOME`
: Directory in which to look for init files. E.g. '`c:\users\joe`' or '`/home/joe`'.

`TEMP`
`TMP`
: Directory for various temporary files. '`TEMP`' is given preference to '`TMP`'.

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

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

2.5.4 Environment Variables for Edwin

These environment variables are referred to by Edwin.

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

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

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

`ESHELL`
: Unix and OS/2 only. Filename of the shell program to use in shell buffers. If not defined, '`SHELL`' is used instead.

`SHELL` (default: '`/bin/sh`' (Unix), '`cmd.exe`' (OS/2))
: Unix and OS/2 only. Filename of the shell program to use in shell buffers and when executing shell commands. Used to initialize the `shell-path-name` editor variable.

`PATH`
: Unix and OS/2 only. Used to initialize the `exec-path` editor variable, which is subsequently used for finding programs to be run as subprocesses.

`DISPLAY`
: Unix running X11 only. Specifies the display on which Edwin will create windows.

`TERM`
: Unix terminals only. Terminal type.

`LINES` (default: auto-sense)
: Unix terminals only. Number of text lines on the screen, for systems that don't support '`TIOCGWINSZ`'.

`COLUMNS` (default: auto-sense)
: Unix terminals only. Number of text columns on the screen, for systems that don't support '`TIOCGWINSZ`'.

2.5.5 Environment Variables for Windows

These environment variables are specific to the Windows implementation.

`MITSCHEME_FONT` (default: determined by operating system)
: A string specifying a font name and characteristics, for example, '`Courier New 16 bold`'. Allowed characteristics are *integer*, specifiying the font size, and the following style modifiers: '`bold`', '`italic`', '`regular`', '`underline`' and '`strikeout`'. Note that the size is the total height in pixels, and **not** the point size. Thus the fonts tend to be 20-50% smaller than you might expect. You should specifiy only fixed-width fonts as variable width fonts are not drawn correctly.

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

`MITSCHEME_GEOMETRY` (default: '`-1,-1,-1,-1`')
: Four integers separated by commas or spaces that specify the placement and size of the MIT Scheme window as a *left,top,width,height* quadruple. The units are screen pixels, and '`-1`' means allow the system to choose this parameter. E.g. '`-1,-1,500,300`' places a 500 by 300 pixel window at some system-determined position on the screen. The width and height include the window border and title.

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

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

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

`USERNAME`
`USER`
: Specifies the login name of the user running Scheme. This is used for several different purposes. '`USERNAME`' is preferred; '`USER`' is used if '`USERNAME`' is not defined. If neither of these variables is defined, `"nouser"` is used as the user's name.

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

2.5.6 Environment Variables for OS/2

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

`USER`
: Specifies the login name of the user running Scheme. This is used for several different purposes. If this variable is undefined, `"nouser"` is used as the user's name.

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

2.6 Starting Scheme from Windows

The Windows version of MIT Scheme runs as a graphics-based application. Under Windows NT and Windows 95, Scheme can be started from the command line as described at the beginning of this chapter.

Under Windows 3.1, Scheme must be run from the Program Manager or the File Manager. Scheme cannot be run from the command line, because it is not a DOS program (although some people run other shell programs in place of '`command.com`' that can run Windows applications).

Shortcuts (Windows 95) or Program Manager groups and icons (Windows 3.1 and Windows NT) are a convenient way to start Scheme. The rest of this section gives some tips on how to set up shortcuts that run Scheme. If you are unfamiliar with this concept you should read about it in the system help.

- Under Windows NT program manager groups can be *common* or *personal*. When setting up icons in a common group it is important to make the icon properties independent of the vagaries of the environment of the user who is running them. It is often worthwhile doing this under Windows 3.1 and for personal groups too.
- Give the shortcut an accurate *Description*.
- Include absolute pathnames to '`scheme.exe`' and '`bchscheme.exe`' in the icon *Command line*.
- I you specify the '`-library`' command line option then you do not have to worry about the '`MITSCHEME_LIBRARY_PATH`' environment variable.
- Set the shortcut's *Working Directory* to something sensible. On Windows NT you can use '`%HOMEPATH%`' to make Scheme start up in the user's home directory.
- There are several icons available in the Scheme executable — choose one that best represents the options given on the command line.
- Specifying a band that contains Edwin is not sufficient to invoke the editor. You also have to put

  ```
  -eval (edit)
  ```

  at the end of the command line.

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 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 operating system's command interpreter. Scheme remains stopped, and can be continued using the job-control commands of your command interpreter. 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.* These commands include `save-buffers-kill-scheme`, normally bound to *C-x C-c*, and `suspend-scheme`, normally bound to *C-x C-z*. These two commands correspond to the procedures `exit` and `quit`, respectively.
- *Graphical-interface buttons that you can activate.* Under OS/2 and Windows, closing the console window (Scheme's main window) causes Scheme to be terminated. Under any operating system, closing an Edwin window causes that window to go away, and if it is the only Edwin window, it terminates Scheme as well.

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

3 Using Scheme

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

3.1 The Read-Eval-Print Loop

When you first start up Scheme from the command line (i.e. not under Edwin), you will be typing at a program called the *Read-Eval-Print Loop* (abbreviated *REPL*). It displays a prompt at the left hand side of the screen whenever it is waiting for input. You then type an expression (terminating it with *RET*). Scheme evaluates the expression, prints the result, and gives you another prompt.

