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

Problem Set 1

Getting Started

Issued: Tuesday, February 4

Due: In recitation on Wednesday, February 12 for recitations held at 9, 12, 1 and 2; and on Friday, February 14 for recitations held at 10 and 11.

Tutorial Preparation: None for this week.

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

Introduction

In general, 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.** 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. Because the first week of classes will be a bit chaotic as section assignments get sorted out, we will not hold tutorials next week and thus there are no tutorial exercises in this first problem set. Subsequent problem sets will have such exercises to prepare.

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 first part of the problem set is intended to get you used to using the machines in the lab.

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.

You may prefer to use means other than floppies for storing your work. In particular, you can **ftp** from a lab machine to other machines, such as Athena. In this way, if you prefer, you can store your work in your own Athena area, simply transfer it into the lab when you are ready to start work, and save it back out from the lab when you are done.

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

[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.

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.

```
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 18 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. 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:

```
                         SUBPROBLEM LEVEL: 2
```

[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.

```
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

The formal assignment part of this problem set has two parts. We strongly suggest that you start on the first part as soon as possible, then work on the second part after Thursday's lecture.

Exploring the system

The following exercises are meant to help you practice editing, and using on-line documentation.

Exercise 1: Practice with the editor Copy the results of evaluating some of the expressions in Section 1 from the *scheme* buffer into an answer buffer.

Exercise 2: More practice with the editor Find the Free Software Foundation copyright notice at the end of the Edwin tutorial. Copy it into the answer buffer.

Exercise 3: Learning to use `Info` Start up the `info` program with the Edwin command `M-x info`. `Info` is a directory of useful information. You select a topic from the menu, by typing `m` followed by the name of the topic. You can type a few characters that begin the topic name, and then type a space, and Edwin will complete the name. One of the `info` options (type `h`) gives you a brief tutorial in how to use the program. Use `info` to find the cost of a 12-inch cheese pizza from Pizza Ring, and copy this to the answer buffer. (Extra credit: Is the price listed in the file still correct?)

Exercise 4: Hacker Jargon The info `Jargon` entry is a collection of hacker terms that was eventually published in 1983 as the book *The Hacker's Dictionary.* The `New Jargon` info entry is an expanded version that was published in 1991 by MIT Press as *The New Hacker's Dictionary.* The old Jargon file is structured as an `info` file, with submenus; the new version is a single text file, which you can search. Find the definition of "terminal brain death" from the new jargon file, and find the definition of "Phase of the Moon" from either jargon file. Include these in the answer buffer.

Exercise 5: Scheme documentation The `Scheme` info entry is an on-line copy of the Scheme reference manual that is distributed with the notes. Find the description of "identifiers" in the documentation. What is the longest identifier in the list of example identifiers?

Exercise 6: Getting information at MIT The Edwin command `M-x shell` will create a Unix shell buffer, which you can use to run various Unix programs. One of the more interesting ones is `finger`. For example, typing

`finger` *name*`@mit`

will show you information from the MIT directory. Find out who the following people are at MIT and what they do: Joel Moses, Michael C. Behnke, Rosalind Williams.

Exercise 7: Getting information from around the world You can also use the finger program to query computers on the Internet, all over the world. You can finger a particular name at some location, or just finger the location (e.g., `finger @mit.edu`) to get general information. Try fingering some of the following places:

- `cs.berkeley.edu`—a computer run by the computer science department at UC Berkeley
- `idt.unit.no`—a computer at the Norwegian Institute of Technology, a part of the University of Trondheim, Norway.
- `whitehouse.gov`—the White House

See what else you can find. Include some piece of this in your answer file.

Turning in your answers to this part

The section on "Feedback" in the *Don't Panic* manual describes how to send Email from the 6.001 lab. Send your answer buffer to your tutor. Do this by starting a mail message and using the Edwin `M-x insert-buffer` command. Don't forget to include a reply email address (or else put your name in the message) so your tutor will know that the message is coming from you.

Mail in your answers only for the first part of this problem set.

Pi Not? – computing approximations to π

In this last part of the problem set, we are going to consider some simple procedures for computing approximations to π.

Most computer systems will provide built in procedures for numerical computation, such as trigonometric and inverse trigonometric functions, as well as providing values for important constants such as π or e. In this part of the problem set, we are going to consider some relatively simple methods for computing approximations to π.

Our first method for computing π is due to a method of Leibniz. In particular, we can approximate $\frac{\pi}{4}$ by the following infinite sum:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

and thus one way to get an approximation to π is to write a procedure that will add up the first n terms of this sum. Thus, if we ask the procedure to sum the first two terms, it should return the value:

$$1 - \frac{1}{3} = .6666666666666667$$

and if we ask it to sum the first three terms, it should return the value

$$1 - \frac{1}{3} + \frac{1}{5} = .8666666666666667$$

and so on.

