MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

Problem Set 1

- Issued: Tuesday, February 3
- Tutorial preparation for: Week of February 9
- Written solutions due in two parts:
  - "Getting started" assignment is due by email to your tutor and recitation instructor by the end of the weekend (i.e., before Monday morning, February 9)
  - Computer assignment is due in recitation on Friday, February 13
- Reading:
  - "6.001 General Information," which was handed out at the first lecture. Be especially sure to read the 6.001 policy on collaborative work.
  - Scan the 6.001 Home Page on the web at http://mit.edu/6.001. All handouts (including this one) will be available on this page in case you need extra copies. In addition, the web page contains announcements, software for Scheme and for the weekly problem sets, documentation, advice on where to get help, and other useful information . You should make it a habit to look at this page at least once a week during the semester.
  - Textbook reading: In general, the lectures will assume that you've read the appropriate sections of the text **before** coming to lecture. For this week read section 1.1 before lecture on Feb. 5 and section 1.2 before lecture on Feb. 10.

Every homework assignment describes two sorts of activities:

- *Written programming assignments:* Solutions should be handed in at recitation, and **late work will not be accepted.** You can begin working on the assignment now, and through next week. It is to your advantage to get computer work done early, rather than waiting until the night before it is due. You should also read over and think through the assignment before you sit down at the computer. It is generally much more efficient to test, debug, and run a program that you have thought about beforehand, rather than doing the planning "online". Diving into program development without a clear idea of what you plan to do generally ensures that the assignments will take much longer than necessary. Your tutor will look over the homework you hand in and review it with you in tutorial.
- *Tutorial preparation:* There are some questions that you should be ready to present orally in tutorial. 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 questions very similar to them, may also appear on quizzes this semester.

This first assignment also contains two additional parts:

- "Getting started" is designed to get you set up with Scheme. Please do this as soon as possible, but in any event, by no later than this weekend.
- "Debugging tools" is computer exercise designed to acquaint you with Scheme's debugging facilities, which we suggest you work through after you've completed the other computer assignment in section 3. There is nothing to turn in for section 4, so we won't know whether or not you've really done it: but we *strongly* suggest that you take the time to go through the exercise, since knowledge of the debugging tools can save you a lot of time on future problem sets.

1. Getting started

The purpose of this section is to get you started using Scheme as quickly as possible. Start by looking at the subject web page labeled "Using Scheme in 6.001". You'll see there that the 6.001 Scheme system runs in the 6.001 Lab (34-501), on PCs running GNU/Linux, Windows NT4, or Windows 95 (sorry, no Macs or Athena); and that you can move among the different implementations as you require. If you have a PC capable of running Scheme, we suggest that you install it there, since it will be convenient for you to work at home a lot of the time. The 6.001 lab is probably the best place to work if you want help, since that is staffed by knowledgeable and friendly Lab Assistants.

Whether you're working with your machine or MIT's, the source of all useful information is the 6.001 web page: **http://mit.edu/6.001**

Starting scheme

The first thing to do is to get an implementation of Scheme and start using it:

- *In the 6.001 lab* ... A good way to get started learning Scheme is to do this first section of the assignment in the Lab, where there are LAs to help you, and then install Scheme on your own PC and run through the section again to verify that your home system works just like the one in the lab. Find a free lab computer and log in with the username `u6001`. You will need to use a floppy disk in order to save your files.[1]
- *On your home computer* ... follow the instructions on the Web page for downloading and installing Scheme. *Important:* The version of Scheme we are using this spring—Scheme 7.5a—is not the same version as the one used last fall. If you have an installation from the fall, get a new one. Some of the problem sets (PS2, for example) will not work on the old one. Once you've installed the Scheme system you should be able to simply start it and do this part of the assignment. If you are using your own computer, you'll also have to download the code for each weekly assignment, which you'll find on the web page. You'll need to download the PS1 code for doing the later parts of this assignment, but don't worry about that now.

[1] The lab system encourages you to use a floppy disk if you want to save your files. There is no permanent storage for students on the 6.001 lab system. You can also transfer your files to your Athena directory over the network. See the 6.001 web page for information on how to transfer files to and from the lab: don't use FTP!

Learning to use Edwin

In 6.001, we use a text-editing system called *Edwin*. Edwin is a implementation in Scheme of the Emacs text editor. If you've used Emacs before, you'll find Edwin virtually identical. If you have never used Emacs before (or even if you have) spend about 15 minutes going through the on-line tutorial, which you start by typing `C-h` followed by `t`. (`C-h` means "control-`h`": to type it, hold down the CTRL key and type `h`. Release the CTRL key before typing `t`.) You will probably get bored before you finish the tutorial. (It's too long, anyway.) But at least 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 problem sets.[2] To get out of the tutorial, type `C-x k`, which will kill the buffer.

Evaluating expressions

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.

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 $3 + 4$.

Try typing a simple expression such as

```
1375
```

Typing this expression inserts it into the buffer. To ask Scheme to evaluate it, we type `C-x C-e`, which causes the evaluator to read the expression immediately preceding the cursor, evaluate it (according to the kinds of rules described in lecture) and print out the result. Try this.