3.1.1 The Prompt and Level Number

The REPL *prompt* normally has the form

```
1 ]=>
```

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

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

2 error>
```

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

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

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

3.1.2 Interrupting

Scheme has several interrupt keys, which vary depending on the underlying operating system: under Unix, *C-g* and *C-c*; under Windows, *C-b*, *C-x*, *C-u*, *C-g* and *C-c*; under OS/2, *C-b*, *C-x*, *C-u* and *C-g*. 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 systems that support *C-c*, type *C-c ?* for more possibilities. Note that in any given implementation, only a subset of the following keys is available.

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

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

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

```
(cmdl-interrupt/abort-nearest)
```

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

```
(cmdl-interrupt/abort-previous)
```

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

```
(continue)
```

in that REPL at any time.

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

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

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

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

3.1.3 Restarting

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

restart [*k*] procedure+

This procedure selects and invokes a *restart method*. The list of restart methods is different for each REPL; 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*)

Your programs create new environments whenever a procedure is called.

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

ge *environment* procedure+

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

pe procedure+

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

gst *syntax-table* procedure+

In addition to the current environment, each REPL maintains a current *syntax table*. The current syntax table tells the REPL which keywords are used to identify special forms (e.g. `if`, `lambda`). If you write macros, you may want to make your own syntax table, in which case it is useful to be able to make that syntax table be the current one; `gst` allows you to do that.

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. Likewise *syntax-table* is the syntax table to use.

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 39). If *purify?* is given and true, this is done; otherwise it is not.

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

load-noisily? variable+

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

load/default-types variable+

When load is given a pathname without a type, it uses the value of this variable to determine what pathname types to look for and how to load the file. `load/default-types` is a list of associations that maps pathname types (strings) to loader procedures. `load` tries the pathname types in the order that they appear in the list. The initial value of this variable has pathname types in this order:

```
"com" "so" "sl" "bin" "scm"
```

This means that, for example, `(load "foo")` will try to load '`foo.com`' first, and '`foo.scm`' only after looking for and failing to find the other pathname types.

All pathnames are interpreted relative to a working directory, which is initialized when Scheme is started. The working directory can be obtained by calling the procedure `pwd` or modified by calling the procedure `cd`; see the reference manual for details. Files may be loaded when Scheme first starts; see the '`-load`' command-line option for details.

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

Loads the option specified by *symbol*; if already loaded, does nothing. Returns *symbol*; if there is no such option, an error is signalled. However, if *no-error?* is specified and true, no error is signalled in this case, and `#f` is returned.

A number of built-in options are defined:

`compress` Code to compress and uncompress files. Undocumented. Used by the runtime system for compression of compiled-code debugging information.

`format` The `format` procedure. Documented in the Reference Manual.

`hash-table`
: The hash-table data type. Documented in the Reference Manual.

`ordered-vector`
: Code to search and do completion on vectors of ordered elements. Undocumented.

`rb-tree` The red-black tree data type. Documented in the Reference Manual.

`stepper` Code to step through the evaluation of Scheme expressions. Undocumented. Used by the Edwin command *`M-x step-expression`*.

`subprocess`
: Code to run other programs as subprocesses of the Scheme process; implemented under Unix and OS/2 only. Undocumented, but used extensively by Edwin.

`wt-tree` The weight-balanced tree data type. Documented in the 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 in the running process. The file '`runtime.com`' that is loaded by the microcode is just such a band. To make your own band, use the procedure `disk-save`.

disk-save *filename* [*identify*] procedure+

Causes a band to be written to the file specified by *filename*. The optional argument *identify* controls what happens when that band is restored, as follows:

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

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

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

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

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

Note: with the C back end, `disk-save` is not very useful. The reason is that compiled procedures are compiled C code that has been dynamically linked in, and `disk-save` does not save any C procedures. Your system is a C back end system if the following expression does not evaluate to `#f`:

```
(system-library-directory-pathname "shared")
```

Note: when restoring a saved band, the Scheme executable must be configured with a large enough constant space and heap to hold the band's contents. If you attempt to restore a band using the '`-band`' option, and the band is too large, Scheme will write an error message that tells you the appropriate command-line options needed to load that band. If you attempt restore a too-large band using `disk-restore`, Scheme will signal an error, but will not provide the configuration information. In general, the configuration that was used to save a band is sufficiently large to restore it.

3.4 Garbage Collection

This section describes procedures that control garbage collection. See Section 2.3 [Memory Usage], page 16, for a discussion of how MIT Scheme uses memory.

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

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

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

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

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

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

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

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

flush-purification-queue! procedure+

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

print-gc-statistics procedure+

Prints out information about memory allocation and the garbage collector. The information is printed to the current output port. Shows how much space is "in use" and how much is "free", separately for the heap and constant space. The amounts are shown in words, and also in 1024-word blocks; the block figures make it convenient to use these numbers to adjust the arguments given to the '`-heap`' and '`-constant`' command-line options. Following the allocation figures, information about the most recent 8 garbage collections is shown, in the same format as a GC notification.

Note that these numbers are accurate at the time that `print-gc-statistics` is called. In the case of the heap, the "in use" figure shows how much memory has been used since the last garbage collection, and includes all live objects as well as any uncollected garbage that has accumulated since then. The only accurate way to determine the size of live storage is to subtract the value of '`(gc-flip)`' from the size of the heap. The size of the heap can be determined by adding the "in use" and "free" figures reported by `print-gc-statistics`.

