MIT Scheme User's Manual

Edition 1.21 beta
for Scheme Release 7.3
17 December 1993

by Stephen Adams
Chris Hanson
and the MIT Scheme Team

1 Installation

1.1 Unix

We will use as an example the installation for HP 9000 series 400 computers. The installation for other Unix systems is similar.

MIT Scheme is distributed as a compressed 'tar' file. The tar file contains a single directory, called '`dist-7.3`', 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 hp400.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-7.3
  ```

  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. 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, MIT Scheme's 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, uncomment the definitions of the variables '`infodir`', '`emacslisp`', and '`movemail`' in the installation script. You will also need to edit these definitions so that they correctly indicate where the GNU Emacs files are located.
- Once you are satisfied that the '`install.sh`' script is correct, execute it:

  ```
  ./install.sh
  ```

- After '`install.sh`' finishes, you can delete the '`dist-7.3`' 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 PC Operating Systems

This section describes how to install MIT Scheme on DOS, Windows 3.1, and Windows NT. We would prefer that the Windows version be used, rather than the DOS version, because we intend to maintain the DOS version only so long as it is convenient. For the most part the installation any of these platforms uses the same files, and the procedure is similar. It is possible to install MIT Scheme so that it will run under all three operating systems on one computer, but this does require some care with the configuration of the system.

Note that we have only tested the DOS version on Microsoft DOS 5.0.

1.2.1 System Requirements

MIT Scheme requires at least a 386SX with 8Mb RAM. The bare minimum disk space required is 5.5Mb, which gives you a command line interface for Scheme. We strongly recommend a more advanced environment. To build the Edwin editor band requires an additional 4.3Mb. The whole installation without source code occupies 36Mb of disk.

1.2.2 Manifest

The installation is split into several files according to functional units and the size constraints of a 1.4M high density 3.5" floppy disk. The following files are distributed:

`BIN.ZIP`	Scheme binaries for Windows 3.1/Windows NT
`DOSBIN.ZIP`	Scheme binaries for DOS
`LIB.ZIP`	Smaller files from Scheme library
`RUNTIME.ZIP`	'`runtime.com`' band
`RUNNOFLO.ZIP`	'`runtime.com`' band for DOS machines with no FP hardware
`EDDEL.ZIP`	'`eddel.com`': a kit to build '`edwin.com`' band
`COMPDEL.ZIP`	'`compdel.com`': a kit to build '`compiler.com`' band
`HELP.ZIP`	WinHelp User and Reference Manuals
`BCIRUN1.ZIP`	Debugging information for runtime system
`BCIRUN2.ZIP`	" " " " "
`BCIRUN3.ZIP`	" " " " "
`BCIED1.ZIP`	Debugging information for Edwin
`BCIED2.ZIP`	" " " "
`BCIED3.ZIP`	" " " "
`BCINOFLO.ZIP`	Extra debugging information for machines with no FP hardware
`SRCRUN.ZIP`	Source code for the runtime system
`SRCUC.ZIP`	Source code for the microcode (C)
`SRCED.ZIP`	Source code for Edwin
`SRCCOMP.ZIP`	Source code for i386 compiler
`WIN32S.ZIP`	Win32s installation floppy from Microsoft
`INSTALL.TXT`	These instructions
`UNZIP.EXE`	Program to unpack the '`.zip`' files

Minimal installation on Windows NT requires: '`BIN.ZIP`', '`LIB.ZIP`' and '`RUNTIME.ZIP`'.

For the Edwin editor and the native code compiler add '`EDDEL.ZIP`' and '`COMPDEL.ZIP`' repectively.

Any configuration for Windows 3.1 is the same as for Windows NT except that Win32S also needs to be installed.

For DOS the minimal installation comprises '`DOSBIN.ZIP`', '`LIB.ZIP`' and one of '`RUNTIME.ZIP`' or '`RUNNOFLO.ZIP`'.

1.2.3 PC Installation

These installation instructions describe how to install MIT Scheme on one or more of DOS, Windows 3.1, and Windows NT. If you are installing for DOS and another operating system, you should do the bulk of the installation using the windowing environment.

In each of the following steps the amount of disk space consumed is indicated in square brackets. These sizes do not include the '`.zip`' files which are required only during installation.

1. MIT Scheme under Windows 3.1 requires the Win32s system (version 1.1) to be installed. If you have not previously installed the Win32s system then you should create a floppy disk containing the contents of the '`win32s.zip`' file, e.g:

   ```
   a:
   unzip wherever\win32s.zip
   ```

   Run the '`setup`' command on the floppy disk.

   If you are not sure whether you have Win32s installed, or what version you have installed, try to install it anyway. If you have version 1.0 then the Win32s installation disk will upgrade your Win32s system to version 1.1.

   ```
   a:setup
   ```

2. Decide on where to install MIT Scheme. We suggest the default: '`C:\SCHEME`'. Create the root directory which we from now on refer to as *scheme*. If for example, you choose the default:

   ```
   mkdir c:\scheme
   cd \scheme
   ```

   scheme is the string '`C:\scheme`'.

3. In the *scheme* directory unzip the following essential files: '`bin.zip`', '`lib.zip`' and '`runtime.zip`' [5.5Mb].

   ```
   unzip wherever\bin.zip
   unzip wherever\lib.zip
   unzip wherever\runtime.zip
   ```

   This will create the directory structures '*scheme*`\bin`', '*scheme*`\lib`' and '*scheme*`\etc`', and unpack the essential files. (*Wherever* stands for the place that you have put the '`.zip`' files, which might be another directory or a floppy disk.)

If you have a computer without floating-point hardware (e.g. a 386 machine or a 486SX) and you wish to run the DOS version then you must install the runtime with special floating point support instead of '`runtime.zip`':

