MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996-97

Problem Set 1

Programming in Scheme

Issued: Thursday, September 5, 1996
Due: Friday, September 13, in recitation.
Tutorial preparation: None for this week.

Reading Assignment: Sections 1.1 and 1.2 of *SCIP.*

Introduction

Every homework assignment contains two kinds of assignments:

- *Written problems:* Your solutions to assigned problems should be handed in at recitation, **late work will not be accepted.** Begin working on the problems now. Assigned problems include both *pre-laboratory* exercises that should be worked at home and *laboratory exercises* that require interaction with a computer that interprets Scheme. Laboratory exercises should be read and understood *before coming to lab:* your efforts in lab will be more effective if you have a good understanding of what is required and have developed an initial plan for working the lab exercises. It is generally much more efficient to test, debug, and run a program that you have planned before coming into the lab than to try to do the planning "online". The experience of students who have taken 6.001 in previous terms has consistently indicated that failing to prepare ahead for laboratory assignments generally ensures that the assignments take much longer than necessary. Also, it is much to your advantage to start the lab work early, rather than waiting until just before it is due.
- *Tutorial preparation:* In tutorial you will be asked to discuss questions raised in the assignment. Your tutor may choose not to cover every question every week, but you should be prepared to discuss them. You need not write up formal answers to these, other than making notes for yourself, if you choose. These tutorial questions, or ones very similar to them, may also appear on quizzes this semester. Your tutor will also review your written homework and discuss the problems with you.

To work on this assignment, you will need a copy of the text and copy of the lab manual package. To obtain the latter, follow the instructions given in the General Information memo.

1. Getting started in the lab

This section is a repeat of material you should have seen in Problem Set 0. We are repeating it here for those of you who may have joined the class late. If you have already done this material for Problem Set 0, you may move straight to Section 2.

When you come to the lab, find a free computer and log in. You should log in as `u6001`. If necessary, initialize[1] one of your floppy disks, as described in Chapter 3 of the Don't Panic manual. You should also write your name and address on a label affixed to the floppy disk. Floppy disks are notoriously unreliable storage media, so it is a good idea to copy your data onto a second disk (this is used only for this purpose) when you have completed each laboratory assignment. See the description of how to copy disks in the Don't Panic manual, and do not hesitate to ask the lab assistants for help.

To work on assignments you will need a copy of the course notes and the lab manual (*Don't Panic*), but you can work the finger exercises in this handout without that material. The purpose of the finger exercises is to familiarize you with the 6.001 laboratory and programming system. Spending a little time on simple mechanics now will save you a great deal of time over the rest of the semester.

Most of your interactions with the system will be through a text editing system called Edwin, which is an Emacs-like editor. If you are not familiar with such editors, you may find it useful to work your way through the on-line Edwin/Emacs tutorial. This can be invoked via `C-h` followed by T.[2] The tutorial is fairly long, and you may get bored before you finish. Try to skim all the topics so you know what's there. You'll need to gain reasonable facility with the editor in order to complete the exercises below.

Evaluating expressions

The language in which you will be working this term is Scheme, a dialect of Lisp. You will be hearing lots about this language next week, but you can already get some experience in using the language here. All Scheme procedures are built out of expressions, with the simplest expressions consisting of 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.

After you have learned something about Edwin, go to the Scheme buffer (i.e., the buffer named `*scheme*`).[3] As with any buffer, you can type Scheme expressions, or any other text into this buffer. What distinguishes the Scheme buffer from other buffers is the fact that underlying this buffer is a Scheme evaluator, which you can ask to evaluate expressions, as explained in the section of the Edwin tutorial entitled "Evaluating Scheme expressions."

Type in and evaluate (one by one) at least some of the expressions listed below. If the system gives a response you do not understand, discuss this among your group or with a lab TA to try to clarify your uncertainty.

[1] The disks provided by the EECS Instrument Room have already been initialized.

[2] `C-h` is entered by holding the CONTROL key down while typing `h`.