```
(print-gc-statistics)
constant in use:   534121 words =   521 blocks +  617 words
constant free:        128 words =     0 blocks +  128 words
heap in use:        34845 words =    34 blocks +   29 words
heap free:         205530 words =   200 blocks +  730 words
GC #1: took: 0.13 (81%) CPU time, 0.15 (1%) real time; free: 207210
```

```
;No value
```

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

Controls whether the user is notified of garbage collections. If *on?* is given and `#f`, notification is enabled; otherwise notification is disabled. By default, 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 percentage CPU time is consistently high (over 20%), you should consider running with a larger heap. A rough rule of thumb to halve the GC overhead is to take the amount of free storage, divide by 1000, and add this figure to the current value used for the '`-heap`' command-line option. Unfortunately there is no way to adjust the heap size without restarting Scheme.

toggle-gc-notification! procedure+

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

4 Compiling Programs

Note: the procedures described in this section are only available when the compiler is loaded, as in the '`compiler.com`' world image. Furthermore, `cf` is only available on machines that support native-code compilation.

4.1 Compilation Procedures

cf *filename* [*destination*] procedure+

This is the program that transforms a source-code file into native-code binary form. If *destination* is not given, as in

```
(cf "foo")
```

`cf` compiles the file '`foo.scm`', producing the file '`foo.com`' (incidentally it will also produce '`foo.bin`', '`foo.bci`', and possibly '`foo.ext`'). If you later evaluate

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

'`foo.com`' will be loaded rather than '`foo.scm`'.

If *destination* is given, it says where the output files should go. If this argument is a directory, they go in that directory, e.g.:

```
(cf "foo" "../bar/")
```

will take '`foo.scm`' and generate the file '`../bar/foo.com`'. If *destination* is not a directory, it is the root name of the output:

```
(cf "foo" "bar")
```

takes '`foo.scm`' and generates '`bar.com`'.

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

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

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

save-uncompressed-files? variable+

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

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

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

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

uncompressed-file-lifetime variable+

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

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

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

sf *filename* [*destination*] procedure+

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

The simplest way to use `sf` is just to say:

```
(sf filename)
```

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

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

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

4.2 Declarations

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

4.2.1 Standard Names

Normally, all files have a line

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

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

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

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

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

4.2.2 In-line Coding

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

integrate *name* ... declaration+

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

integrate-operator *name* ... declaration+

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

integrate-external *filename* declaration+

Causes the compiler to use the top-level integrations provided by *filename*. *filename* should not specify a file type, and the source-code file that it names must have been previously processed by the compiler.

If *filename* is a relative filename (the normal case), it is interpreted as being relative to the file in which the declaration appears. Thus if the declaration appears in file '`/usr/cph/foo.scm`', then the compiler looks for a file called '`/usr/cph/`*filename*`.ext`'.

Note: When the compiler finds top-level integrations, it collects them and outputs them into an auxiliary file with extension '`.ext`'. This '`.ext`' file is what the `integrate-external` declaration refers to.

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

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

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

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

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

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

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

First use the `integrate-operator` declaration:

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

Next use the internal `integrate` declaration:

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

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

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

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

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

Useful tip

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

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

4.2.3 Operator Replacement

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

Declaration:

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

Replacements:

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

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

replace-operator *name ...* declaration+

The syntax of this declaration is

```
(replace-operator
  (name
    (nargs1 value1)
    (nargs2 value2)
    ...))
```

where

- *name* is a symbol.
- *nargs1*, *nargs2* etc. are non-negative integers, or one of the following symbols: `any`, `else` or `otherwise`.
- *value1*, *value2* etc. are simple expressions in one of these forms:

  `'`*constant* A constant.

  variable A variable.

  `(primitive` *primitive-name* [*arity*]`)`
  The primitive procedure named *primitive-name*. The optional element *arity*, a non-negative integer, specifies the number of arguments that the primitive accepts.

  `(global` *var*`)`
  A global variable.

The meanings of these fields are:

- *name* is the name of the operator to be reduced. If is is not shadowed (for example, by a let) then it may be replaced according to the following rules.
- If the operator has *nargsN* arguments then it is replaced with a call to *valueN* with the same arguments.
- If the number of arguments is not listed, and one of the *nargsN* is `any`, `else` or `otherwise`, then the operation is replaced with a call to the corresponding *valueN*. Only one of the *nargsN* may be of this form.
- If the number of arguments is not listed and none of the *nargsN* is `any`, `else` or `otherwise`, then the operation is not replaced.

4.2.4 Operator Reduction

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

Declaration:

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

Replacements:

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

Declaration:

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

Replacements:

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

Declaration:

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

Replacements:

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

Declaration:

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

Replacements:

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

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

Declaration:

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

Replacements:

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

reduce-operator *name ...* declaration+

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

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

where

- *n* and *m* are non-negative integers.
- *name* is a symbol.
- *binop*, *value*, *unop*, and *wrap* are simple expressions in one of these forms:

  `'`*constant* A constant.

  variable A variable.

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

  `(global` *var*`)`
  A global variable.

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

The meaning of these fields is:

- *name* is the name of the n-ary operation to be reduced.
- *binop* is the binary operation into which the n-ary operation is to be reduced.
- The `group` option specifies whether *name* associates to the right or left.
- The `null-value` option specifies a value to use in the following cases:

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

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

`any`
`always` *binop* is used on the "last" argument, and *value* provides the 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 left. 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 you code then we hope that you find the following grab-bag of tips, hints and explanations useful.

4.3.1 Coding style

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
         ...) ; You decide - '() or an error?
```

