DON'T PANIC

An Introductory Guide to the 6.001 Computer System
Edition 1.2
December 1997

by
Jason A. Wilson
and the Scheme Team

1 The 6.001 Lab and other Scheme Implementations

1.1 Important Information

Please make special note of the following points:

- Don't Turn Off the Lab Computers. The switch is below the yellow flashing light near the floppy disk drive so be careful not to mistake it for the floppy disk eject button.
- Read The Entire Problem Set BEFORE you begin your work. Often, a question you have will be explained in a later section of the problem set.
- Browse the 6.001 Web page at least once a week. It contains course handouts, announcements Scheme software, documentation, advice on where to get help, and other useful information. The page is located at `http://mit.edu/6.001`.
- Read Chapter 3 [Logging In], page 6, for information about logging in for the first time in the 6.001 lab.

We have put this information here in case you decide not to read this entire guide. Generally, this manual is organized on a "learn-as-you-go" basis. Anything that is not essential to doing a problem set is not explained here; however, we will describe how you can find additional information on-line. Although this extra information might not be necessary to complete the problem sets, it will make the problem sets easier and so we urge you to explore.

1.2 Scheme Implementations for 6.001

The 6.001 Lab is reserved for the exclusive use of students in 6.001. You'll find it convenient to work here because the Lab Assistants are available to help you, and you can share the warmth and camaraderie of your classmates while you work on problem sets. If you want to use your own computer, the 6.001 staff provides implementations of Scheme for Linux, Windows NT 4, and Windows 95 (sorry, no Macs). See the course web page for software and installation instructions.

The problem sets are designed so that they run on all of these implementations, and you can move your work between them if you find this useful. For example, you might start problem set at home, then spend some time debugging it in the lab where the Lab Assistants can help you, and then finish things up at home.

6.001 is *not* supported on Athena. The implementation of Scheme on Athena is not compatible with the 6.001 implementations, and we do not guarantee that it can run all the problem sets (and we do guarantee that cannot run some of the problem sets, e.g., PS2).

One way you *can* use Athena is as a repository for your own working files, and for transferring the files between the 6.001 lab and your personal machine. See the course web page for information on how to do this. (Don't use FTP from the 6.001 lab! See the 6.001 web page for information.)

When you work in the lab, you'll normally be storing your files on a floppy disk because there is no permanent storage for students on the 6.001 lab system.

The 6.001 staff is also experimenting with ways for you to get help on problem sets without coming having to come to the lab. Again, see the web page for information.

1.3 Lab Location

The 6.001/.004 lab is located in room 34-501. The easiest way to get there is by the elevator in building 36. Since the door outside the lab has an electronic lock, you will want to bring along the combination, which is given out in lecture. Make sure you also bring your textbook, your floppy disks, this guide, and a copy of the problem set.

1.4 Lab Assistants

Lab Assistants (LAs) are an integral part of the 6.001 lab. They make sure that everything runs smoothly by maintaining hardware and fixing certain problems so that the lab is available to as many students as possible. However, LAs are not in the lab just to maintain hardware; their real purpose is to offer you assistance on problems that you encounter while trying to do the problem sets. If you get stuck, check the blackboard for the help queue status. If the queue is on, write the name of your machine in the next available space. If the queue is off, look for an LA walking around or using a computer to assist you.

Don't think that you must have a major problem to ask for help. Although the LAs are not there to write your code for you, they can assist you when you stop making progress on your problem set. The best way to make sure that all of your questions get answered is to get to the lab early in the week. The night before the problems set is due is not a good time to ask an LA to explain a complicated issue. Instead, come to the lab early or make an office appointment with your recitation instructor or TA if you begin having difficulties. The lab will be very busy the night before the problem set.

If you notice hardware problems, you should contact one of the LAs on duty. If one is not in the room, place a note by the computer that explains what you think the problem is and when you noticed it. If an extreme problem occurs (like a fire) when an LA is not around, contact someone at the equipment desk located outside of the lab.

2 The Scheme of Things

The 6.001 text book describes how to program in Scheme but it gives none of the details of how these programs are entered into the computer, tested and changed. This chapter describes the basic things that happen when you use the computer.

2.1 Flow of Control

Control of the computer begins with you. You can direct the computer by moving the mouse[1] or by typing at the keyboard. These keystrokes and mouse actions are first inspected by a program called the "window manager" to see if you are requesting that it perform some operation. The window manager then passes your keystrokes to the application that has the *focus* (the highlighted window has the focus). Ordinarily, the only application that you see is Edwin, a text editor very similar to Emacs. You will usually want Edwin to have the focus. You can tell the window manager to give Edwin the focus by clicking the mouse on the border surrounding the Edwin window. This should make the border dark grey and allow Edwin to receive your keystrokes.

A text editor is a program that allows you to compose text (programs, memos, letters, books, etc.) much as you would with a typewriter. Unlike a typewriter, Edwin will allow you to modify text that you have already written so that you don't have to type it in all over again. Edwin also allows you to work on more than one piece of text at a time, each in its own *buffer*.

Buffers are not permanent, however, Edwin lets you save buffers in files. If you are working in the lab with a floppy disk, then when you log in Edwin will read all of the files from your floppy disk and place them onto your lab machine's disk so that you can edit them. When you log out, Edwin will automatically checkpoint your floppy disk (store your changes onto the floppy). The files on the computer's hard disk are not permanent and will go away when someone else logs into the machine. To retain these files, Edwin must place them onto your floppy disk. This operation is known as checkpointing. If something goes wrong, talk to an LA who may be able to recover your work from the the hard disk. Whenever you would like to make sure that changes to your files on the hard disk appear on the floppy disk, you may also explicitly request that Edwin do a checkpoint and return to your work.