[3] If you don't know how to do this, go back and learn more about Edwin.

```
25

-37

(- 8 9)

(> 3.7 4.4)

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

(* (- 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 Sec. B.2.1 of *Don't Panic.*). Make a habit of typing `ctrl-j` at the end of a line, instead of RETURN, when you enter Scheme expressions, so that the automatic indentation takes place.

Creating a file

Since the Scheme buffer will chronologically list all the expressions you evaluate, and since you will generally have to try more than one version of the same procedure as part of the coding and

debugging process, it is usually better to keep your procedure definitions in a separate buffer, rather than to work only in the Scheme buffer. You can save this other buffer in a file on a floppy disk so you can split your lab work over more than one session. (The course notes package contains two floppy disks.)

Chapter 5 of *Don't Panic* describes files in more detail. The basic idea is to create another buffer, into which you can type your code, and later save that buffer in a disk file. You do this by typing `C-x C-f` *filename*. If you already have a buffer open for that file Edwin simply switches you into this buffer. Otherwise, Edwin will create a new buffer for the file. If you give the system a name that has not yet been used, with an extension of `.scm`, Edwin will automatically create a new buffer with that file name, in `scheme` mode. In a Scheme-mode buffer some editing commands treat text as code, for example, typing `C-j` at the end of a line will move you to the next line with appropriate indentation.

Once you are ready to transfer your procedures to Scheme, you can use any of several commands: `M-z` to evaluate the current definition, `ctrl-x ctrl-e` to evaluate the expression preceding the cursor, `M-o` to evaluate the entire buffer, or by marking a region and using `M-x eval-region`. Each of these commands will cause the Scheme evaluator to evaluate the appropriate set of expressions (usually definitions). By returning to the `*scheme*` buffer, you can now use these expressions.

2. Debugging

During the semester, you may need to correct errors in your programs. Often these errors become apparent only when you try to run the programs. Correcting errors ("bugs") under these conditions is called "debugging". On February 20 at 7PM (room to be announced) we will be offering an optional lecture on debugging. But some basic ideas can help your programming activities now.

Use the Edwin `M-x load-problem-set` command to load the code for problem set 1. This will load definitions of the following three procedures `p1`, `p2` and `p3`:

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

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

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

A very simple technique called "tracing" can help you understand how your procedures operate. When you evaluate the expression `trace p2` you cause procedure `p2` to report the values of its arguments when it is called and the value it returns. Thus in evaluating `(p2 3 4)` you would 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 is a simple way to get help you detecting errors in the logic of your program design, calling calling procedures with the wrong arguments, or calling the wrong procedures. Moreover, there is no modification to the code of the procedure being traced (which can, by itself, introduce errors). The *Don't Panic* manual describes the `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.

A more powerful way to get help with programming errors is to use the Scheme debugger itself. The following tutorial is intended to demonstrate the basic elements of debugging. Additional information about the debugger can be found in *Don't Panic*, and by typing `?` in the debugger.

In the Scheme buffer, evaluate the expression `(p1 1 2)`. This should signal an error, with the message:

```
;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. Beginners have a tendency, when they encounter an error, to quickly type `q`, often without even reading the error message. Then they stare at their code in the editor trying to discover what caused the problem. Indeed, this example is simple enough so that you probably can find the bug by just reading the code. Instead, however, let's see how Scheme can be coaxed into producing some helpful information about the error.

First of all, there is the error message itself. It tells you that the error was caused by a procedure being called with one argument, which is the wrong number of arguments for that procedure. Unfortunately, the error message alone doesn't say where in the code the error occurred. In order to find out more, you need to use the debugger. To do this type `D` to start the debugger.

Using the debugger

The debugger also allows you to examinine pieces of the execution in progress, in order to learn more about what may have caused the error. When you start the debugger, it will create a new window showing two buffers. The top buffer should look like this.

```
COMMANDS:   ? - Help   q - Quit Debugger   e - Environment browser
```

```
This is a debugger buffer:
Lines identify stack frames, most recent first.
   Sx means frame is in subproblem number x
   Ry means frame is reduction number y
The buffer below describes the current subproblem or reduction.
-----------
The *ERROR* that started the debugger is:
  The procedure #[compound-procedure 119 p2] has been called with 1
argument;
  it requires exactly 2 arguments.

>S0  (#[compound-procedure 119 p2] 2)
    R0  (p2 b)
 S1  (+ (p2 a) #(p2 b)#)
    R0  (+ (p2 a) (p2 b))
    R1  (p3 x y)
 S2  (+ (p2 x y) #(p3 x y)#)
    R0  (+ (p2 x y) (p3 x y))
    R1  (p1 1 2)
 --more--
```

You can select a frame by clicking on it with the mouse or by using the ordinary cursor line-motion commands to move from line to line. Notice that the information bottom buffer changes as the selected line changes.