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 Global variables

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

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

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

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

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

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

ignore-reference-traps *variables* declaration+

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

ignore-assignment-traps *variables* declaration+

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

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

set *name ...* variable-specification

All of the explicitly listed names.

all variable-specification
none variable-specification
free variable-specification
bound variable-specification
assigned variable-specification

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

union *set1 set2* variable-specification
intersection *set1 set2* variable-specification
difference *set1 set2* variable-specification

For example, to ignore reference traps on all the variables except *x*, *y* and any variable that is assigned to

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

4.3.3 Fixnum arithmetic

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

A *fixnum* is an exact integer that is small enough to fit in a machine word. In MIT Scheme, fixnums are typically 24 or 26 bits, depending on the machine; 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. The Scheme Reference Manual lists all the fixnum operations.

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

4.3.4 Flonum arithmetic

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

Flonum consing

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

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

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

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

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

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

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

Flonum vectors

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

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

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

The flonum vector operations are:

flo:vector-cons *n* — procedure+

Create a flonum vector of length *N*. The contents of the vector are arbitrary and might not be valid floating-point numbers. The contents should not be used until initialized.

flo:vector-ref *flonum-vector index* — procedure+
flo:vector-set! *flonum-vector index value* — procedure+
flo:vector-length *flonum-vector* — procedure+

These operations are analogous to the ordinary vector operations.

Examples

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

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

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

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

Common pitfalls

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

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

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

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

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

5 Debugging

This chapter is out of date and will be revised.

This chapter is 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 planned carefully and written well 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 Debugger

There are three debuggers available with MIT Scheme. Two of them require and run under Edwin, and are described in that section of this document (see Chapter 7 [Edwin], page 75). The third is command oriented, does not require Edwin, and is described here.

The *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 8 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.

5.3 Debugging Aids

An important step in debugging is to locate the piece of code from which the error is signalled. The Scheme debugger contains a history examiner and an environment examiner to aid the user in locating a bug.

bkpt *message irritant* special form+

Sets a breakpoint. When the breakpoint is encountered, *message* and *irritant* are typed and a read-eval-print loop is entered in the current environment. 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> (proceed)

bar
;Value: done
```

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

pa *procedure* procedure+

The pp procedure is described in the MIT Scheme Reference Manual. We just note here that, by default, MIT Scheme keeps debugging information, so pp is useful for looking at the source code of a procedure (compiled or interpreted).

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.

```
1 ]=> for-all?
;Value 40: #[compiled-procedure 40 ("boole" #x6) #xC #x20ECB0]

1 ]=> (pa for-all?)
(items predicate)
;No value

1 ]=> (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* [*package/env* [*search-parents?*]] procedure+

Search an environment for bound names containing *string* and print out the matching bound names. If *package/env* 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 *package/env* is specified, and `#t` if *package/env* is not specified.

```
1 ]=> (apropos "search")
#[package 41 (user)]
#[package 33 ()]
list-search-negative
list-search-positive
search-ordered-subvector
search-ordered-vector
vector-binary-search
;No value
```

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-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-both *procedure* procedure+
trace *procedure* procedure+

Equivalent to calling both `trace-entry` and `trace-exit` on *procedure*. `trace` is the same as `trace-both`.

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 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 GNU Emacs Interface

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

Note that this interface is supported under Unix only. We have explored the possibility of making it work under OS/2 and concluded that it is impossible.

To invoke Scheme from Emacs, use *M-x run-scheme*, which is defined when either of the libraries 'scheme' or 'xscheme' is loaded. 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, the run-scheme command exists, but is different from the one described here! In order to get this interface, you must load the 'xscheme' library before executing run-scheme.

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

```
--**-*scheme*: 1 [Evaluator]          (Scheme Interaction: input)------
```

The first field, showing '1' in this example, is the level number.

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

```
[Debugger]
[Where]
```

The *mode* after 'Scheme Interaction' is one of:

'input' Scheme is waiting for input.

'run' Scheme is running an evaluation.

'gc' Scheme is garbage collecting.

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

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

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

ESC C-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 *ESC RET*.

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

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

The following commands provide interrupt capability:

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

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

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

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

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

7 Edwin

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

7.1 Starting Edwin

To use Edwin, start Scheme with a world image containing Edwin (for example by giving the '`-edwin`' command-line option), then call 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.

This procedure is sometimes evaluated from the command line to start Scheme with Edwin running:

```
scheme -edwin -eval (edit)
```

The procedure `edwin` is an alias for `edit`.

inhibit-editor-init-file? variable+

When Edwin is first initialized, it loads your init file (called '`~/.edwin`' under Unix, '`edwin.ini`' on PCs) if you have one. If the Scheme variable `inhibit-editor-init-file?` is true, however, your init file will not be loaded even if it exists. By default, this variable is false.

create-editor *arg* ... procedure+

Initializes Edwin, or reinitializes it if already initialized. `create-editor` is normally invoked automatically by `edit`.