If you are working in the Lab without a floppy disk and keeping your files on Athena, remember to transfer your modified files back to your Athena directory before logging out.

Once you have written your programs using Edwin, you can instruct Edwin to send them to the Scheme interpreter and record the results. Although almost any buffer can send expressions to the Scheme interpreter, the Scheme interpreter writes its responses only into the '`*scheme*`' buffer. The '`*scheme*`' buffer is the default buffer that Edwin displays when you first log in.

2.2 Special Buffers

Here is a short list of most of the buffers that you will working with:

---

[1] A mouse is a device that controls a small "pointer" (usually shaped like an arrow or an '`x`') on the screen.

'`*scheme*`'
: This is the buffer where you get responses from the Scheme interpreter. Also, you should evaluate test expressions in this buffer.

'`*transcript*`'
: This buffer also gets all the responses from the Scheme interpreter. It differs from the '`*scheme*`' buffer in that you are not meant to type expressions here. Like a "black-box", think of it more as a diary of what has taken place since you have logged in. You copy text out of this buffer to make a transcript. You very rarely want to type things into this buffer.

Problem Set Code Buffers
: These buffers contain the code that makes the problem sets do their magic. Normally they are read-only signifying that you should not change them. If it becomes necessary for you to change the code in one of these buffers, copy only those parts which you need to change into your answer buffer.

Answer Code Buffers
: This is where you spend time writing programs. Normally your answer buffers end with '`.scm`' which tells Edwin that the contents of the buffer is Scheme code. Usually you only need one answer buffer per problem set for code but using additional buffers to store partial transcripts and comments to yourself is very useful. You will also want your answer buffers to correspond to a file on your floppy disk. To do this, remember to save your changes and then checkpoint your disk or log out (which will automatically checkpoint the disk) or else your work will be lost when you leave the lab.

2.3 Standard Conventions

We have already told you enough about communicating with the window manager to get you through 6.001. Obviously, much more will be said about communicating with Edwin and thus the Scheme interpreter.

Normally, pressing an alphanumeric key tells Edwin to insert a character in the current buffer. However, you can press special keys in combination with normal keys to give Edwin commands. The ***Control*** key is located next to the '`a`' key. It works like the shift key meaning that you should hold it down while pressing another key. For example, when we say ***C-x*** we mean: hold down the key labeled ⟨CTRL⟩ while you press '`x`'. The ***Meta*** key also works like the shift key. It is located on either side of the space bar and is labeled ⟨EXTEND CHAR⟩ on the lab machines and ⟨ALT⟩ on most PC keyboards. When we say ***M-x***, we mean hold down the ***Meta*** key while you press '`x`'. Sometimes it is necessary to hold down both ⟨CTRL⟩ and ⟨EXTEND CHAR⟩ at the same time. When we want you to do this, we will say ***C-M-x***.

Some commands take immediate effect while others wait for you to type a response to a question posted in the mini-buffer. At this point, pressing just one more key will sometimes cause the command to take effect. Otherwise, Edwin expects you to type a few words into the mini-buffer and press ***RET*** to tell Edwin when you are done. For more information, see Section 4.2.1 [Mini-buffer], page 8.

Like ***Meta*** and ***Control***, there are other keys that we refer to with special abbreviations.

`TAB` the tab key

`RET` the Return key

`SPC` space or the space-bar

`LFD` line-feed or `C-j`

`ESC` the escape key

3 Logging In and Loading the Problem Set

3.1 Logging In

Here's how to log in at the 6.001 lab:

1. Place your Primary floppy disk in the disk drive. Your primary disk is the one that you will normally use for storing your 6.001 files. It's a (very) good idea to regularly back up your primary disk to another backup disk.
2. Type `u 6 0 0 1 RET` at the "Login:" prompt and then type `u 6 0 0 1 RET` at the "Password:" prompt.
3. Indicate to Edwin that you want to log in normally, by pressing ***RET*** or typing `L` (which is the default).[1]
4. You are done with the log in. Make sure the Primary disk is in the disk whenever you want to checkpoint it or whenever you want to log out or else all of your new work will be lost.

The floppy disks use an MS-DOS file format so you should be able to access your files with a PC, an Athena workstation (using `dosread` and `doswrite`), and some Macintoshes (although we don't support the 6.001 Scheme system for Macs or for Athena). Note that this means that all of your files (and buffers) must have names that contain only alphabetic and numeric characters. Due to DOS restrictions, the file names may have one period in them and you can put up to 8 characters before the period and three characters after it.

3.2 Loading Problem Sets

Type ***M-x load-problem-set***. Edwin prompts you for the problem set number in the mini-buffer. Edwin should also show a default problem set in parentheses. To select the default just type ***RET***. Otherwise, enter the correct problem set number and then hit ***RET***.

If you are working on your own computer, and starting a new problem set, you'll need to download the problem set files to the correct directory on your own machine. After that, ***M-x load-problem-set*** will work as above. You'll find the weekly problem set files, together with instructions for downloading them, on the course web site.

[1] There are other login options for logging in without a disk, and for initializing a disk. The disks you got at the Instrument Room should have been pre-formatted. Reformatting a disk takes about 5 minutes; don't do this unless you really need. And of course, don't reformat a disk that contains files that you need—the files will be destroyed.