When you load up[5] problem set 1, a file called `pi-code.scm` will be loaded into your machine which contains the following definitions:

[5] Remember this means using the Edwin **`M-x load-problem-set`** command

```
(define leibniz-term
  (lambda (i)
  ;;; this procedure takes one argument, an integer, and returns the value of
  ;;; the ith term in the Leibniz approximation
    (/ (expt -1.0 (- i 1))
       (- (* 2 i) 1))))

(define pi-approx-leibniz-iterative
  (lambda (n)
    ;; this procedure takes one argument, an integer, and returns the value of
    ;; the sum of the first n (starting from 1) terms of the leibniz
    ;; approximation
    (define helper
      (lambda (i sum)
        ;; an internal procedure to sum up the n terms
        ;; i is the index of the next term  to add
        ;; sum is the current sum of the previous terms
      (if (= i n)    ; test to see if we are done
          sum        ; if so return the sum
          (helper (+ i 1)   ; otherwise go on to the next term
                  (+ (liebniz-term (+ i 1))
                            ; while adding this term to the sum
                     sum)))))
    ;; we need to start the process with the first term
    (helper 1 (leibniz-term 1))))

(define pi-approx-leibniz-recursive
  (lambda (n)
    ;; this procedure takes one argument, an integer, and returns the value of
    ;; the sum of the first n (starting from 1) terms of the leibniz
    ;; approximation
    ;; in this case the procedure will count down from the nth term
    (if (= n 1)  ; test to see if done
        1        ; if so return the value of the 1st term
        (+ (liebniz-term n)  ; otherwise add in the value of nth term
                             ; to the value of the previous n-1 terms
           (pi-approx-liebniz-recursive (- n 1))))))
```

These are two different procedures to compute the Leibniz approximation. We'll see more about how these procedures differ in lecture on February 11, but for now we can just use them.

Exercise 8: You should look these procedures over carefully, to ensure that you understand how they are implementing the idea of summing up terms. Try evaluating these two procedures, with different values for the argument, `n`. What value of $\pi/4$ do they produce for $n = 1, 10, 100, 1000$? Do they produce the same value? Note, by the way, that the constant `best-pi-fourth` provides a very close approximation to the correct value. Turn in a transcript of your evaluations.

This particular series is fairly slow in converging. A variation on the Leibniz series is:

$$\frac{\pi}{4} = 2\left[\frac{1}{1\cdot 3} + \frac{1}{5\cdot 7} + \frac{1}{9\cdot 11} + \ldots\right]$$

In general, the ith term of this series is

$$\frac{1}{[4i-3]\,[4i-1]}$$

Exercise 9: Create procedures to implement this approximation to $\pi/4$, again with the idea that your procedure will take one argument `n` and will return the value of adding the first n terms of the approximation. You may use either of the approaches used in the previous exercise as a template, if you like. Turn in a listing of your procedures.

Try evaluating your procedure, with different values for the argument, `n`. What value of $\pi/4$ does it produce for $n = 1, 10, 100, 1000$? Turn in a transcript of your evaluations.

A different approximation to $\pi/4$ was discovered by a mathematican named Wallis:

$$\frac{\pi}{4} = \frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdots$$

Here, the ith term of this product is

$$\frac{i+1}{i+2}$$

if i is odd, and

$$\frac{i+2}{i+1}$$

if i is even.

Exercise 10: Create procedures to implement the Wallis approximation to $\pi/4$, again with the idea that your procedure will take one argument `n` and will return the value of adding the first n terms of the approximation. You may use either of the approaches used in the previous exercise as a template, if you like. Turn in a listing of your procedures.

Try evaluating your procedure, with different values for the argument, `n`. What value of $\pi/4$ does it produce for $n = 1, 10, 100, 1000$? Turn in a transcript of your evaluations.

As you can see, these procedures get close to the true value at different rates (i.e. take different numbers of terms to get to the same value). It would be useful to have a way of measuring how many terms each procedure takes in order to get to within some specified amount of the true value (as represented by `best-pi-fourth`).

Exercise 11: Using your procedures for the three different methods (the two Leibniz methods and the Wallis method) as a starting point, create new procedures that will take as argument a real number (ϵ) and will return the number of steps required for the procedure to get an approximation that is no more than ϵ different from the true value.

Turn in a listing of your procedures. Also fill out a small chart indicating how many steps are required for each method to get to within $\epsilon = .1, \epsilon = .01, \epsilon = .001, \epsilon = .005$.