If no *args* 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 *args* are given, they are used to update code`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.

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

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

7.2 Leaving Edwin

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

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

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

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

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

7.3 Last Resorts

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

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

The last full release of MIT Scheme was version 7.3.1 in 1994. The first section summarizes the changes that have occurred since version 7.3.1. Following that are changes from previous versions.

A.1 Changes from Release 7.3 to 7.4

This highlights some of the more important changes to Scheme since the 7.3.1 release. For complete details, see the '`RCS.log`' files in the source code.

Changes to Scheme:

- MIT Scheme is now supported under OS/2.
- Illegally-placed internal definitions now signal an error.
- Stream operations from the upcoming new edition of SICP are implemented.
- The `load-option` procedure is now extensible, allowing users to add their own options to the database.
- Daylight-savings-time information is now included in the decoded time data structure.

Changes to Edwin:

- Cut and paste between Edwin and the system's clipboard (in X11, the "selection" mechanism) is now supported. This support is directly integrated into the kill and yank commands: killing some text (e.g. *C-w*, *C-k*, *C-M-k*) stores that text into the clipboard; yanking text (e.g. *C-y*, *M-y*) inserts text from the clipboard. This support is transparent in that it has no effect on Edwin except when some other program puts text into the clipboard.
- Old RCS support has been replaced with VC ("Version Control") as in Emacs 19. VC is significantly easier to use. At present, VC supports only RCS.
- New command: *M-x step-expression*; steps through the evaluation of a Scheme expression.
- New command: *M-x grep*, as in Emacs.
- RMAIL and Unix mail files are now stored in Unix text format, regardless of the underlying operating system. This allows these files to be more easily transferred between operating systems.
- The editor variable `evaluate-in-inferior-repl` now defaults to `#t`.

- Customizable imaging of characters is now supported. The per-buffer variable `char-image-strings` holds a vector of 256 strings that determine how characters are displayed. Using an approriate vector allows the use of fonts with extra characters, "hex" displays, compound characters for displaying ISO Latin text in plain ASCII, and lots of other tricks.
- Dired's commands are now more like those in Emacs 19.
- Dired mode is no longer case-sensitive.
- Dired now supports marking of multiple files. Many commands, such as *M*, *C*, and *R*, now work on the marked files if there are any.
- New command in Dired: `+` creates a new directory.
- OS/2 and Unix only: Fetching mail from a POP server is now supported, using the '`popclient`' program as a subprocess. Supports three different '`popclient`' password schemes: standard, Debian GNU/Linux, and Kerberos V4. Do *M-x variable-apropos RET pop RET* to see the variables that control the use of the '`popclient`' program.
- OS/2 and Unix only: Compressed files are now fully supported. Edwin can read, write, and append compressed files. Methods are provided to support the '`gzip`' and '`compress`' compression programs, using the suffixes '`.gz`' and '`.Z`', respectively. If Edwin is attempting to open a file that doesn't exist, but it sees what appears to be a compressed version of that file, it opens the compressed file instead.
- OS/2 and Unix only: In Dired, *s* now toggles between alphabetic sorting and time sorting.

Changes Specific to Windows:

- Long filenames are now supported.
- The network '`\\`*host*`\`*directory*' notation is now supported.
- Scheme no longer supports the `^Z` end-of-file convention for files. A `^Z` appearing in a file is treated as an ordinary character, and output files do not have `^Z` placed at their end.
- The runtime system now has better heuristics for determining the user name and home directory.
- The catatonia timeout has been extended to 120 seconds — much more reasonable on slow machines.
- Window text color now defaults to the system default rather than to black.
- The default window font family and size is now selectable by the environment variable '`MITSCHEME_FONT`'.
- `process-time-clock` now returns the process time, rather than the real time, under NT or any variant of Win32 that supports `GetProcessTimes`.
- A bug has been fixed in the directory reader in which the OS handle was never closed. This was manifested under Windows 3.1 as Edwin being unable to do filename completion or file saving after 32 handles were opened but never closed.

Changes Specific to Edwin under Windows:

- File systems with either long or short file names are now supported. Edwin modifies its behavior depending on the file-system type, for example, using Unix-style '`.~`*n*`~`' backup names with long files.
- Multiple Edwin screens are now supported.
- The handling of windows has been significantly improved; it is now faster and less buggy.
- New Edwin commands include *M-x set-font*, *M-x set-default-font*, *M-x set-icon*, *M-x set-foreground-color* and *M-x set-background-color*.

Changes Specific to OS/2 (since the 7.4.1 release):

- The main console window now supports a Copy command. Any text appearing on the window can be copied to the clipboard by marking it with the mouse and invoking the Copy command.
- The main console window now has a menu bar, which provides the following operations: Exit, Copy, Paste, Set font, and About.
- The main console window now has a list of fonts that it tries to use to set the default font, rather than just `"4.System VIO"`. It tries these fonts in order until it finds one. This allows Scheme to be used on systems that don't support the '`System VIO`' font, as is true of some non-US versions of OS/2.
- Closing the main console window using the window Close command now asks whether you want to kill Scheme or just close the window and leave Scheme running; usually you will want the former. Previously, it did the latter without asking.
- All windows created by Scheme now have icons to give them a distinctive appearance when minimized.
- Compilation of the microcode is supported using IBM Visual Age C++, Watcom C/C++, or EMX/GCC.
- Unix file systems are supported specially, e.g. Linux `ext2` or Unix NFS. End-of-line translation is not done for files on such file systems, so that the files are read and written properly.
- The network '`\\`*host*`\`*directory*' notation is now supported.