4 Editing Your Code

Edwin normally displays the '`*scheme*`' buffer, the mode line, and the mini-buffer when it starts up. The mode line tells you useful information like the name of the buffer above it and whether that buffer is read-only, modified or unmodified. It has the following appearance:

--*ch*-`Edwin:` *buffer* (*major minor*)----*pos*-----------

The italicized fields have the following meaning:

ch is '`**`' if the text has been modified or '`--`' if the buffer has not been modified since last being saved. A read-only buffer contains '`%%`' in this field.

buffer is the name of the buffer.

major is the major mode of the buffer. For example, most buffers contain "Scheme" here.

minor (an optional field) is the minor mode of the buffer. For example, "Fill" appears when a buffer is in `auto-fill` mode.

pos is information about the position of the point in the buffer. This field may be either '`All`','`Top`', '`Bot`', or a percentage.

The '`*scheme*`' buffer and any buffer that ends with '`.scm`' have an additional field, the *run-light*, to tell you whether it is currently evaluating a Scheme expression. For example:

```
--**-Edwin: *scheme*            (REPL: listen)----ALL-----------
```

means that Edwin is ready to evaluate something. Edwin changes `listen` to `eval` when it is busy evaluating an expression. If Edwin has been evaluating for a long time and you would like to stop it, type *`C-c C-c`* to interrupt the Scheme interpreter and get back to "listen" status.

The mini-buffer is the bottom line in the Edwin window. It is used for communicating with Edwin when you request certain commands. For example, when you visit a file, Edwin prompts you for the file's name in the mini-buffer.

The commands you will use most often will deal with editing your code. We will briefly describe those commands in this chapter.

4.1 Loading a File into a Buffer

In order to modify a file or to create a new one, you must first instruct Edwin to "visit" the file. Do this by typing *`C-x C-f`* *file-name*. This will cause Edwin to create a new buffer and place the contents of the file into it. If the file does not exist, Edwin will make an empty buffer and allow you to edit that instead. If you didn't want to start a new file, type *`C-x k RET`* to kill the buffer and try again.

4.2 Switching Between Buffers

Once you have a few buffers in memory, you will want to switch between them quite often. Typing *`C-x b`* *buffer-name* allows you to enter the name of the buffer that you would like to switch to. It will also display a default buffer that you may select by pressing return.

C-x C-b switches you to the '`*buffer list*`' buffer. This buffer isn't meant for editing text. Instead, you use '`*buffer list*`' to select buffers from a list of all the buffers that Edwin knows about. You may use the cursor keys to move to the line of the buffer and press '`f`' to select it.

Other commands have a natural side effect of moving you to a new buffer. For example *C-x k* will kill the current buffer and then place you into the previous buffer. Remember that if you kill a buffer without saving it, all of your changes will be lost. Edwin will warn you if you try to kill an unsaved buffer, and give you a chance to back out of the command. If you really want to kill the buffer without saving your changes, type *yes*.

4.2.1 The Mini-buffer

The mini-buffer is the last line of the Edwin window where Edwin sometimes expects you to type responses to questions that it asks. There are two things that you should know about the mini-buffer. First of all, pressing *C-g* will exit the mini-buffer at any time. Second, pressing ⟨SPC⟩ or ⟨TAB⟩ while in the mini-buffer will generally cause Edwin to try to complete what you have begun to type. If you have typed a unique prefix to a word, then Edwin will complete that word. Otherwise, Edwin will list all of the possible completions to your prefix. You may then type just enough characters to give you the command, file, or buffer that you really want. This feature of Edwin will save you countless keystrokes. To learn how Edwin can complete Scheme procedure and variable names, see Section B.3 [Scheme Completion], page 22.

4.3 Editing A Buffer

4.3.1 What you should and should not edit

In general, you should never change the buffers that contain read-only code given to you in the problem sets, nor should you change the '`*transcript*`' buffer. If you want to change these buffers, you should copy the sections you want into one of your buffers first (see Section 4.3.4 [copying text], page 9). Otherwise, you will end up with tiny changes that no one will be able to find. This will make your code much more difficult to debug.

4.3.2 Inserting Letters and Moving The Point

Once you are in the buffer you want to edit, typing characters causes text to be inserted into the buffer at the *point* (the little reverse-video box), but nothing is overwritten. If you would like to delete a character or two use *Back space* to remove the character to the left of the point and *C-d* to delete to the right of the point. If you want to delete a whole line, use *C-k* to erase all the text right of the point on the current line (*C-y* will bring the text back if you make a mistake).

The cursor motion keys (the keys with triangles on them) allow you to move the point to a new location and begin inserting characters there. There are also commands that allow you to move the point around much faster over larger areas. *C-v* will move the point down an entire screen while *M-v* will move the point up by the same amount. *C-e* will move the point to the end of a line while *C-a* will move it to the beginning. Refer to the on-line documentation for more commands to move the point around.

4.3.3 Undoing Mistakes

If you make a mistake while editing your code, type *C-x u*. Edwin remembers about 8000 characters worth of changes that it will allow you to undo. Consecutive repetitions of *C-x u* will cause Edwin to undo older and older changes. Any command other than an undo breaks the sequence. At this point *C-x u* will allow you to undo your undos, known as redoing an undo.

If you really mess up, it is sometimes desirable to restore the buffer to the way it was the last time you saved it. You can do this by typing *M-x revert-buffer*.