```
unzip wherever\runnoflo.zip
```

4. To install the DOS binaries, unzip the '`dosbin.zip`' file.

   ```
   unzip wherever\dosbin.zip
   ```

   This creates a '*scheme*`\dos-bin`' directory containing the DOS versions of the '`.exe`' files. These files are different from the Windows versions, so they are placed in a different directory to allow both versions to co-exist on your computer. It is only the '`.exe`' files that differ between DOS and Windows. The other parts of the MIT Scheme system are shared. The DOS version will run under Windows 3.1 but not under NT. Either running on DOS or Windows 3.1, the DOS version has *no* graphics support

5. If you are installing for Windows 3.1 only, do *one* of the following:
   - Put the following line in the '`autoexec.bat`' file to ensure that the '*scheme*`\bin\win31`' directory is on the path:

     ```
     path %PATH%;scheme\bin\win31
     ```

     This command must be in a '`.bat`' file to work
   - Copy the files from '*scheme*`\bin\win31`' into '*scheme*`\bin`'.

6. If you are installing for Windows NT only, do *one* of the following:
   - As system administrator add '*scheme*`\bin\nt`' to the `PATH` in the system environment (using the Registry Editor, `regedt32`
   - As an individual place `PATH=`*scheme*`\bin\nt` in your environment (use the control panel's system option
   - Copy the files from '*scheme*`\bin\nt`' into '*scheme*`\bin`'.

7. If you are installing for both Windows 3.1 and Windows NT then you must use use environment variables and the `PATH` rather than copying files, i.e. you must arrange for Windows 3.1 to be run with '*scheme*`\bin\win31`' on the path and for Windows NT to be run with '*scheme*`\bin\nt`' on the path. This can be done by putting

   ```
   path %PATH%;scheme\bin\win31
   ```

   in the '`autoexec.bat`' file and adding '*scheme*`\bin\nt`' to the Windows NT system environment path.

8. If you are installing for DOS there is no need to add things to the `PATH`.

9. If you did not choose the default installation directory, make sure that the environment variable `MITSCHEME_LIBRARY_PATH` is defined:

   ```
   set MITSCHEME_LIBRARY_PATH=scheme\lib
   ```

10. Now test the installation so far. Under either Windows system, you should be able to get a Scheme system running in its terminal window by running the following from the Program Manager or the File Manager

scheme`\bin\scheme`

From DOS you should be able to get Scheme running by typing the following at the DOS prompt:

scheme`\dos-bin\scheme`

If there are any problems at this stage, review the installation so far. Remember that you might have to restart your machine to get the effect of any changes that you made in '`autoexec.bat`' or the NT registry.

11. Windows versions only. Now you should create a Program Manager group for MIT Scheme. This can be done by running a Scheme program from the Program Manager using the `File|Run` option:

    scheme`\bin\scheme -load` *scheme*`\etc\pmgrp`

    This program creates a program group called 'MIT Scheme' which contains

    - Icons for running scheme in several configurations
    - Windows Help versions of the Scheme Reference Manual and the MIT Scheme User's Manual.
    - 'Mincer' icons for building the rest of the system

    At this point only the Scheme icon called 'Runtime' will execute. Test it. The other Scheme icons ('Edwin', 'Compiler' and 'MIT Scheme' shield) should report that they cannot find their bands.

12. Windows versions only. To install the help files, unpack '`help.zip`':

    `unzip` *wherever*`\help.zip`

    This installs two windows help files, '`user.hlp`' and '`scheme.hlp`', in '*scheme*`\bin`'.

13. To install the Edwin editor you need to build the '`edwin.com`' band. First unpack the delta file '`eddel.com`' [1.6Mb]:

    `cd` *scheme*
    `unzip` *wherever*`\eddel.zip`

    To build '`edwin.com`' start the 'build EDWIN.COM band' icon, or run the following command:

    scheme`\bin\scheme -large -load` *scheme*`\etc\build -eval (edwin.com)`

    This will load in '`eddel.com`' and create the new band [4.3Mb]. After a successful build the program will exit. The '`edwin.com`' band can be used by both the DOS and Windows versions, so you only need to do this step once, even if you are installing for more than one of DOS, Windows 3.1, and Windows NT.

    If you are installing only for DOS you will have to build '`edwin.com`' from the command line. Be sure to run the DOS '`scheme.exe`' rather than the Windows version:

    scheme`\dos-bin\scheme -large -load` *scheme*`\etc\build -eval (edwin.com)`

14. To install the compiler you need to build the '`compiler.com`' band. First unpack the '`compdel.com`' delta file [2.5Mb]:

