MIT Scheme User's Manual

Edition 0.9
for Scheme Release 7.1
DRAFT: January 1991

by Chris Hanson

1 Running Scheme

This chapter describes how to run MIT Scheme on a unix system. It also describes the command-line options and environment variables that affect how Scheme runs.

1.1 Basics of Starting Scheme

Usually, MIT Scheme is invoked by typing

```
scheme
```

at your operating system's command interpreter. Scheme will load itself, clear the screen, and print something like this:

```
Scheme saved on Monday December 10, 1990 at 9:55:37 PM
  Release 7.1.0 (beta)
  Microcode 11.59
  Runtime 14.104
```

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.

Scheme supports *init files*: if the file '`~/.scheme.init`' exists, it is loaded immediately after the identification banner, and before the input prompt is printed.

1.2 Command-Line Options

Scheme accepts the following command-line options. The options may appear in any order, but they must all 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.)

`-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 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 &
```

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

`-gcfile` *filename*
: Specifies that *filename* should be used for garbage collection. This option is recognized only by `bchscheme`. If the option is not given, a unique filename is generated in the '`/tmp`' directory.

`-library` *path*
: Sets the library search path to *path*. This is a colon-separated 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, '`/usr/local/lib/mit-scheme`' is used.

`-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.

1.3 Environment Variables

This is a summary of the environment variables that are specific to MIT Scheme.

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

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

`MITSCHEME_EDWIN_BAND` (default: '`edwin.com`')
: 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` (default: '`/usr/local/lib/mit-scheme`')
: A colon-separated list of directories. These directories are searched, left to right, to find bands and various other files.

`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`')
: The file containing the microcode tables. Overridden by `-utabmd` and `-utab`.

1.4 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 is known to work on BSD Unix, Ultrix, SunOS, HP-UX (version 6.5 or later). It does NOT work on AT&T

Unix. (Specifically, for unix or POSIX systems, suspension is available if the system supports job control.)

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.

2 The Read-Eval-Print Loop

When you first start up Scheme, 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.

2.1 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:

```
1 ]=> foo

Unbound variable foo

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. 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.

2.2 Interrupting

Scheme has two interrupt keys under unix (other systems may have more than two): `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`.

`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`.

`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`.

2.3 Proceeding

Another way of exiting a REPL is to use the `proceed` procedure:

proceed [*value*] procedure+

There are two ways to use this procedure from an error REPL (usage from other kinds of REPL is not necessarily the same). If *value* is not given, `proceed` retries the expression that caused the error. Unless you have done something to fix the error condition, this will just cause the error to happen all over again. If, for example, you are in an error REPL caused by evaluating an unbound variable, the proper way to use `proceed` is to first define a value for the variable, then to evaluate `(proceed)`. Your program should continue from that point normally.

One caveat: when you get an unbound variable error, the environment for the error REPL is the environment in which you looked up the variable, which is not necessarily the environment in which the variable should be defined. It is best to use the `ge` procedure to guarantee that your definition goes into the right place.

If *value* is given, `proceed` returns it in place of the expression that caused the error. Thus, if you cannot easily fix a particular bug, but you know a correct value for the erring expression, you can continue the computation this way.

2.4 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. When an error occurs, the error REPL that is created is usually initialized so that its current environment is the one in which the error occurred. When it is not possible to supply the error's environment, the following message is printed:

```
There is no environment available;
using the current REPL environment
```

This message tells you that the error REPL is using the same environment as the REPL whose level number is one less than the error REPL's.

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.

3 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+

This variable is a list of strings that are the pathname types to look for, in that order, if `load` is given a pathname that has no type. The initial value of this variable is

```
("com" "bin" "scm")
```

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.

4 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: 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 in to the distributed executable files — if you wish to use `dump-world`, you must build your own executable file from the source code.

5 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.

5.1 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.binf`', 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 '`.binf`' 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 looks at the compiled code from that file, it attempts to find the '`.binf`' file in the same directory from which the '`.com`' file was loaded. Thus it is a good idea to leave these files together.

Another useful thing contained in the '`.binf`' file is the symbol table produced by the assembler. If you have both the '`foo.com`' and '`foo.binf`' files, you can generate an assembly listing in '`foo.lap`' by evaluating:

```
(compiler:write-lap-file "foo")
```

Unfortunately, the '`.binf`' files are somewhat large in the current implementation. If you wish to save space, these files may be deleted; if Scheme tries to find one and cannot, it will still permit debugging, but will be unable to produce much information about your program.

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.

5.2 Declarations

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

5.2.1 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)))
```

5.2.2 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* should 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)
```

5.2.3 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.

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)]
  ))
```

where

- *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.
- *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.

6 GNU Emacs Interface

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

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`).

7 Edwin

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

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+

Enter the Edwin text editor. If entering for the first time, the editor is initialized (by calling `create-editor` with no arguments). Otherwise, the previously-initialized editor is reentered.

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

inhibit-editor-init-file? variable+

When Edwin is first initialized, it loads your init file (called '`~/.edwin`') 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.

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.

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.

Variable, Declaration, and Option Index

Key Index

C

E

Concept Index

Table of Contents