4.3.4 Using the Region to Copy or Move Text

Many Edwin commands operate on a region of text. First you set up the region and then you type a command to operate on it. You begin the region by setting up what is known as the *mark*. To do this, move to the beginning of the text that you want to define as the region and then press *C-SPC*. Next move the point to where you want the region to end and invoke a command. For example, *C-w* will delete the region and place it into what is affectionately called the *kill ring*. You can get this text back by typing *C-y*. Also, you can type *M-w* which will place a copy of the text into the kill ring without deleting it. You can also extract this copy with *C-y*. To find out where the mark is at any time, press *C-x C-x* which swaps the point and the mark. Pressing *C-x C-x* again will swap the point and mark back.

4.4 Saving A Buffer

You should save your answer buffers periodically so that if the computer crashes, you do not lose all your work. This is done by typing *C-x C-s*. If you are working in the lab with a floppy disk then when you log out, saved files are transferred onto your floppy disk replacing the old versions. Remember that if you log out incorrectly and your files are not checkpointed then you will lose your work when someone else logs into the machine.

4.5 Using Dired to Manage Your Files

Generating all these files means that we will need some way to manage them. Edwin provides a command, *M-x dired*, for showing a visual representation of a directory of files. Dired creates a special buffer which you can not edit in the normal way. Instead, you have an easy way to rename files, to mark files for deletion, and to load files into buffers for editing. When you are in the Dired buffer, typing *C-h m* will display all of the commands available there.

To select a file for editing, move the point to the line of a file and then press '`f`'. This is like typing *C-x C-f file-name*. Pressing '`o`' will do nearly the same thing except that the file will be loaded into another text window and the Dired buffer will stay where it is.

To delete individual files, move the cursor to the correct line and press '`d`' and Edwin will place a '`D`' beside the file. You can unmark the file by pressing '`u`' when you are on the line. When you have marked all the files that you are sure you want to be deleted, press '`x`' and answer *yes*. Now when you checkpoint your disk or when you log out, Edwin will remove these files from your floppy drive too. If you accidentally erase a file talk to an LA because sometimes it is possible to recover lost files.

To delete backup copies of files with Dired, press '`~`' and all the backup files will have a D placed beside their names. Now pressing '`x`' will cause Edwin to ask you if you would like to delete these files. Type *yes* and then those files will be deleted. Again, the process isn't complete until you checkpoint your disk or log out.

5 Running and Testing Your Code

5.1 Evaluating Code

There are a few ways to evaluate code from a buffer. One way is to type *M-o*. This command tells Edwin to send the entire buffer to the Scheme interpreter to be evaluated. *M-o* is not defined in the '`*scheme*`' buffer. Another command, *M-z*, tells Edwin to only send the current definition to Scheme in order to be interpreted. A definition in this case is not necessarily an expression such as `(define (foobar baz) ...)`. Instead, Edwin searches backwards until it finds an open parenthesis on the left margin, searches forward to find its mate and evaluates the expression in between. This makes it convenient for all sorts of "top-level" expressions. A third way to evaluate expressions gives more control than either of the first two methods. *C-x C-e* evaluates the expression just before the point regardless of where it is in relation to all the parentheses.

Some people try to remember which definitions they have changed so that they only evaluate those portions of their program. This is often more trouble than it is worth because if you forget to update just a single definition, then a bug that you thought you just fixed still appears and you could be confused for a long time. If you always use *M-o* then your code will be up-to-date with what is written when you test it.

5.2 The '`*scheme*`' buffer

The most sensible place to test your code is in the '`*scheme*`' buffer. Typing a test case (a Scheme expression meant to test portions of your code) and then *M-z* or *C-x C-e* generates a nice listing of trial expressions and their results. If you would like to repeat a test case that you have evaluated already, *M-p* will allow you to cycle through your most recent expressions. Every time you press *M-p*, Edwin shows copies of older and older expressions that have been evaluated in the '`*scheme*`' buffer. When this history runs out, Edwin just starts over at the beginning. It is often the case that you would like to repeat the last evaluated expression with different arguments. There is a right and a wrong way to do this. The right way is to type *M-p*, edit the expression, and then type *C-x C-e*. The wrong way is to use the cursor keys to move up through the Scheme buffer and edit the previous expression in its original context. It is much faster and easier to use the history mechanism, and using it won't leave you with a confusing buffer, unlike direct editing.

5.3 Debugging

When testing your code you will find that often it doesn't work as expected. Sometimes your code will stop dead in its tracks and produce an error message on the screen, sometimes it will return an incorrect result, and sometimes it won't return anything at all because it is caught in an infinite (or at least an unreasonably long) loop. These mistakes in your code are called *bugs* and getting rid of them is an art-form known as *debugging*[1].

The first kind of bug is often the easiest to fix. Sometimes you have just misspelled a variable or procedure name, or have misplaced a parenthesis. Usually your typos are obvious

[1] The Cynic's First Law states that programs don't become more reliable as they are debugged; the bugs just become harder to find.

when the computer points them out to you. At other times the computer screams at you when your program is trying to do an illegal operation (like trying to add two procedures!). In cases like this, it is useful to enter the debugger and see what you can learn about the bug. Questions you should ask yourself are:

- How did my procedure get into this state?
- Do any variables have values that they clearly should not?
- Are all the procedures that I use in this procedure working correctly?
- Do I have the arguments in the correct order?
- Am I using the correct variables?
- Does my procedure have a logical error?
- Did I remember to evaluate all the changes that I made?
- Did I do anything that would cause any old definitions to still be around?
- Did I write this code?

These are useful questions to ask no matter what kind of bugs you have. Remember that even if your procedure seems to work with one or two test cases, you could still have errors in it. Make sure to test the boundary conditions (if you don't your TA will). For example, maybe you forgot to test your absolute value procedure with the number zero. Question: how many fence posts do you need to buy to make 100 feet of fence with a fence post every 10 feet? If you quickly answered 10, then you are especially susceptible to fence-post errors. Otherwise, see how many of your friends will fall for this one.

