MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Problem Set 0

Getting Started

Issued: Wednesday, September 3

Due: In tutorial Monday, September 8 or Tuesday, September 9

Reading Assignment: Sections 1.1 of *SICP.*

Introduction

This is a practice problem set intended to familiarize you with the software environment used in 6.001.

1. Setting Up Your Scheme System

If you do not have access to your own computer, the course has about forty workstations in its student laboratory, and about half of the Athena workstations available on campus can also run the course software. So, there's no problem if you don't already have your own computer.

On the other hand, based on surveys from the Spring 1997 term, if you are taking 6.001 you are very likely to

- have easy access to a modern PC running Windows 95,
- be able to connect the PC to the World Wide Web and use a browser like Netscape, and
- prefer to work on this PC, when possible, rather than machines in the 6.001 Lab or the MIT Athena system.

This term we have made a serious effort to accommodate those of you with your own PC's. (At this point, we do not support the Macintosh, even though Scheme interpreters for this platform do exist.)

Whether you're working with your machine or MIT's, the source of all course information is the course web page:

http://www-swiss.ai.mit.edu/~u6001

There you will find links to obtain the course's Windows 95 software, all the course handouts, archives from previous terms (see the General Information handout to learn about these), instructions for obtaining online help from the course Laboratory Assistants, and other useful reference links. You can also obtain the software for the weekly problem sets from the web page.

The 6.001 Lab Assistants (LA's) can be particularly helpful. They are available to give you personal help with some of the maddening puzzles imposed (on us all) by current computer technology. Of course, we hope 6.001 will start you on getting the skills and perspective to improve this technology.

The LA's are 6.001 veterans who know the course material well and are experienced with the 6.001 software systems. They are regularly available in the 6.001 lab. If you have a problem such as finding yourself stuck on a programming problem, being confused about Scheme syntax, or spending undue amounts of time trying to correct small mistakes, you should speak to an LA. You can contact LA's online and expect reasonably prompt response during LA working hours (generally M–F, 5pm to midnight). Send email to `6001-help@mit.edu`, or, on MIT's Athena system, use the online "chat" program called `zephyr`. See the 6.001 homepage link "Getting Help" for more info. We recommend you take the opportunity to work with the LA's in person; LA's give priority to helping students actually present in the 6.001 Lab.

2. Getting started with the 6.001 system

If you go to the 6.001 web page, you should find instructions on how to download a Scheme interpreter so that you can do your problem sets at home.

This first part of the problem set is intended to clarify the resources 6.001 expects you to manage in order to do the programming exercises.

So, beginning at the Web page or the lab, your first task is to get a Scheme system running and to figure out how to type in and evaluate expressions. In this problem set, we are going to assume that you are using an Edwin editor with a Scheme subsystem when we give descriptions of how to evaluate expressions and manage and edit files. If you are not using MIT Scheme's Edwin, (if you are using EdScheme, for example), then our helpful hints will not apply to you.

Evaluating Expressions

The language in which you will be working this term is Scheme, a dialect of Lisp. Scheme programming consists of writing and evaluating *expressions*. The simplest expressions are things like numbers. More complex arithmetic expressions consist of the name of an arithmetic operator (things like $+, -, *$) followed by one or more other expressions, all of this enclosed in parentheses. So the Scheme expression for adding 3 and 4 is `(+ 3 4)` rather than $3 + 4$.[1]

[1]Having the operator come first is called *prefix notation*. Ordinary notation — $3+4$ — is *infix*. There's also *postfix* (also known as *reverse Polish notation*, or *RPN*) where the operator comes last, as with HP calculators.

Exercise 1: Edwin comes with a tutorial. The *Don't Panic* course manual has instructions on how to run the tutorial. The first part of the tutorial says:

```
Edwin commands generally involve the CONTROL key (sometimes labeled
CTRL or CTL) or the META key (sometimes labeled EDIT or ALT).  Rather than
write that in full each time, we'll use the following abbreviations:

 C-<chr>  means hold the CONTROL key while typing the character <chr>
  Thus, C-f would be: hold the CONTROL key and type f.
 M-<chr>  means hold the META or EDIT or ALT key down while typing <chr>.
  If there is no META, EDIT or ALT key, instead press and release the
  ESC key and then type <chr>.  We write <ESC> for the ESC key.
```

We will use the same convention when we want you to type an edwin command.