The frames in the list in the top buffer represent the steps in the evaluation of the expression. There are two kinds of steps—subproblems and reductions. This idea will be discussed in lecture on September 21. For now, you should think of a reduction step as transforming an expression into "more elementary" form, and think of a subproblem as picking out a piece of a compound expression to work on.

So, starting at the bottom of the list and working upwards, we see `(p1 1 2)`, which is the expression we tried to evaluate. The next line up indicates that `(p1 1 2)` reduces to `(+ (p2 x y) (p3 x y))`. Above that, we see that in order to evaluate this expression the interpreter chose to work on the subproblem `(p3 x y)`, and so on, moving upwards until we reach the error: the call to `(p2 b)` from within the procedure `p3` has only one argument, and `p2` requires two arguments.[4]

Take a moment to examine the other debugger information (which will come in handy as your programs become more complex). Specifically, in the top buffer, select the line

```
>S2  (+ (p2 x y) #(p3 x y)#)
```

The bottom buffer should now look like this:

[4]Notice that the call that produced the error was `(p2 b)`, and that `(p2 a)` would have also given an error. This indicates that in this case Scheme was evaluating the arguments to `+` in right-to-left order, which is something you may not have expected. You should never write code that depends for its correct execution on the order of evaluation of the arguments in a combination. The Scheme system does not guarantee that any particular order will be followed, nor even that this order will be the same each time a combination is evaluated.

```
                          SUBPROBLEM LEVEL: 2
Expression (from stack):
 Subproblem being executed highlighted.
    (+ (p2 x y) (p3 x y))
--------------------------------------------------------------------------
ENVIRONMENT named: (user)
    p1 = #[compound-procedure 31 p1]
    p3 = #[compound-procedure 32 p3]
    p2 = #[compound-procedure 27 p2]
==> ENVIRONMENT created by the procedure: P1
      x = 1
      y = 2
--------------------------------------------------------------------------
;EVALUATION may occur below in the environment of the selected
frame.
```

The information here is in three parts. The first shows the expression again, with the subproblem being worked on. The next major part of the display shows information about the *environments*. We'll have a lot more to say about environments later in the course, but, for now, notice the line

```
==> ENVIRONMENT created by the procedure: P1
```

This indicates that the evaluation of the current expression is within procedure `p1`. Also we find the environment has two *bindings* that specify the particular values of `x` and `y` referred to in the expression, namely `x = 1` and `y = 2`. At the bottom of the description buffer is an area where you can evaluate expressions in this environment (which is often useful in debugging).

Before quitting the debugger try one final experiment (you may have already done this). Continue to scroll down through the stack past the line: `R1 (p1 1 2)` (you can also click the mouse on the line `--more--` to show the next subproblem). You will then see additional frames that various complicated expressions. What you are looking at is some of the guts of the Scheme system—the part shown here is a piece of the interpreter's read-eval-print program. In general, backing up from any error will eventually land you in the guts of the system. (Yes: almost all of the system is itself a Scheme program.)

You can type `q` to return to the Scheme top level interpreter.

3. Formal Assignment

Assignments will usually contain both Exercises and Laboratory Work.

Exercises

You should work the following exercises *before* going to lab. This will require that you have mastered the basic skills required for the main assignment. Although you can use a Scheme interpreter to check your work, you should not postpone working the exercises until you go to lab. Write up your solutions to the exercises and include them as part of your homework.

Exercise 1 Work Exercise 1.1 of the test. You should determine the answers to this exercise without the help of a computer, then check your answers in lab.

Exercise 2. Work Exercise 1.2 in *SICP.*

Exercise 3. Work Exercise 1.5 in *SICP.*

Exercise 3. Work Exercise 1.6 in *SICP.*

Laboratory Work: The New York Sines – All the digits that are fit to print

Now that you've gained some experience with Scheme, you should be ready to work on the programming assignment. When you are finished in the lab, you should write up and hand in the numbered problems below. You may want to include listings and/or pictures in your write-up. Chapter 1 of the *Don't Panic* manual explains how to use the lab printers.