Only experience can help you become a master debugger. Often the first thing that a beginner does when he or she gets an error message is to type `q` to avoid entering the debugger. This violates a very important rule of debugging: don't throw away information. So when you get an error, go ahead and see what information you can gather.

If the debugger doesn't give you the needed information, sometimes it is useful to put a `display` expression into your code to gather information. Also, you may make your procedures robust so that when they get illegal values, they give you information that the debugger wouldn't give you. These two methods are especially useful with the never-ending-procedure variety of bugs.

A more useful way to gather information is to use the procedure `error`. `error` displays its first argument which should be a string and then displays the rest of its arguments which are any objects of special concern to debugging. Then you are asked if you would like to enter the debugger. Here is an example:

```
(define (cube x)
  (if (not (number? x))
      (error "Argument should be a number instead of" x)
      (* x x x)))
;cube --> #[compound-procedure 28 cube]
;Value: #[undefined-value]

(cube 'hi)
;Argument should be a number instead of hi
;Type D to debug error, Q to quit back to REP loop:
```

One of the best ways of debugging code is to get a lab assistant to help you. Even if they can't immediately find your bug, they can probably tell you whether what you are trying to do is a good idea. Another powerful debugging technique is to completely rewrite some of your code in an improved way. It is often easier to avoid bugs than to find them so use a clear design instead of clever or tricky code that is sure to fail during the next waning crescent. However, if you enjoy debugging code, feel free to make lots of mistakes so that you can find them later.

For more information on using the debugger, see Appendix C [Debugger], page 24.

5.4 Stepper

If you type an expression in '`*scheme*`' buffer and, instead of evaluating it with *C-x C-e*, type *M-s*, Edwin will invoke the Scheme stepper, which permits you to go through step through evaluation of the expression element by element and see the evaluation of each subexpression.

In general, you use the debugger and stepper to home in on bugs from two different "directions." If you have a program that signals an error, you can just let the error occur and use the debugger to try to figure out what happened; or you can step through the program up to the point where you see the error happen and try to figure out what is causing it.

6 Using On-Line Documentation

There are different forms of on-line documentation. Some on-line documentation is meant to teach you skills. For example, the Edwin tutorial is an interactive tutorial useful for when you first begin using Edwin. It is accessed by typing *C-h t*.

Most of the on-line documentation is meant to be used as reference material. This is available through web from the 6.001 home page. The documentation files are included with the 6.001 distributions as HTML files that you can read locally with a web browser if you are using your own computer and are not connected to the network.

You can also access some of this documentation directly through Edwin via the Info system.

6.1 Info

When you first type *M-x info*, you are placed into the directory node which contains a menu of the nodes beneath it. A node is an informational unit: it contains descriptive text and possibly a menu from which you can quickly access other nodes. At this point you can press *h* to get a tutorial on using Info. Also, pressing a question mark will bring up a short list of commands.

The most useful command that you can type is '`m`' followed by the name of a node. This will allow you to access a subnode. For example, typing *m R4RS*, and then *RET* will allow you to visit the on-line documentation for the fromal definition of the Scheme language.

Sometimes a complete node will not fit on one screen. In this case you could use the cursor motion keys to move around the buffer. As an alternative, *SPC* will move forward a whole screen while *Back space* will move you back a whole screen.

If you have typed *m* and selected a menu, you can usually get back to where you were by typing *u*. This command brings you back "up" in the node structure. Also, *d* will bring you to the directory node. *n* and *p* bring you to the next and previous nodes.

6.2 What is Especially Useful

Here is a short summary of the on-line information available by typing *M-x info*

'`Don't Panic`'
: This manual in its on-line incarnation.

'`Scheme Reference Manual`'
: This is the MIT Scheme Reference Manual. It documents the special forms and procedures that are available in MIT Scheme.

'`Scheme User's Manual`'
: This has some details on MIT Scheme and Edwin.

'`R4RS`'
: *The Revised Report on the Algorithmic Language Scheme* is the official (and not very readable) specification of the Scheme language.

'`Emacs`'
: This is the on-line version of the GNU Emacs Manual. Since Edwin and Emacs are so much alike, you may find it very handy. Note that this is available in printed form at the Coop. (This is available on-line in the 6.001 lab only.)

'`Info`' This describes the on-line Info system.

7 Printing and Backing Up Disks

7.1 Printing A Buffer

To print out a buffer type *M-x print-buffer*. Remember to get your print-out right away or it may get lost in a huge heap of paper. The header has the machine name on it.

7.2 Making A Transcript

The easiest way to get a transcript is to copy text from the '`*transcript*`' buffer and accumulate it into another buffer. You should remove the test cases that you do not wish to keep and place your name and other information including the problem set number and the exercise at the top. You might also want to make special comments about some of the test cases. When you are done with the entire problem set, you can print out this buffer. It's a good idea to comment the transcript after each problem, while the details are still in your mind. Incrementally building your final transcript takes less effort than doing it all at the end.

7.3 Printing Out Graphics

To print out graphics in the 6.001 Lab, type *M-x print-graphics*. Edwin will ask you to indicate which window you would like to print. You can either type in a window name (such as `g1`), or simply press ⟨RET⟩. In this last case, Edwin will expect you to click on a desired graphics window with a mouse.