```
cd scheme
unzip wherever\compdel.zip
```

To build '`compiler.com`' start the 'build COMPILER.COM band' icon, or run the following command:

```
scheme\bin\scheme -large -load scheme\etc\build -eval (compiler.com)
```

This will load in '`compdel.com`' and create the new '`compiler.com`' band [4.8Mb]. After a successful build the program will exit. As for Edwin, this step needs to be done only once.

If you choose to build this band using the DOS version be sure to run the DOS '`scheme.exe`' rather than the Windows version:

```
scheme\dos-bin\scheme -large -load scheme\etc\build -eval (compiler.com)
```

15. If you want both Edwin and the compiler in one band you should build the '`all.com`' band. First unpack the delta files [4.1Mb unless already unpacked in previous steps]:

    ```
    cd scheme
    unzip wherever\eddel.zip
    unzip wherever\compdel.zip
    ```

    To build '`all.com`' start the 'build ALL.COM band' icon, or run the following command:

    ```
    scheme\bin\scheme -large -load scheme\etc\build -eval (all.com)
    ```

    This will load in both '`eddel.com`' and '`compdel.com`' into the runtime band and create the new band [6.7Mb]. After a successful build the program will exit.

    If you choose to build this band under DOS, be sure to run the DOS '`scheme.exe`' rather than the Windows version:

    ```
    scheme\dos-bin\scheme -large -load scheme\etc\build -eval (all.com)
    ```

16. Any combination of '`edwin.com`', '`compiler.com`' and '`all.com`' may be used. They may be built in any order: it is not necessary to build either of '`edwin.com`' and '`compiler.com`' before building and using '`all.com`'. The bands are shared by all of the supported operating systems so you only have to build the bands once, even if you want to use them from, say, both DOS and Windows 3.1.

    After building the bands you may tidy the MIT Scheme group by removing the mincer icons and recover disk space by deleting the delta files [4.1Mb] and the '`runtime.com`' band [2.3Mb] if you do not need it.

    To create icons that use `bchscheme` instead of `scheme` copy the icons and edit the command lines to change 'scheme' to 'bchschem' (that is right: no tailing 'e').

17. Debugging information can be installed by uncompressing the '`bci*.zip`' files in the Scheme root directory *scheme*. The total space required for all of the debugging information is 7.5Mb. The most useful is the runtime debugging info which is in '`bcirun1.zip`' through '`bcirun3.zip`' [3.6Mb installed].

    If you have installed the DOS version for machines without hardware floating point support then '`bcinoflo.zip`' should be uncompressed also. This must be done *after* the '`bcirun*.zip`' files.

Debugging information files can be installed in the Scheme root directory or in another directory. If another directory is chosen then set the `MITSCHEME_INF_DIRECTORY` environment variable to this directory. '`bcied1.zip`' through '`bcied3.zip`' [3.8Mb installed] hold the debugging information files for Edwin.

2 Release Notes

This chapter describes interesting features and problems for the 7.3 release.

2.1 News

The last full release of the MIT Scheme system was version 7.1.3 in 1991. Since that time, there was a partial release of version 7.2 for i386/i486 machines and some MIPS R3000 machines. This section summarizes the changes that have occurred since version 7.1.3. The changes are divided into two parts: those that were incorporated in the 7.2 partial release, and those that were subsequently incorporated.

2.1.1 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.

2.1.2 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

2.2 C Back-End Release Notes

This release introduces an important new feature: the compiler can generate output in C, which can then be compiled and linked using the usual C development tools. This feature is being used to support architectures, such as the SPARC and IBM RS6000, for which the compiler does not have native-code support.

MIT Scheme systems that have been built using this compiler feature are referred to as *C back-end* systems. This section provides a brief description of the features and known problems for such systems.

2.2.1 General Notes

This section describes the general design of C back-end systems.

- The C back end was written very recently, and we do not use it ourselves for development–we have native back ends for the hardware that we use regularly. Furthermore, most of the system-building tools and changes were put together and tested immediately preceding the release. It is not unlikely that there are serious problems and shortcomings. We'd like to hear about them so we can fix them or bypass them for a future release. If something does not work, do not assume that we know about it. Please report it. The worse that can happen is that we hear about it several times.
- The compiler produces C code that works in the context of the MIT Scheme runtime library. It is not stand-alone code, and there is currently no way to generate stand-alone executables.
- We have done some performance testing on machines where we have both a native back end and the C back end. The C back end code is somewhat larger and slower than the native code. The speed varies depending primarily on the number of inter-C-procedure calls (even within a single source file). In some cases the C code is somewhat faster than the native code, but it is usually noticeably slower. The average is about 2.5 times slower.
- The output code should work on virtually any modern general-purpose architecture with an ANSI-compliant (perhaps even a traditional K&R) C compiler. We've tried it on several 32-bit architectures (HP PA, Sparc, Motorola 68040, RS6000/Power) and a 64-bit architecture (Alpha).

  However, there may very well be assumptions of the following form:

  - Memory is byte-addressable.
  - Addresses and ordinary data live in the same memory. Some special-purpose processors do not satisfy this constraint.
  - `unsigned` data items and addresses fit into the same memory, and if `union`-ed, take the same space.
  - Signed integers are represented using 2s-complement representation.
  - The machine/OS uses the ASCII character set. This is not a property of the compiler per-se, but of the MIT Scheme system.

  In addition, some configuration of the compiler, microcode and build-scripts would have to be done for new architectures/operating systems. The things that come to mind are:

  - Dynamic loading primitives are OS-dependent.
  - Basic microcode configuration, including:
    - OS interface.
    - Choosing a new machine type (FASL numbers in '`config.h`').
  - Build scripts dispatch on the OS and architecture type, and only understand a handful of types.
  - The compiler has a table of C compiler and linker switches for the configurations we have tried. It needs to be extended for new configurations, although this can be done easily.

If you are interested in bringing up the system on a new configuration, please contact us for further instructions, and to obtain the set of crude tools that we use to do this.

- The performance of the output code could be greatly improved by judicious use of assembly language to achieve faster cross-module tail recursion. So far, we've given more weight to portability than to performance.
- The resulting C code can be compiled and statically linked with the MIT Scheme *microcode* (the runtime library is linked in this way), or compiled and dynamically loaded and linked if the operating system supports this feature. The C code must be compiled defining the `COMPILE_FOR_STATIC_LINKING` if it is to be statically linked into the *microcode* or collected into a larger dynamically-loadable library (e.g. edwin or the compiler). If you'd like to put together a shared library of a large subsystem that you have written, please contact us for advice on how it can be done.

2.2.2 Problems with the C Back-End

This section details problems and shortcomings of the system that are specific to the C back-end.

- `compile-procedure` and `compile-scode` do not work. This will be fixed for a future release and is not very hard to bypass in most cases. `compile-procedure` gives an error. `compile-scode` returns something which is not a compiled expression. `cf` and `compile-bin-file` work.
- Dynamically loaded compiled modules are not unloaded when no longer accessed. In particular, after a `disk-restore`, previously loaded modules are still mapped. In addition, at least in some versions of unix, the shared object file cannot be deleted while it is loaded, so recompilation of a module in place will fail if some process has it already loaded.
- Multiple loading does not work. This is bad for development. After loading a compiled module once, it cannot be reloaded (presumably after recompilation).
- `fasdump` does not work for compiled code. It gives an error. Only bands can be dumped.
- Several C compilers are unhappy about the size of the resulting C procedures. They often turn their optimizers off.
- Some C compilers/optimizers produce non-working code for some of the output. In particular, we've had serious problems with the RS6000/AIX-3.2 C compiler.
- Several C compilers give copious "statement not reached" warnings about the resulting code. This arises from the heavy use of macros expanding into the pattern