Perhaps the most important edwin command is C-g. Whenever edwin does something that you do not understand or desire, try hitting C-g several times. It is the general purpose, "stop that!" command.

Note that the bar across the bottom of the window should contain "`*Scheme*`". That means that you are communicating with the Scheme interpreter, rather than using your text editor to do something else, like write a love letter or compose a haiku.

Type in and evaluate each of the expressions listed below. To evaluate an expression, put the cursor just to the right of the expression you want to evaluate, and type **C-x C-e**.

```
25

-37

(- 8 9)

(> 3.7 4.4)

(- (if (> 3 4)
       7
       10)
   (/ 16 10))

; This is a comment.  That is what the semicolon is for.
; The next expression generates an error.
; After evaluating it, use ''Q'' or C-c C-c to get out of the debugger.
(* (- 25 10)
   (+6 3))
```

Observe that some of the examples printed above are indented and displayed over several lines for readability. An expression may be typed on a single line or on several lines; the Scheme interpreter ignores redundant spaces and carriage returns. It is to your advantage to format your work so that you (and others) can read it easily. It is also helpful in detecting errors introduced by incorrectly placed parentheses. For example the two expressions

```
(* 5 (+ 2 (/ 4 2) (/ 8 3)))

(* 5 (+ 2 (/ 4 2)) (/ 8 3))
```

look deceptively similar but have different values. Properly indented, however, the difference is obvious.

```
(* 5
   (+ 2
      (/ 4 2)
      (/ 8 3)))

(* 5
   (+ 2
      (/ 4 2))
   (/ 8 3))
```

Edwin provides several commands that "pretty-print" your code, indenting lines to reflect the inherent structure of the Scheme expressions (see Section B.2.1 of the *Don't Panic* course manual). In this system, make a habit of typing C-j at the end of a line, instead of RETURN, when you enter Scheme expressions, so that the automatic indentation takes place. Tab and M-q are other useful edwin formatting commands.

While the Scheme interpreter ignores redundant spaces and carriage returns, it does not ignore redundant parentheses. Try evaluating

```
((+ 3 4))
```

and observe what happens.

Copying, Printing, and Emailing

Exercise 2: Print out a transcript of your work from Exercise 1, and bring it to tutorial next week. (You may discuss some of the things you learned about using the Scheme system in Exercise 1, but the real point of this problem is to lead you to learn about your system's print facility.) There is a printer in the lab. If your own machine is not connected to a printer, you can save your work on a floppy and bring it to the lab and print it there.

Exercise 3: Send an excerpt of your answer to Exercise 1 along with a brief, but warm and enthusiastic, `email` message to your tutor and to your recitation instructor. If you're not sure who they are, use whomever you were assigned by the registrar as your recipients. Email addresses are in the General Information handout and are available on the course web page. (The point here is not only to lead you to acquire the rudiments of email management, but also to allow your tutor to add your email return address to the recitation section email list.)

Developing Code Incrementally

When you work on problem sets, you will often develop a large body of code bit by bit. If you evaluate pieces of code in the same way as in the last section, then after a while, your buffer will be very cluttered. It will be hard for you to save it and reload it later, or to organize it to turn in. (If you turn it in without organizing and commenting it, of course, your TA will not look at it.) For example, you might end up with different versions of the same procedure next to each other, and you might not remember which one was the working version. The code will be out of order, and you will have your code mixed in with the results of evaluating your expressions. LA's are unable to help students who work this way, because their code is unreadable.

Instead, take advantage of a special feature of edwin. Edwin can have many buffers open at once. For example, you have already seen that you can have a buffer that you can use to interact with the Scheme interpreter. In addition, you might have a buffer containing Scheme code that we gave you, one buffer containing your own Scheme code, one buffer containing a draft of the write up you are planning to turn in to your TA, one buffer that you are using to compose email to 6001-feedback, and one buffer in which you are working on a haiku.

Type `M-x load-problem-set`, and then the number 0 at the prompt to load the code for Problem Set 0. This will load definitions of the following three procedures, `p1`, `p2`, and `p3`:

```
(define (p1 x y)
  (+ (p2 x y) (p3 x y)))

(define (p2 z w)
  (* z w))

(define (p3 a b)
  (+ (p2 a) (p2 b)))
```

Your editor will also have a buffer with the file containing these definitions. You can switch to it by opening the buffer menu, using C-x C-b (which will also display two buffers at once), and then using C-x o to put the cursor in the other buffer, and then putting the cursor on the line labeled "debug.scm", and then pressing 1.