To print graphics using a Windows PC, select the graphics window you want to print and type ⟨Alt⟩-⟨PrintScreen⟩, which copies the active window to the clipboard. You can then paste the clipboard into an appropriate Windows application (Wordpad, for example) to print the result.

7.4 Backing Up Your Primary Disk in the 6.001 Lab

Sometimes floppy disks lose the data that you have stored on them. To counter this problem, you should back up your Primary disk on to your Backup disk from time to time. *M-x checkpoint-floppy* enables you to update your Backup diskette. To do this, follow the steps below.

1. Insert the Backup disk into drive.
2. Type *M-x checkpoint-floppy*.
3. When Edwin is done, put the primary disk back into the drive.
4. You might also want to checkpoint the Primary disk to make sure that it is up to date. To do this at this time, type *M-x checkpoint-floppy*.

If you have done all of these steps, then both your Primary disk and your Backup disk should contain the same files.

8 Logging Out and Giving Feedback

8.1 Logging Out

To logout, make sure your have saved all of your files and then type `M-x logout`. If you have forgotten to save any of your buffers, Edwin will give you a chance to do it now. If you still don't save all of the buffers, Edwin will say "Modified buffers exist. Exit anyway?" If you don't want to save these buffers then answer `yes`. Now Edwin will checkpoint your floppy disk. Edwin will ask if you want to kill Scheme. Type `yes` if you really want to exit Edwin and Scheme and return to the main Login screen. Now you should take the floppy disk out of the floppy drive. If you do leave something behind in the lab and can't find it later, check the lost and found box near the front of the lab.

8.2 Feedback

If you find any bugs, have a complaint, or anything at all, send mail to:

`6001-feedback`

You can send email to the 6001 staff via the address `6001-feedback@sicp-00.mit.edu`. You can also use Edwin to send email to anyone from the 6001 lab, and continue to do your work. Type `C-x m` to get a mail buffer, then fill in the "To:" field. The first time you type `C-x m`, Edwin will also prompt you for an email return address so that someone can respond to your message. If you don't enter your email address here, you will be sending mail anonymously. When you have finished writing the text of your letter, you can send the message by typing `C-c C-c`. If you don't want to send the letter, kill the buffer using `C-x k`.

9 Common Problems and Solutions

If you have any problems that aren't listed here, please pass them along to us so that we can include them in the next printing of this document. For directions on doing this, see Section 8.2 [Feedback], page 17.

9.1 Edwin won't respond to anything I type!

One cause of this is that Edwin is not receiving any keystrokes from the window manager. If this is the case, the border around Edwin will be light grey instead of dark grey. To fix this problem, click the left mouse button when the cursor is in the border of the Edwin window.

9.2 How Can I Reach a "Steady State?"

Often times you will lose and just want to return to some sort of steady state so that you can continue to do your work. If all of a sudden you do something that seems to trash half of your buffer, stop and relax. The first thing that you should try is *`C-x u`*, the undo command. If this fails to do any good, you might try performing a yank using *`C-y`*. If neither of these seems to work, use the write command, *`C-x C-w`* which will allow you to save the buffer under a different name. Now you can load up the old version using *`C-x C-f`*. Next, visually compare these two buffers to see if you can recover any recent changes. You might also want to save your buffers often so that you can always restore to a steady state by doing a *`M-x revert-buffer`*. If your problem is that Edwin does not seem to be evaluating things properly, try typing *`C-c C-c`* a few times. Look at the mode line in the '`*scheme*`' buffer and make sure you see the word "listen" which means that Scheme is ready to accept expressions. For really drastic problems, try saving all of your buffers that don't seem screwed up and logging out. When you log back in, things should be back to normal. If this doesn't help then it is time to talk to an LA.

Appendix A Edwin Major Modes for Scheme

Edwin uses major modes to determine which keyboard commands are available in a buffer. Edwin defines two major modes that facilitate the writing and testing of Scheme programs. The REPL major mode is the default mode of the '`*scheme*`' buffer and Scheme mode is normally used in buffers of Scheme source code. REPL and Scheme mode are very similar to each other and to the Emacs LISP major modes. REPL and Scheme modes include the editing commands from fundamental mode and enhance these with more commands for evaluating Scheme expressions.

A.1 Scheme Mode

The Scheme major mode is specialized for editing Scheme code. It adds indentation and evaluation commands to the normal array of editing commands.

TAB indent the current line for Scheme.

C-M-q indent the next expression.

The following commands evaluate Scheme expressions:

M-ESC Read and evaluate an expression in mini-buffer (`eval-expression`).

M-z Evaluate the current definition (`eval-defun`).

C-x C-e Evaluate the expression preceding point, placing the result in the '`*scheme*`' buffer (`eval-last-sexp`).

M-o Evaluate the buffer (`eval-current-buffer`).

C-M-z Evaluate the current region (`eval-region`).

TAB indents the current line to show the nesting of parentheses. Pressing *TAB* anywhere on a line has the same effect: attractive source files that make parenthesis-balancing clear.

C-M-q is a great way to indent the current definition. Move to the beginning of the current definition (using *C-M-a*) before you use this command.

M-z is the work horse evaluation command in a REPL buffer, but it is also useful in a Scheme buffer. *M-z* looks backwards to find an expression that starts on the left margin and then evaluates it.

For more precise control over what is evaluated, use *C-x C-e*. *C-x C-e* evaluates the expression before the point regardless of where it is in a parenthesized structure. Unlike *M-z*, it is not meant for evaluating just top-level expressions (more importantly, it can evaluate expressions that are not combinations).