```
do
{
  body
} while (0)
```

Most of these macros are not really necessary, and the code generator could just issue the *body* directly.

- The mechanism for accreting object files into large dynamically-loadable libraries is currently ad-hoc.

2.3 PC Release Notes

2.3.1 Known Problems in this Beta Release

- The Windows version of MIT Scheme is more robust under NT than Windows 3.1
- [NT] Exclusive file access problems. Edwin sometimes can't write a file that is still open for reading.

  Typically you will have entered `(load "foo.scm")` in the interaction buffer and got an error, say a mis-matched parenthesis. You then switch to the `foo.scm` buffer, fix the problem and then try to save the file. Edwin refuses, saying

  ```
  Unable to open file "your-path\\foo.scm" because: Unknown
  ```

  Work-around: In the interaction buffer: Quit to top level. Do a `(gc-flip)`, which forces the file to be closed when the file port is garbage-collected. Now you will be able to save the file.
- `MITSCHEME_INF_DIRECTORY` sometimes does not work. The runtime system cannot find the debugging information unless it is on the '`C:`' drive. To test if the debugging information is available, try

  ```
  (pp pp)
  ```

  If the debugging information is available the `pp` procedure pretty-prints procedures as Scheme code. If the information cannot be found then it prints the procedure as an opaque object, similar to this:

  ```
  #[compiled-procedure 13 ("pp" #x2) #xF #x646BF7]
  ```
- Edwin autoloads. A few files that are automatically part of Edwin under unix are not part of Edwin in the PC. These can be loaded manually from the auto-load directory, for example, by putting