Look in sections 4 and 5 and Appendix B of the *Don't Panic* manual for other useful commands, including C-x C-f, C-x C-s, C-x 1, and C-x 2.

Note, finally, that even though your Scheme code is not in the buffer that you are using to interact with the Scheme interpreter, you can still evaluate definitions easily, using C-x C-e or M-o (once again, see the *Don't Panic* manual.)

As a last resort, although not recommended, you can always use cut and paste. See C-k, C-space, C-w, M-w, C-y, and M-y.

Exercise 4: Try adding the following definition to your `debug.scm` buffer, and use either C-x C-e or M-o to evaluate it:

```
(define (square x)
  (* x x))
```

If you are successful, you should be able to evaluate the following expression without generating an error:

```
(square 3)
```

Debugging

Correcting errors in programs is an inevitable part of programming, and you will surely need to correct errors in your programs during the semester. Often these errors become apparent only when you try to run the programs. In computer jargon, these "run time" errors are called *bugs* and their detection and repair is called *debugging*.

There is one straightforward debugging facility called "tracing" which is illustrative of professional debugging tools and is easy to understand. Tracing helps you understand and detect errors in your programs by "watching them run."

Exercise 5: When you evaluate the expression `(trace p2)` you cause procedure `p2` to report the values of the arguments and the value it returns every time it is applied to arguments. Thus, if you evaluate `(p2 3 4)`, you will see the following on your screen:

```
(p2 3 4)
[Entering #[compound-procedure 1 p2]
    Args: 3
          4]
[12
      <== #[compound-procedure 1 p2]
    Args: 3
          4]
;Value: 12
```

When you are satisfied that you understand how `p2` works you can evaluate `(untrace p2)` to stop the reporting process.

Tracing in this way can help you spot situations where the wrong procedure is being applied, or the right procedure is applied to the wrong arguments. Moreover, there is no modification to the code of the procedure being traced – an important feature since custom modifying the code for tracing can, by itself, introduce errors. The *Don't Panic* manual describes the MIT Scheme `error` procedure that can be used to display similar information on a conditional basis, such as when the wrong type of argument is passed to a procedure.

Exercise 6: Evaluate the expression `(p1 1 2)`. This should generate an error message from your Scheme system. In MIT Scheme the message would be:

```
;The procedure #[compound-procedure P2] has been called with 1 argument
;it requires exactly 2 arguments.
;Type D to debug error, Q to quit back to REP loop:
```

Don't panic. The message tells you that the error was caused by a procedure `P2` being called with one argument, which is the wrong number of arguments for that procedure. Unfortunately, the error message doesn't say where in the code the error occurred. This example is simple enough that, along with this partial error information about the source of the problem, you will likely find the bug without much trouble just by reading the code. Try it, and prepare to explain in tutorial how you found (or failed to find) the bug. There is no written material to submit for this Exercise.

Exercise 7: Scheme documentation One technique used a lot in 6.001 is the choice of informative names for procedures and their arguments, as in

```
(define (square-and-add number-1 number-2)
  (+ (* number-1 number-1)
     (* number-2 number-2)))
```

Another example is

```
(define pi 3.14159)
(define twice-pi
 (* 2 pi))
```

Here `twice-pi` is not a number, but a Scheme *variable* whose value is the number resulting from the evaluation of `(* 2 pi)`, namely, two times 3.14159. You might even try

```
(define sqrt2
  (sqrt 2))
(define 3plus4
  (+ 3 4))
```

Most Scheme systems will allow use of such variable names, but some are not officially allowed.

All the 6.001 Scheme systems, and all other Scheme systems we know of, offer the user access to the "Revised ^ 4 Report on the Algorithmic Language Scheme" which is the most recent edition of the official definition of Scheme. Find the description of "identifiers" in the copy of the R4RS reference in your system. Be prepared to explain where you found it in tutorial next week. There is nothing written to turn in for this exercise.

(You are welcome to try to figure out which of the variables above are officially allowed, but this is not of any real importance. Don't be discouraged if you are unable to decipher the information in R4RS. R4RS is a reference, not a user's manual, and its specifications are often technical and terse. You are also welcome to test your own system to see which of the above variables it allows — Scheme systems differ on how they handle these examples.)