M-o evaluates the buffer, which is useful because it insures that all of your changes are evaluated. To make this command even more useful, you should refrain from putting procedure definitions and operations that are meant to test these procedures in the same source file. For example, if you have the fibonacci definition in a file along with `(fib 100000)` then evaluating the entire buffer will take too long. Instead, do all your testing in the '`*scheme*`' buffer. To preserve your tests, you can write them out as a separate file. Once you have fully tested a procedure, you should save the transcript of your successful test cases into a separate buffer and write it to disk. At the end of the problem set, you can collect all of these transcripts into a single buffer and then print it out all at once.

A.2 REPL Mode

REPL is the major mode for communicating with an inferior read-eval-print loop (the '`*scheme*`' buffer is normally in the REPL major mode). All the editing and evaluation commands from Scheme mode are available in addition to the ones listed below.

C-c C-c Stops an evaluation and returns to the top level. Can also be used like '`Q`' when Edwin displays `;Type D to debug error, Q to quit back to REP loop`.

Expressions submitted for evaluation are saved in an expression history. The history may be accessed with the following commands:

M-p Cycle backward through the history.

M-n Cycle forward through the history.

C-c C-r Search backward for a matching string.

C-c C-s Search forward for a matching string.

M-p is the most frequently used history command. Imagine that you have typed `(+ 2 5)` and then *M-z*. Now you wish to evaluate the same expression using a zero instead of the five. Typing *M-p* will recall the previous expression and place it into the current buffer where the point is located. Now you can use the normal editing commands to change the expression into the desired form. When you are done, you can type *M-z* to evaluate your new expression. At this point, typing *M-p* twice would recall the first expression, `(+ 2 5)`, while typing *M-p* just once would recall just the most recent expression, `(+ 2 0)`.

Appendix B Editing Commands

This chapter documents the special features that Edwin shares with Emacs that allow you to edit Scheme code efficiently. These commands are almost always available in any major mode meant for editing.

B.1 Lists and S-expressions

By convention, Edwin keys for dealing with balanced expressions are usually `Control-Meta` characters. They tend to be analogous in function to the corresponding `Control-` and `Meta-` commands.

These commands fall into two classes. Some deal only with *lists* (parenthetical groupings). They see nothing except parentheses, brackets, braces, and escape characters that might be used to quote those.

The other commands deal with expressions or *s-expressions*. The word 's-expression' is derived from *symbolic expression*, the ancient term for any kind of expression in LISP. S-expressions are symbols, numbers, string constants, and lists.

C-M-a Move to the beginning of the current definition (`beginning-of-defun`).

C-M-e Move to the end of the current definition (`end-of-defun`).

C-M-f Move forward over an expression (`forward-sexp`).

C-M-b Move backward over an expression (`backward-sexp`).

C-M-k Kill expression forward (`kill-sexp`).

C-M-u Move up and backward in list structure (`backward-up-list`).

C-M-d Move down and forward in list structure (`down-list`).

C-M-n Move forward over a list (`forward-list`).

C-M-p Move backward over a list (`backward-list`).

C-M-t Transpose expressions (`transpose-sexps`).

C-M-SPC Put mark after following expression (`mark-sexp`).

These commands can be used to find an unbalanced parenthesis quite easily. To find an extra close parenthesis, move to the the beginning of the buffer and type an open parenthesis. Then move backwards one character and type *C-M f*. The point should move to the definition with the extra parenthesis.

B.2 Indentation

⟨TAB⟩ Indent current line "appropriately" for the current mode.

C-j Perform ⟨RET⟩ followed by ⟨TAB⟩ (`newline-and-indent`).

*M-^* Join current line with line above (`delete-indentation`). This cancels out the effect of ⟨LFD⟩.

C-M-o Split line at point; text on the line after point becomes a new line indented to the same column that it now starts in (`split-line`).

M-m Move (forward or back) to the first non-blank character on the current line (`back-to-indentation`).

C-M- Indent several lines to same column (`indent-region`).

C-x ⟨TAB⟩ Shift block of lines rigidly right or left (`indent-rigidly`).

M-x indent-relative
Indent from point to under an indentation point in the previous line.

In a Scheme mode buffer, lines are indented according to their nesting in parentheses. To indent a line, press ⟨TAB⟩. In Scheme or REPL mode, ⟨TAB⟩ aligns the line according to its depth in parentheses. No matter where in the line you are when you type ⟨TAB⟩, it aligns the line as a whole.

B.2.1 Indenting Several Lines

There are several commands that will re-indent several lines of code.

C-M-q Re-indent all the lines within one expression (`indent-sexp`).

C-u ⟨TAB⟩ Shift an entire expression rigidly sideways so that its first line is properly indented.

C-M- Re-indent all lines in the region (`indent-region`).

You can re-indent the contents of a single expression by positioning the point before the beginning of it and typing *C-M-q* (`indent-sexp`). The indentation of the current line is not changed; therefore, only the relative indentation within the list, and not its position, is changed. To correct the position as well, type a ⟨TAB⟩ before the *C-M-q*.

C-M-q is a great way to indent the current definition. Move to the beginning of the current definition using *C-M-a* before you use this command.

If the relative indentation within an expression is correct but the indentation of its beginning is not, go to the line the expression begins on and type *C-u* ⟨TAB⟩. When ⟨TAB⟩ is given a numeric argument, it moves all the lines in the grouping starting on the current line sideways the same amount that the current line moves. It is clever, though, and does not move lines that start inside strings.

Another way to specify the range to be re-indented is with the point and mark. The command *C-M-* (`indent-region`) applies ⟨TAB⟩ to every line whose first character is between the point and mark.