Changes Specific to Edwin under OS/2:

- The Close command on the window menu of an Edwin window now acts like *C-x C-c* if this is the only Edwin window for this process.
- In Dired, *S* now toggles between showing and hiding system/hidden files; these files are initially hidden.

- Dired has an `M` command to set the attributes of a file. It prompts for an attribute string such as '`+r`' or '`-h`' and makes the corresponding change to the file's attributes.

A.2 Changes from Release 7.2 to 7.3

This is an abbreviated list of the changes that have been incorporated in the 7.3 release since the (partial) 7.2 release.

- Support for MS Windows (3.1 and NT).
- Native compiled-code support for DEC Alpha.
- Compiler can now generate C code. This is used to support the Sun SPARC and IBM RS6000 architectures.
- Support for dynamic loading of compiled C code on several operating systems, including HP-UX, OSF/1, SunOS 4, and AIX.
- New data structures: red/black trees and weight-balanced trees.
- Redesign of hash tables makes them much faster and somewhat more flexible.
- Completely new random-number generator. The interface to the random-number generator has been changed to be like that of Common Lisp.
- Record abstraction now uses a special *record* type to represent records (previously it used vectors). `define-structure` now creates records by default; these records are identical to those created with the record abstraction.
- `call-with-current-continuation` is now properly tail-recursive.
- New command-line switch '`-no-suspend-file`' prevents the generation of '`scheme_suspend`' files.
- Uncompression of compiled-code debugging information is now memoized to save time when doing many accesses to the same information (a common occurrence).
- Constant space now grows automatically when needed.
- New procedures:

```
call-with-values                ;replaces with-values
current-load-pathname
error-output-port
fold-left
fold-right
interaction-i/o-port
notification-output-port
prompt-for-command-char
prompt-for-command-expression
prompt-for-confirmation
```

```
prompt-for-evaluated-expression
prompt-for-expression
standard-unparser-method        ;replaces unparser/standard-method
stream-first
stream-rest
symbol<?
trace-output-port
```

- `string-pad-left` now truncates from the left instead of the right.
- Argument to `make-graphics-device` is now a symbol.
- Each REPL now has its own working directory pathname and pathname defaults. `set-working-directory-pathname!` changes only the value for the current REPL.
- The definitions of the following procedures have been changed to implement more useful behavior:

```
string-capitalized?
string-lower-case?
string-upper-case?
substring-capitalized?
substring-lower-case?
substring-upper-case?
```

- New Edwin features:

  M-x dabbrev-expand
  M-x describe-syntax
  M-x rcs-ci-locked-files
  M-x rcs-diff
  M-x rcs-list-locked-files
  M-x rcs-log
  M-x recover-file
  M-x show-parameter-list
  M-x sort-lines and other sorting commands
  `auto-mode-alist` variable (as in Emacs)
  Encryption/decryption of files in `Dired`
  Undo upgraded to match GNU Emacs 18.57
  Buffers grow/shrink with constant amortized time
  Emacs 19 tags support (multiple tags files, inclusion)

- Edwin compression/uncompression commands changed to use `gzip` instead of `compress`.
- Edwin now has text-properties mechanism that permits implementation of highlighted and mouse-sensitive regions.

A.3 Changes from Release 7.1 to 7.2

This is an abbreviated list of the changes that were incorporated in the (partial) 7.2 release since the 7.1 release.

- Support for MS DOS.
- Native compiled-code support for the Intel i386 and i486 architectures.
- Fixes to MIPS compiled-code support MIPS R3000 with large cache-line sizes.
- Complete redesign of the error system.
- X11 graphics interface redesigned. The new design is smarter about interaction with the window manager (for example, it implements the `WM_DELETE_WINDOW` protocol) and has some limited support for event handling. Windows are now closed when they are reclaimed by the garbage collector.
- Redesign of I/O ports. New design supports ports that perform both input and output.
- Ports now have better-defined control over blocking mode and terminal mode. Flushing of output buffers is not done automatically, except for the console port.
- New procedures:

```
*default-pathname-defaults*
->namestring
->truename
close-port
directory-namestring
directory-pathname
directory-pathname-as-file
enough-namestring
enough-pathname
file-access
file-attributes-direct          ;same as file-attributes
file-modification-time-direct
file-modification-time-indirect ;same as file-modification-time
file-namestring
file-pathname
file-readable?
host-namestring
i/o-port?
make-generic-i/o-port
make-i/o-port
open-i/o-file
pathname-simplify
pathname-wild?
pathname=?
port/copy
port/input-channel
port/operation
port/operation-names
port/output-channel
port/state
port?
record-copy
record-modifier                 ;replaces record-updater
```

```
set-port/state!
```