Subsequent laboratory assignments will include large amounts of code, which will be loaded automatically into an edit buffer when you begin work on the assignment. This time, however, you are to type the code yourself, to get practice with the editor.

Scheme, like most programming languages, comes with a set of procedures predefined, or "built-in", especially procedures for doing numerical computations. Thus, for example, the Scheme procedures `sin` and `cos` will return the sine and cosine of a single argument which specifies an angle in radians. In this problem set, we are going to explore ways of building such procedures, in particular the sine function, using simpler pieces.

The first way we will do this is to use the following infinite product definition:

$$\begin{aligned}\sin(x) &= x \prod_{k=1}^{\infty} \left[1 - \left(\tfrac{x}{k\pi}\right)^2\right] \\ &= x \left[1 - \left(\tfrac{x}{\pi}\right)^2\right] \left[1 - \left(\tfrac{x}{2\pi}\right)^2\right] \left[1 - \left(\tfrac{x}{3\pi}\right)^2\right] \ldots\end{aligned}$$

Problem 1 We can use this idea to write a procedure to estimate the sine function. Below is an outline of the procedure `my-sine-prod` which should take two arguments, an angle in radians `x`, and a number of terms `n`. We will assume that $n \geq 1$, and `n` refers to the number of terms in $[\cdot]$'s that will be included in the product. In other words, if $n = 1$ we want the procedure to compute

$$x \left[1 - \left(\frac{x}{\pi}\right)^2\right]$$

and if $n = 3$, we want the procedure to compute

$$x \left[1 - \left(\frac{x}{\pi}\right)^2\right] \left[1 - \left(\frac{x}{2\pi}\right)^2\right] \left[1 - \left(\frac{x}{3\pi}\right)^2\right].$$

Here is the outline. Normally, we will give you such templates automatically loaded, but for this first time, we ask you to type it in to get the practice using the system.

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

(define my-sine-prod
  (lambda (x n)
    (define helper
       (lambda (x m)
         (cond ((> m n) 1)
               (else
                 ;; need to fill this in
                ))))
    (* x (helper x 1))))
```

You should be able to complete the procedure by modeling it after the recursive procedures you saw in lecture or in the text. Turn in a listing of your definition.

Problem 2 Try out your procedure on a set of test values. Turn in a listing of the tests you run.

Problem 3 We should be able to carefully examine the behavior of our procedure, especially as we increase the number of terms in the product. Write a procedure `walk-through-prod` which takes two arguments, an angle in radians, and a maximum number of terms to include. This procedure should print out a sequences of values showing the approximation to sine of the angle when including 1 term, 2 terms, ..., n terms. For example, we would expect:

```
(define pi 3.1415927)