B.3 Completion for Scheme Symbols

Completion maximizes the entropy of your keystrokes. Whenever the computer can figure out what you mean to type, you may tell it to type the rest for you. Usually completion happens in the mini-buffer. But one kind of completion is available in Scheme and REPL buffers: completion for Scheme symbol names.

The command *M-⟨TAB⟩* takes the partial Scheme variable name before point to be an abbreviation, and compares it against all bound variables in the REPL environment. Any additional characters that they all have in common are inserted at the point.

If the partial name in the buffer is not a unique prefix, a list of all possible completions is displayed in another window. At this point you can type enough characters to make the symbol unique and press *M-⟨TAB⟩* again.

C-u M-⟨TAB⟩ works like *M-⟨TAB⟩* except that it also completes symbols.

Appendix C Debugger

When an error occurs in your code, you will be asked whether you would like to enter the debugger. The debugger creates two buffers in which debugging information is presented. The contents of the buffers will change based on the commands you enter.

left-mouse-button
: Select a subproblem or reduction and display information in the description buffer.

C-n

down-arrow
: Move the cursor down the list of subproblems and reductions and display info in the description buffer.

C-p

up-arrow
: Move the cursor up the list of subproblems and reductions and display info in the description buffer.

e
: Show the environment structure.

q
: Quit the debugger, destroying its window.

p
: Invoke the standard restarts.

SPC
: Display info on current item in the description buffer.

?
: Display help information.

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

The description buffer contains three major regions which contain information associated with the selected subproblem or reduction. The first region contains a pretty printed version of the expression. The second region contains a representation of the environment. The variables in the frames are listed along with there values. The bottom of the description buffer contains a region for evaluating expressions in the environment of the selected subproblem or reduction (similar to the '`*scheme*`' buffer). This is the only portion of the buffer where editing is possible. Evaluating expressions here can be useful in gathering information about the circumstances of the bug.

Typing `e` creates a new buffer in which you may browse through the current environment. In this new buffer, you can use the mouse, the arrows, or *C-n* and *C-p* to select lines and view different environments. The environments listed are the same as those in the description buffer. If the selected environment structure is too large to display (if there are more than `environment-package-limit` items in the environment) an appropriate message is displayed. To display the environment in this case, set the `environment-package-limit` variable to *#f*. This process is initiated by the command *M-x set-variable*. You can not use `set!` to set the variable because it is an editor variable and does not exist in the current scheme environment. At the bottom of the new buffer is a region for evaluating expressions similar to that of the description buffer.

Type `q` or to quit the debugger, killing its primary buffer and any others that it has created.

NOTE: The debugger creates description buffers in which debugging information is presented. These buffers are given names beginning with spaces so that they do not appear in the buffer list; they are automatically deleted when you quit the debugger using `q`. If you wish to keep one of these buffers, rename it using `M-x rename-buffer`: once it has been renamed, it will not be deleted automatically.

C.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 the reduction for the following expression:

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

This is because the entire subproblem of the `if` combination 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` combination.

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` combination 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 debugger.

Glossary

Buffer A *buffer* is a block of text that you may examine and change. Whenever you edit in Edwin, you change the contents of a buffer. To preserve a buffer for another time you will use Edwin, you must *save* the buffer to a *file* on disk. Then you must remember to checkpoint your floppy disk.

Commands
A *command* is an instruction you give Edwin through a special key combination. For example, use the `C-x C-f` command to copy a file from a disk into a new buffer.

Cursor The cursor is a solid rectangle visible on the screen whenever the computer is waiting for you to type something. If you type an ordinary character (letter, number, or punctuation), it will be inserted at the current cursor position, and the cursor will move past it.

Extended Commands
An *extended command* is a command used by name, rather than by a special key. To use an extended command, press `M-x`, followed by the name of the command. Your typing will appear at the bottom of the screen, in the mini-buffer (see below). When you've typed the full name of the extended command, press `RET`.

File A *file* is a block of text stored on a disk. A file may be copied from a disk into a buffer. A file is created by saving the contents of a buffer on a disk.

Kills The several most recently-deleted regions are kept in the *kill ring*. Killed text can be retrieved with a *yank*.

Mark The *mark* is an invisible point in a buffer. Many commands set it to the buffer location where they were performed. Each buffer has its own mark. `C-x C-x` exchanges the point and the mark, thus revealing the mark.

Mini-buffer
The *mini-buffer* is that last line of the Edwin window. It is used by Edwin to display informative messages and to prompt you for various inputs, for example, file names, confirmation, etc.

Modes Every buffer has one major mode, and maybe some minor modes. *Major modes* determine special Edwin behavior particular to the language you are using (e.g., Scheme or English). *Minor modes* add special features regardless of the language you are using in the buffer.

Point The current cursor location in each buffer is called the *point*. The cursor is just a reflection of the point in the current buffer.

Region Text between the *point* and the *mark* is called the *region*.

Saving To *save* is to create a disk file from a buffer. The file and the buffer are identical, but only the file remains when you log out; the buffer is erased. However if you don't checkpoint your files onto your floppy disk, your files will be erased when someone else logs into the machine.

Window A *window* has two meanings. When referring just to Edwin, it is a rectangular area on the screen sometimes called a *text window*. When referring to a program in general, it is an area of the screen that contains text or graphics and is manipulated by the window manager. Each text window displays part of some buffer. Because most buffers are too big to fit in one window, only part of the buffer is visible at any time; you can see other parts by using Edwin commands to scroll through the buffer.

Concept Index

S

T

U

V

W

Y

Z

Table of Contents