  ```
  (edwin-load "dabbrev")
  ```

  in your '`edwin.ini`' file.
- DOS version 6. Although we have not tested the DOS version under anythig other than MS-DOS 5.0, we have the following anecdotal evidence:
  - On DOS 6.2 with `emm386` you may have to put the following in '`config.sys`':

    ```
    device=emm386 noems novcpi
    ```

3 Running Scheme

This chapter describes how to run MIT Scheme on a unix system or a PC running DOS, Windows 3.1, or Windows NT. It also describes how you can customize the behavior of MIT Scheme using command-line options and environment variables.

3.1 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 *Run* command, or an icon.) Scheme will load itself, clear the screen, and print something like this:

```
Scheme saved on Thursday December 2, 1993 at 6:18:35 PM
  Release 7.3.0 (beta)
  Microcode 11.146
  Runtime 14.166
```

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 S&ICP 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.

3.2 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.
- *Command scripts or batch files.* You might like to write scripts that invoke Scheme with your favourite command line options. For example, under DOS 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) or a '`.bat`' file (PC) that will invoke Scheme with the appropriate '`-heap`' and other parameters.
- Scheme supports *init files*: if the file '`.scheme.init`' exists in your home directory, it is loaded immediately after the identification banner, and before the input prompt is printed. In addition, when it starts up, edwin loads the file '`.edwin`' from your home directory into the edwin environment. You can use both of these files to define new procedures, commands, or change defaults in the system.

  The `-no-init-file` command line option causes Scheme to ignore the '`.scheme.init`' file (see See Section 3.3 [Command-Line Options], page 19).

  On PC systems these initialization files are called '`scheme.ini`' and '`edwin.ini`' respectively and are searched for in the directory identified by the `HOME` environment variable.
- *Environment variables.* Most microcode parameters, and some runtime system and Edwin parameters, can be specified by means of environment variables. See See Section 3.4 [Environment Variables], page 24.
- *Icons.* Under Windows and Windows NT it is possible to create icons in the Program Manager that invoke Scheme with different parameters.

One of the important parameters that can be customized is how much memory Scheme uses and how that memory is used. Scheme uses four kinds of memory:

- A *stack* which is used for recursive procedure calls,

- A *heap* which is used for dynamically allocated objects, like '`cons`' cells and strings. Storage used for objects in the heap which become unreferenced is eventually reclaimed by *garbage collection*.
- A *constant space*, which 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 from the runtime system.
- An amount of extra storage which is used by the Microcode (the part of the system that is implemented in C).

All aspects except the last may be controlled both by command-line options and by environment variables. MIT Scheme uses a two-space copying garbage collector for reclaiming storage in the heap. There are two version of Scheme which handle garbage collection differently. Ordinary `scheme` has two heaps, one for each 'space'. `bchscheme` 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 may be 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 the 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 your own 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 is sufficient.

The Scheme procedure `(print-gc-statistics)` shows how much heap and constant space is available.

3.3 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 procedure `(print-gc-statistics)` shows how much heap and constant space is available and in use.

`-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. This will prevent 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 only makes sense under unix.

`-nocore`
: Specifies that Scheme should not generate a core dump under any circumstances. Under unix, 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 on PC versions.

`-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*

`-utab` *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 maintainance and development of the MIT Scheme runtime system.

The following options are runtime options. They are processed after the microcode options and after the runtime, Edwin or some other band 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).

`-no-suspend-file`

Under some circumstances Scheme will write out a file called '`scheme_suspend`' in the user's home directory.[1] This file is a "band" 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.

If the `-no-suspend-file` option is given, Scheme will not write a '`scheme_suspend`' file under any circumstances.

`-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.

---

[1] On unix, this file is written when Scheme is terminated by the `SIGUSR1`, `SIGHUP`, or `SIGPWR` signals. On the PC, this file is never written.

`-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.

In addition to the above, `bchscheme` recognises 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-end-position` *number*
: It specifies the last byte position in `-gc-file` at which this invocation of scheme can write. If the option is not given, the value of environment variable `MITSCHEME_GC_END_POSITION` is used instead, and if that is not defined, it is computed from the start position (as provided with -gc-start-position) and the heap size. The area of the file used (and locked if possible) is the region between `-gc-start-position` and `-gc-end-position`.

`-gc-file` *filename*
`-gcfile` *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*
: It specifies the first byte position in `-gc-file` at which this invocation of scheme can write. If the option is not given, the value of environment variable `MITSCHEME_GC_START_POSITION` is used instead, 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-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 only then if the `gcdrone` program is installed in the lib directory.

`-gc-drone` *program*
: Specifies that *program* should be used as the drone program for overlapped I/O during garbage collection. This option is recognized only by `bchscheme`. 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. This option is recognized only by `bchscheme`. 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. This option is recognized only by `bchscheme`. 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.