- `with-input-from-file` and `with-output-to-file` no longer close the port when they are exited abnormally.
- Redefined `#!optional` and `#!rest` to be something distinct from symbols.
- Strings now accept "*nnn*" octal character escapes, much like those in C.
- Redesign of pathname abstraction. New design is very similar to the Common Lisp pathname abstraction.
  - ->PATHNAME no longer accepts a symbol as an argument. Only strings and pathnames are valid.
  - Procedures eliminated:

    ```
    canonicalize-input-filename
    canonicalize-input-pathname
    canonicalize-output-filename
    canonicalize-output-pathname
    canonicalize-overwrite-filename
    canonicalize-overwrite-pathname
    home-directory-pathname
    init-file-truename
    pathname->absolute-pathname
    pathname->input-truename
    pathname->output-truename
    pathname->overwrite-truename
    pathname->string
    pathname-components
    pathname-default-host
    pathname-directory-path
    pathname-directory-string
    pathname-name-path
    pathname-name-string
    pathname-new-host
    pathname-relative?
    string->pathname
    symbol->pathname
    ```

  - `pathname-default` no longer accepts a *host* argument.
  - Representation of pathname directories changed to match Common Lisp. Directory is now either `#f` or a list of symbols and strings with first element either `absolute` or `relative`.
  - Unix pathnames now set *device* and *version* to `unspecific`. `unspecific` now means that the field is not used by the operating system.
  - Parsing rules for Unix filenames changed: the file type is the part of the name after the last dot, if any. If the dot occurs at the beginning or end of the filename, then it doesn't count — in that case there is no type. Thus, names like '`.`' and '`..`' have no type.
- New Edwin features:

M-x manual-entry
M-x outline-mode
M-x shell-resync-dirs
M-x shell-command
M-x shell-command-on-region
M-x telnet
RMAIL summary
RMAIL sort
REPL mode and asynchronous evaluation
Printing commands
"run light" for evaluation commands
Reading and writing of compressed files (".Z" suffix)
Reading and writing of encrypted files (".KY" suffix)
Compress/uncompress commands for `Dired`
Support for X protocols: `WM_DELETE_WINDOW` and `WM_TAKE_FOCUS`
Time, load, and mail notification in mode line

Appendix B Comparison of Edwin to Emacs

This section documents known differences between Edwin 3.90 and Emacs 18.57. A reference to a "documented" feature of Emacs means that the feature is documented in the GNU Emacs version 18 manual.

B.1 Incompatibilities

These are differences in design, unlikely to be 'fixed'.

- Edwin's keyboard character set is a superset of Emacs'. Edwin has support for real MIT-style bucky bits. Under X windows and OS/2 (and perhaps other systems) Edwin can distinguish between `C-a` and *`C-A`*, *`C-SPC`* and `C-@`, etc. When running on a terminal, the input set is limited to 8-bit ASCII as in Emacs.
- Edwin is case-insensitive for command and variable names.
- Edwin additionally defines `C-z` and `C-^` as prefix keys. `C-z` is the *`CTRL-META`* prefix, and `C-^` is the *`CTRL`* prefix, both as in TOPS-20 Emacs.
- `C-z` doesn't suspend Edwin. `C-x C-z` suspends both Emacs and Edwin, but isn't defined in the Emacs Manual. Under X, Emacs produces an error message when `C-x C-z` is typed; Edwin suspends itself if it was started from a shell with job control, otherwise it dies.
- *`C-h f`* doesn't work for anything but editor commands, i.e. it only documents editor commands, not ordinary Scheme procedures like `car` or `call-with-current-continuation`.
- *`C-h v`* doesn't work for anything but editor variables, i.e. it only documents editor variables, not ordinary Scheme variables.
- In Edwin, `C-w` and `C-y` don't skip to next occurrence in reverse i-search. This is intentional, and matches the behavior of Emacs 19.
- `comment-indent-hook` is defined differently. In Emacs, it is called with no arguments and with point at the beginning of the comment; in Edwin, it is called with a single argument, the mark at the beginning of the comment, and with point undefined.
- Edwin's completion of Scheme variables is dependent on the environment. In `Scheme` and `Scheme Interaction` modes, *`M-TAB`* completes the Scheme variable name. Completion is environment-dependent, so a prefix may complete differently (or not at all) in two different buffers with different associated environments.
- *`M-x load-file`* and *`M-x load-library`* behave somewhat differently in Edwin.
- The evaluation commands are much different in Edwin.

- Edwin doesn't recognize '`Local Variables:`' at the end of a file; instead it recognizes '`Edwin Variables:`'. This is done because it cannot support the full syntax of Emacs local variables.
- Variable `text-mode-hook` is an event distributor rather than a procedure of no arguments.

B.2 Deficiencies

Deficiencies are shortcomings of Edwin that are likely to be fixed.

- *C-x ^* can be used to grow or shrink the minibuffer window in Emacs but not in Edwin.
- *C-h* in Edwin is a subset of that in Emacs. Missing subcommands:

```
n   view-emacs-news
c-c describe-copying
c-d describe-distribution
c-w describe-warranty
```

- Dired is missing these commands in Edwin: *v* and *B*.
- *C-x n* is not disabled by default.
- Edwin does not have the notion of "novice".

B.3 Missing Subsystems

The following documented subsystems are implemented by Emacs but not by Edwin.