(walk-through-prod (/ pi 2) 5)
1.1780972625
1.10446618359375
1.0737965673828124
1.057008652267456
1.0464385657447812
;Value: #t
```

Write such a procedure. Turn in a copy of your listing. Use your procedure to see how many terms you need to include to approximate $\sin 0$ to an accuracy of 0.05. What about $\sin \frac{\pi}{2}$ to an accuracy of 0.05, to an accuracy of 0.01, to an accuracy of 0.005?

Problem 4 We know that $\sin \frac{\pi}{2} = 1$, and in fact, $\sin \frac{(4k+1)\pi}{2} = 1$ for any positive integer k. We would like to find out how many terms in the product expansion for sin we need to include to get a good approximation, as we make the angle bigger. Write a procedure `test-prod` which takes two arguments, a value k and a limit ϵ. This procedure should return the miminum number of terms needed to bring the approximation for $\sin \frac{(4k+1)\pi}{2}$ to within ϵ of the correct answer of 1. Turn in a listing of your procedure. How many terms are needed to approximate $\sin \frac{\pi}{2}$ to within 0.05, to within 0.01? What about $\sin \frac{5\pi}{2}$ to within 0.05? What about $\sin \frac{9\pi}{2}$ to within 0.05?

You can see that this approximation can be quite slow to converge, that is, we need a lot of terms. So we want to look at a second way of approximating the sine function.

The second way we will do this is to use a Taylor series approximation. Any function $f(x)$ can be approximated by choosing a point x_0 about which to perform the approximation, and using a series of values of increasingly larger derivatives of the function:

$$f(x) \approx f(x_0) + \frac{(x - x_0)}{1!}f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \frac{(x - x_0)^3}{3!}f'''(x_0) + \ldots$$

If we want to get an approximation for $f(x) = \sin(x)$ it is convenient to choose $x_0 = 0$ so that $df(x)/dx = \cos(x)$ which at $x_0 = 0$ has the value 1, $d^2f(x)/dx^2 = -\sin(x)$ which at $x_0 = 0$ has the value 0, ${}^3df(x)/dx^3 = -\cos(x)$ which at $x_0 = 0$ has the value -1, $d^4f(x)/dx^4 = \sin(x)$ which at $x_0 = 0$ has the value 0, and then the pattern repeats.

This means

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

It is convenient to rewrite this as a sequence of continued products:

$$\sin(x) \approx x\left[1 - \frac{x^2}{3 \cdot 2}\left[1 - \frac{x^2}{5 \cdot 4}\left[1 - \frac{x^2}{7 \cdot 6}\ldots\right]\right]\right]$$

Problem 5 Write a procedure `my-sine-taylor` which takes an angle in radians and a number of terms n as arguments, and uses this idea to approximate the sine. Turn in a listing of your procedure and some examples of testing it.

Problem 6 Similar to Problem 4, write a procedure `test-taylor` that allows us to determine how many terms are needed to approximate this version of sine. How many terms are needed to approximate $\sin\frac{\pi}{2}$ to within 0.05, to within 0.01? What about $\sin\frac{5\pi}{2}$ to within 0.05? What about $\sin\frac{9\pi}{2}$ to within 0.05?

We still have fairly slow methods for approximating sines of large arguments. So here is a third way to do this.

A common technique in creating computational methods is to reduce a problem to simpler versions of the same problem. In the case of sine functions, we have noticed that our earlier methods work well for small angles, but not so well for larger ones. So if we could reduce the computation of sine of a large argument to sines of smaller ones, we would be in better shape. Well, we can using the following decomposition:

$$\begin{aligned}\sin(x) &= \sin(\tfrac{x}{3} + \tfrac{2x}{3}) \\ &= \sin\tfrac{x}{3}\cos\tfrac{2x}{3} + \cos\tfrac{x}{3}\sin\tfrac{2x}{3}\end{aligned}$$

$$\begin{aligned} &= \sin\tfrac{x}{3}\left[\cos^2\tfrac{x}{3} - \sin^2\tfrac{x}{3}\right] + \cos\tfrac{x}{3}\left[2\sin\tfrac{x}{3}\cos\tfrac{x}{3}\right] \\ &= 3\sin\tfrac{x}{3}cos^2\tfrac{x}{3} - \sin^3\tfrac{x}{3} \\ &= 3\sin\tfrac{x}{3} - 4\sin^3\tfrac{x}{3} \end{aligned}$$

Problem 7 Now we can use this idea to write a new approximation to the sine function, which we will call `reduce-sine`. The idea is that if the first argument to `reduce-sine`, which is the angle, is less than a second argument `epsilon`, then we will use `my-sine-prod` to compute the answer, using the supplied angle and the third argument `n` which specifies the number of terms to use in the product expansion. Otherwise, we will use the above formula to recursively reduce `reduce-sine` to applications of `reduce-sine` with smaller arguments.

Write such a procedure, and turn in a copy of your code. Also show it being applied to some test arguments. Try it on 2.5π with $\epsilon = 0.05$ using 5 terms in the product, on 8.5π, on 20.5π, on 200.5π. Compare the answers you get with the approximations you get using `my-sine-prod` directly on these arguments.

Problem 8 It turns out we can play the same game with other approximations. For example, the following holds:

$$\sin x = 16\sin^5\frac{x}{5} - 20\sin^3\frac{x}{5} + 5\sin\frac{x}{5}$$

Use this to write a second procedure `new-reduce-sine` similar to the previous problem. Turn in a copy of your listing.

Problem 9 Now, let's compare the two. Complete the following table, using $n = 5$ for the number of terms and $\epsilon = 0.05$ in both cases:

angle	`reduce-sine`	`new-reduce-sine`
0.5π		
2.5π		
10.5π		
20.5π		
100.5π		

Which version is better?