3.4 Environment Variables

There are many environment variables that Scheme (and Edwin, etc.) look for. Environment variables that affect the microcode must be defined before you start Scheme, but others can be defined or overwritten within Scheme by using the `set-environment-variable!` procedure, e.g.

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

The rest of this section is a summary of the environment variables that are specific to MIT Scheme. The environment variables are organised according to the parts of MIT Scheme that they affect.

3.4.1 Environment Variables Affecting the Microcode

These environment variables are referred to by the microcode (the executable C programs called '`scheme`' and '`bchscheme`').

`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 is somewhat larger on RISC machines.

`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_INF_DIRECTORY`
    Directory containing the debugging information files for the system. It should contain subdirectories corresponding to the subdirectories in the source tree. For example, if its value is '`f:\random`', then runtime system debugging files will be expected in '`f:\random\runtime`', while edwin debugging files will be expected in '`f:\random\edwin`'.

`MITSCHEME_SMALL_CONSTANT` (default: '`400`')
    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 is somewhat larger on RISC machines.

`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`'.

3.4.2 Environment Variables for Bchscheme

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

`MITSCHEME_GC_DIRECTORY` (default: '`/tmp`')
: The directory where to write gc files. Overridden by `-gc-directory`.

`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_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_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 only then 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`.

3.4.3 Environment Variables for the PC

These environment variables are referred to by the Windows, Windows NT, and DOS versions of MIT Scheme.

`MITSCHEME_DPMI_EXT_KBD` (default: '`false`')
: DOS only. Boolean option specifying whether Scheme inserts its own keyboard handling routine when running under DPMI (DOS Protected Mode Interface) or Windows.

`MITSCHEME_X32_EXT_KBD` (default: '`false`')
: DOS only. Boolean option specifying whether Scheme inserts its own keyboard handling routine when *not* running under DPMI or Windows.

`MITSCHEME_TRAP_ALT_TAB` (default: '`false`')
`MITSCHEME_TRAP_ALT_ESCAPE` (default: '`false`')
: Windows and Windows NT only. Boolean option specifying the handling of system command accelerators. These options do not actually work.

`MITSCHEME_GEOMETRY` (default: '`-1,-1,-1,-1`')
: Windows and Windows NT only. 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)
: Windows and Windows NT only. An 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)
: Windows and Windows NT only. A value specifying the window background color. See `MITSCHEME_FOREGROUND`.

3.4.4 Environment Variables Affecting the Runtime System

These environment variables are referred to by the runtime system.

`HOME` Directory where to look for init files. E.g. '`c:\users\joe`' or '`/homes/joe`'.

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

3.4.5 Environment Variables Affecting 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.

`EDWIN_FOREGROUND` (default: none (white))
: DOS version only. ANSI foreground color specifier. Must be a two-digit sequence in the range 30-37. E.g. 32 (green).

`EDWIN_BACKGROUND` (default: none (black))
: DOS version only. ANSI background color specifier. Must be a two-digit sequence in the range 40-47. E.g. 40 (black).

`TERM`
: Terminal type. For DOS, should be '`ansi.sys`' or '`ibm_pc_bios`'. For Windows and Windows NT it should be '`ansi.sys`', which is the default if not set.

`LINES` (default: auto-sense)
: Unix only. Number of text lines on the screen, depending on the terminal type.

`MITSCHEME_LINES` (default: auto-sense or 25)
: DOS only. Number of text lines on the screen, depending on the video adapter and support software. E.g. 43.

`COLUMNS` (default: auto-sense)
: Unix only. Number of text columns on the screen, depending on the terminal type (Unix) or video adapter and support software (DOS). E.g. 132.

`MITSCHEME_COLUMNS` (default: auto-sense, or 80)
: DOS only. Number of text columns on the screen, depending on the video adapter and support software. E.g. 132.

3.5 Starting Scheme from Windows 3.1/NT

There are two distinct versions of MIT Scheme that run on IBM 'compatible' PCs: the DOS version is a character-mode only implementation, which can also run under Windows 3.1 as a DOS application. The Windows version runs as a graphics-based application under Windows 3.1 or Windows NT. The DOS version does not run under Windows NT.

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 only DOS programs can be run from the command line. (This is the case even with WXSERVER as it appears not to work with win32s-based programs). Windows NT overcomes this restriction, but it is still useful to know how to run Scheme from the Program Manager.

Once an icon is set up to run Scheme with some particular command line options, Scheme is run by double-clicking that icon. The rest of this section gives some tips on how to set up icons in the Program Manager that run Scheme. If you are unfamiliar with this concept you should read about it under the help topic of the Program Manager.

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 icons 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 icon an accurate *Description*.
- Include absolute pathnames to `scheme.exe` and `bchscheme.exe` in the icon *Command line* if these executables are not in a directory on the default `PATH`.
- Set the icon's *Working Directory* to: `%HOMEPATH%`
- There are several icons available — 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.

3.6 Leaving Scheme

There are two ways you can leave Scheme. 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 way to leave Scheme is to suspend it; when this is done you may later restart where you left off. Unfortunately this is not possible in all operating systems — currently it works only on unix versions that support job control (i.e. all of the unix versions for which we distribute Scheme), but not on PCs.

Scheme is suspended by evaluating

```
(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 Control-`Z`, but it allows Scheme to respond by typing a prompt when it is unsuspended.)

4 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.

4.1 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.