- abbrev mode (edwin has dynamic abbreviations)
- emulation modes: edt, vi, vip
- file locks
- language modes: emacs-lisp, fortran, lisp, modula-2, prolog
- overwrite mode
- picture mode
- slow terminal incremental search
- spelling
- tab stops
- text formatter modes: nroff, scribe, tex, latex

B.4 Missing Commands

These commands are implemented by Emacs but not by Edwin. The commands marked with an asterisk are implemented under Unix and OS/2 but not under Windows. Some of the asterisked commands can work under Windows but the code to implement them is not loaded in order to save space; others are require subprocess support that isn't available in the Windows implementation.

```
  abbrev-mode
  abbrev-prefix-mark
  add-change-log-entry
  add-change-log-entry-other-window
  add-global-abbrev
  add-mode-abbrev
  add-name-to-file
  append-to-buffer
  byte-compile-file
  byte-recompile-directory
  cancel-debug-on-entry
* compile
  convert-mocklisp-buffer
  copy-to-buffer
  debug-on-entry
  define-abbrevs
* delete-matching-lines
* delete-non-matching-lines
  describe-copying
  describe-distribution
  describe-no-warranty
  describe-syntax
  disable-command
  disassemble
  display-time                         (run-notifier is similar)
  dissociated-press
  doctor
  edit-abbrevs
  edit-abbrevs-redefine
  edit-options
  edit-picture
  edit-tab-stops
  edit-tab-stops-note-changes
  edt-emulation-on
  emacs-lisp-mode
  emacs-version
  enable-command
  expand-abbrev
  expand-region-abbrevs
  flush-lines
```

```
  fortran-mode
  global-set-key                      (set-key is similar)
  grep
  hanoi
  indent-c-exp                        c-indent-expression
  insert-abbrevs
  insert-kbd-macro                    write-kbd-macro
  insert-parentheses
  inverse-add-global-abbrev
  inverse-add-mode-abbrev
* keep-lines
  kill-all-abbrevs
  latex-mode                          Note: BAL has one
  lisp-complete-symbol                (scheme-complete-variable is similar)
  lisp-interaction-mode               (inferior-repl-mode is similar)
  lisp-mode
  lisp-send-defun
  list-abbrevs
  list-command-history
  list-options
  list-tags
  local-set-key
* lpr-buffer
* lpr-region
  make-symbolic-link
  make-variable-buffer-local
* manual-entry
  modify-syntax-entry
  move-past-close-and-reindent
  next-error
  next-file
  nroff-mode
* occur
* occur-mode-goto-occurrence
  open-dribble-file
  open-termscript
  overwrite-mode
  plain-tex-mode
  prepend-to-buffer
* print-buffer
* print-region
  read-abbrev-file
  recover-file
  run-lisp
  save-buffers-kill-emacs             save-buffers-kill-scheme
  set-gosmacs-bindings
* shell-command
* shell-command-on-region
```

spell-buffer	
spell-region	
spell-string	
spell-word	
suspend-emacs	suspend-scheme
tab-to-tab-stop	
tags-apropos	
tex-mode	
top-level	
unexpand-abbrev	
vi-mode	
view-buffer	
view-emacs-news	
view-file	
vip-mode	
write-abbrev-file	
yow	
zap-to-char	

B.5 Missing Variables

These documented variables are implemented by Emacs but not Edwin. The variables marked with an asterisk are implemented under Unix and OS/2, but not under Windows.

abbrev-all-caps	
abbrev-file-name	
blink-matching-paren	
blink-matching-paren-distance	
buffer-read-only	
c-tab-always-indent	
comment-start-skip	
* compile-command	
completion-ignore-case	(not documented)
ctl-arrow	
debug-on-error	debug-on-editor-error debug-on-internal-error debug-on-evaluation-error are similar, but more specific
debug-on-quit	
default-directory	
default-major-mode	
echo-keystrokes	
initial-major-mode	Scheme variable initial-buffer-mode
insert-default-directory	
inverse-video	

kill-ring-max	
load-path	
* lpr-switches	
major-mode	
mark-ring	
mark-ring-max	mark-ring-maximum
meta-flag	
no-redraw-on-recenter	
save-abbrevs	
selective-display-ellipses	
* shell-file-name	
tab-stop-list	
tags-file-name	(tags-table-pathname is similar)
track-eol	
visible-bell	

B.6 Per-Buffer Variables

These documented variables are per-buffer in Emacs but not in Edwin.

```
abbrev-mode
auto-fill-hook
buffer-auto-save-file-name
buffer-backed-up
buffer-file-name
buffer-offer-save
buffer-read-only
buffer-saved-size
buffer-undo-list
ctl-arrow
default-directory
local-abbrev-table
major-mode
mark-ring
mode-name
overwrite-mode
selective-display
selective-display-ellipses
shell-prompt-pattern
```

B.7 Edwin Bugs

Incorrect behavior of Edwin that will be fixed:

- Negative argument to *M-u*, *M-l*, and *M-c* seems to work by accident.
- '`-*- foo -*-`' is valid if preceded only by blank lines; in Edwin this works only if it is on a file's the first line.
- Closing paren doesn't check for mismatched delimiters and issue warning message. In c-mode, '`]`' isn't treated specially, and don't flash the matching delimiter.
- *C-u C-x ;* isn't handled right in Edwin.
- *M-x indent-for-comment* shouldn't reindent existing comment in column zero.

Index

*

-

A

B

C

S

T

U

V

W

X

Table of Contents