4.2 Interrupting

Scheme has two interrupt keys under unix: `C-g` and `C-c`. Other systems, like the PC, may have more than two. The PC version has `C-b`, `C-x`, and `C-u` as well as `C-g` and `C-c`. The `C-g` key stops any Scheme evaluation that is running and returns you to the top level REPL. `C-c` prompts you for another character and performs some action based on that character. It is not necessary to type `RET` after `C-g` or `C-c`, nor is it needed after the character that `C-c` will ask you for.

Here are the more common options for `C-c`.

`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.

`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 C-c` Identical to typing `C-g`. If no evaluation is running, this is equivalent to evaluating

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

The options `C-c C-g` and `C-c g`, supplied for compatibility with older implementations, are equivalent to `C-c C-c`.

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

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

The option `C-c x`, supplied for compatibility with older implementations, is equivalent to `C-c C-x`.

On the PC version `C-x` is equivalent to `C-c C-x`.

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

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

The option `C-c u`, supplied for compatibility with older implementations, is equivalent to `C-c C-u`.

On the PC version `C-u` is equivalent to `C-c C-u`.

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

```
(proceed)
```

in that REPL at any time.

The option `C-c b`, supplied for compatibility with older implementations, is equivalent to `C-c C-b`.

On the PC version `C-b` is equivalent to `C-c C-b`.

4.3 Restarting

Another way of exiting 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 this 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 ]=>
```

4.4 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.

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, often you will 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.

5 Debugging

This chapter is out of date and currently under revision.

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 which is impossible, like adding a number to a symbol, or using a variable which 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 10 [Edwin], page 63). 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 `proceed`. 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
```

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 environment associated with it: 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 compound procedure is supplied, `where` lets the user examine the environment of definition of the procedure. This is useful for debugging procedure arguments and values.

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 *proc* procedure+

Causes an informative message to be printed whenever the procedure *proc* 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 *proc* procedure+

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

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

`trace-both` is the same as calling both `trace-entry` and `trace-exit` on *proc*. `trace` is the same as `trace-both`.

untrace-entry [*proc*] procedure+

`untrace-entry` stops tracing the entry of *proc*. If *proc* is not given, the default is to stop tracing the entry of all entry-traced procedures.

untrace-exit [*proc*] procedure+

Stops tracing the exit of *proc*. If *proc* is not included, the default is all exit-traced procedures.

untrace [*proc*] procedure+

Stops tracing both the entry to and the exit from *proc*. If *proc* is not given, the default is all traced procedures.

break-entry *proc* procedure+

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

break-exit *proc* procedure+

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

break-both *proc* procedure+
break *proc* procedure+

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

unbreak [*proc*] procedure+

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

unbreak-entry [*proc*] procedure+

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

unbreak-exit [*proc*] procedure+

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

advise-entry *proc advice* procedure+

General entry-advising procedure. `trace-entry` and `break-entry` are examples of entry-advising procedures. `advise-entry` gives *advice* to *proc*. When *proc* is invoked, *advice* is passed three arguments: *proc*, a list of *proc*'s argument values, and the current environment.

advise-exit *proc advice* procedure+

The general exit-advising procedure. `trace-exit` and `break-exit` are examples of exit-advising procedures. *Advice* is a procedure that should accept four arguments: *proc*, its argument values, the result computed by *proc*, and the current environment. *Advice* is responsible for returning a value on behalf of *proc*. That is, the value returned by *advice* is the value returned by the advised procedure.

advice *proc* procedure+

Returned the advice, if any, given to *proc*.

unadvise-entry [*proc*] procedure+

Removes entry advice from *proc*. If *proc* is not given, the default is all entry-advised procedures.

unadvise-exit [*proc*] procedure+

Removes exit advice from *proc*. If *proc* is not given, the default is all exit-advised procedures.

unadvise [*proc*] procedure+

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

proc procedure+

`*proc*` is a procedure which returns as its value the "broken" procedure. It is used only at a breakpoint set by the procedures `break-exit` and `break-entry` or procedures defined in terms of these procedures (like `break-both` and `break`).

args procedure+

`*args*` is a procedure which returns as its value a list of the arguments supplied to the "broken" procedure. It is used only at a breakpoint set by the procedures `break-exit` and `break-entry` or procedures defined in terms of these procedures (like `break-both` and `break`).

result procedure+

`*result*` is a procedure which returns as its value the result that was about to be returned by the "broken" procedure. It is used only at a breakpoint set by the procedure `break-exit` or procedures defined in terms of this procedure (like `break-both` and `break`).

6 Loading Files

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

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

Filename may be a string naming a file, or a list of strings naming many 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.

`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 from the procedure `pwd` or modified by 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.

7 World Images

A *world image* is a file that contains a complete Scheme system, perhaps additionally including user application code. Scheme provides two methods for saving and restoring world images. The first method writes a file containing all of the Scheme code in the world, which is called a *band*. 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*`)` from a running Scheme, 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. If you need to build a band for a C back-end system, please contact us. 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.

Another method for saving the world is the `dump-world` procedure, which accepts the same arguments as `disk-save` and works in much the same way. However, rather than dumping a band, `dump-world` saves an executable image, which is started just like any other program. This has the advantage of being considerably faster to start on some systems, but the image file is typically much larger than the corresponding band. However, `dump-world` is only supported for a few operating systems, and is not built into the distributed executable files — if you wish to use `dump-world`, you must build your own executable file from the source code. Note that `dump-world` is unlikely to work with this release as MIT Scheme now uses shared libraries.

8 Compiling Files

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

8.1 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 old default). The '`.bif`' file remains after Scheme exits. The time interval and the behavior are controlled by variables. (These variables are not in the global environment; perhaps they should be. They are in the `(runtime compiler-info)` package environment.)

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 microseconds; 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 the 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.

8.2 Declarations

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

8.2.1 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)))
```

8.2.2 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 definition'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 which 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)
```

8.2.3 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.

8.2.4 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.

9 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.

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!

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)` (`xscheme-send-proceed`).

10 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.

10.1 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`' on 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 `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:

`(console)`
: This says to run Edwin 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.

`(x)`
: This says to create an X window and run Edwin on it. This requires the `DISPLAY` environment variable to have been set to the appropriate value before Scheme was started.

`(x` *geometry*`)`
: This is 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.

`(#f)`
: This is the default. It says to try running Edwin on Scheme's console, and failing that, to create an X window and run Edwin on that. This signals an error if neither the console nor the X display is usable.

10.2 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.

10.3 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 auto-save files are named so: under unix, '`foo.scm`' will be saved as '`#foo.scm#`'; on PCs it will be saved as '`foo.s00`', a backup file with version '`00`' (which is never used as an actual version).

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.

10.4 Comparison of Edwin 3.82 to Emacs 18.57

This section documents all known differences between Edwin 3.82 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.

10.4.1 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 perhaps other systems) Edwin can distinguish between `C-a` and `C-a`, `C-SPC` and `C-@`, etc. When running on a terminal or a PC/DOS console, 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.
- `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 environment dependent. 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.

  DOS Note: Since `Alt+TAB` is an MS Windows hot key, you can get the same effect (without disabling it) by typing `Alt+Ctrl+I`.
- `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.

10.4.2 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`. Under DOS it is missing the following:

  `v` view this file.
  `B` byte-compile this file.
  `M` change this file's permissions.
  `G` change this file's group.
  `O` change this file's owner.
  `C` compress this file.
  `U` uncompress this file.
- Dired has the following subcommands that Emacs does not have:

  `k` mark file for copying
  `y` copy marked files
- `C-x n` is not disabled by default.
- Edwin does not have the notion of "novice".

10.4.3 Missing Subsystems

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

- abbrev mode (edwin has dynamic abbreviations
- emulation modes: edt, vi, vi
- file lock
- language modes: emacs-lisp, fortran, lisp, modula-2, prolo
- overwrite mod
- picture mod
- slow terminal incremental searc
- spellin
- tab stop
- text formatter modes: nroff, scribe, tex, latex

DOS Note:

Some modes are available under Unix, but are not included in the standard DOS edwin binary to reduce its size or because they don't work under DOS:

```
Missing Command or mode                   Reason
-----------------------                   ------

* dabbrev                                 reduce size
  compile                                 needs subprocess support
  shell                                   needs subprocess support
                                          + there is a DOS replacement
  techinfo                                specific to MIT & Unix
  telnet                                  needs subprocess support
* midas mode (assembly language editing)  reduce size
* pascal mode                             reduce size
  texinfo                                 reduce size
  man                                     specific to unix
  print                                   does not work yet
* run-notifier                            reduce size
* outline mode                            reduce size
* info                                    reduce size
  rcs                                     needs subprocess support
  sendmail                                needs subprocess support
  malias                                  useless without sendmail
* occur (and list-matching-lines)         reduce size
  rmail                                   needs subprocess support
                                          + specific to Unix ?
  rmailsrt                                useless without rmail
```

Many of the missing modes and commands (compile, shell, rcs, sendmail) require subprocess support. They should be easy to bring up if subprocesses become available, however the code (or even the concept) may be Unix specific. There is a pseudo-shell mode for DOS.

Many of the subsystems listed above have not been tried under DOS. The ones that are known to work are marked with an asterisk (*), although for some of them variables have to be set appropriately before use (e.g. `info-directory` for info).

10.4.4 Missing Commands

These commands are implemented by Emacs but not by Edwin. The commands marked with an asterisk are implemented by the unix version of Edwin but not by the PC version. Some of the asterisked comands can work in the PC version but the code to implement them is not loaded in order to save space; others are unix-specific and are not implemented on the PC.

```
  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
```

10.4.5 Missing Variables

These documented variables are implemented by Emacs but not Edwin. The variables marked with an asterisk are implemented by the unix version of Edwin, but not by the PC version.

`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`	

10.4.6 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
```

10.4.7 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.

Concept Index

Table of Contents