Numbers are boring expressions to evaluate, since the printed representation of their value is the same as what you type. To see examples of evaluating more expressions, try these:

```
-37

(- 8 9)

(> 3.7 4.4)

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

[2] If you're running Scheme on your own PC and want a little mindless diversion, you can adjust the screen size, position and color. Click on the shield icon at the upper left of the Edwin window to adjust the font and the background color. You can change the foreground color using the command `M-x set-foreground-color`.

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

```
(* (- 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 line breaks. 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* manual). 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))
```

Buffers and files

One could simply type all one's work into the `*scheme*` buffer, but it is usually better to store versions of your code in a separate buffer, which you can then save in a file.

The most convenient way to do this is to split the screen to show two buffers at once—the scheme* buffer and a buffer for the file you are constructing. You will need know how to split the screen (`C-x 2`) and how to move from one half to the other (`C-x o`).

Starting in the `*scheme*` buffer split the screen by typing `C-x 2`. At this point both halves of the screen will show the same `*scheme*` buffer. Now choose a file name ending with `.scm` and type tt

C-x C-f*file name*. The half of the screen you are in will now show a buffer for the new file. You can type in this buffer and evaluate expressions in this buffer. The results will appear in the `*scheme*` buffer. If an error occurs during evaluation you must go to the `*scheme*` buffer to deal with the error.

To evaluate expressions, you can use any of several commands: you can use `C-x C-e` to evaluate the expression preceding the cursor, `M-z` to evaluate the current definition, or `M-o` to evaluate the entire buffer (this does not work in the `*scheme*` buffer. You may mark a region and use `M-x eval-region` to evaluate the marked region. Each of these commands will cause the Scheme evaluator to evaluate the appropriate set of expressions. Note that you can type and evaluate expressions in either buffer, but the values will always appear in the `*scheme*` buffer. See the *Don't Panic* manual for more details.

As a simple example create a new buffer called `ps1-test.scm`. In this buffer create a definition for a simple procedure by typing in the following (verbatim):

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

Now evaluate this definition by using either `M-z` or `C-x C-e`. Next try evaluating:

```
square

(square 4)
```

In fact, if you typed in the definition exactly as indicated above, you should have hit an error at this point. The system will now be offering you a chance to enter the debugger. For now, go to the `*scheme*` buffer and type `Q` to quit the evaluation. (We'll discuss the debugger below in section 4). Go back to your `ps1-test.scm` buffer, and edit the definition to insert a space between `*` and `x`. Re-evaluate the definition, and try evaluating `(square 4)` again.

The system automagically maintains a special read-only buffer, called `*transcript*`, which keeps a history of your interactions with the Scheme evaluator. You can extract pieces of this buffer to document examples of your work for problem sets. To go to this buffer type `C-x b *transcript*`. You can go back to any buffer with the `C-x b` command.

Sending e-mail

That's it for this first part of the problem set, except for reporting the results. Compose an email message containing your definition of square, together with an excerpt from the `*transcript*` buffer showing that it works. If you are using the 6.001 lab, typing `ctl-X m` will open a mail buffer in which to compose your message. The system will ask for your own email address, so the recipient will know who sent the mail. You may insert work by copying from other buffers. To send your completed message type `ctl-C ctl-C`. If you're using your own machine, you can save your work in a file and then mail this using your favorite mailer.

Include in the message a brief, but warm and enthusiastic greeting to your tutor and to your recitation instructor. As part of the message, say which Scheme implementation you are using:

6001 Lab or your own machine (with which operating system?). If you don't know who your tutor and recitation instructor are, look on the 6.001 web page. The point here is not only to make sure you can send mail, but also to enable your tutor to add your email return address to the recitation section email list.

You can learn more about using Scheme in 6.001 from the *Don't Panic* manual. Also, at some point in the next couple of weeks, you should go back and run the Edwin tutorial again, to see what you glossed over the first time.

If you're a glutton for documentation, you can find an (almost) complete description of Scheme commands in the *Scheme Reference Manual*. The *Scheme User's Manual* has some details on MIT Scheme and Edwin. The *Revised Report on the Algorithmic Language Scheme* is the official (and not very readable) specification of the Scheme language. These documents are available on the web page, and they are included with the Scheme implementations. But you shouldn't need to look at any of them, because the material will introduce new elements of Scheme as you need them.

2. Tutorial Exercises

Prepare these exercises for oral presentation in tutorial.

Tutorial exercise 1: Below is a sequence of expressions. Be prepared to indicate the result that is printed by the interpreter in response to each expression. Be prepared to explain why. Assume that the sequence is to be evaluated in the order in which it is presented. (Try to determine the answers to this exercise without actually trying these on a computer.)

```
(> 10 9.7)

+

(define double (lambda (x) (* 2 x)))

double

(define c 4)

c

(double c)

c

(double (double (+ c 5)))

(define times-2 double)

(times-2 c)
```

```
(define d c)

(= c d)

(if (= (* c 6) (times-2 d))
    (< c d)
    (+ c d))

(cond ((>= c 2) d)
      ((= c (- d 5)) (+ c d))
      (else (abs (- c d))))

(define try (lambda (x) (x 3)))

(try double)

(try (lambda (z) (* z z)))
```

Tutorial exercise 2: What is the difference between the following two ways of defining `cube`?

```
(define cube (lambda (x) (* x x x)))

(define (cube x) (* x x x))
```

Tutorial exercise 3: Do exercise 1.4 of the text.

Tutorial exercise 4: Do exercise 1.6 of the text.

3. Programming assignment: Procedures and pictures

Now that you have gained some experience with Scheme, you are ready to work on the programming assignment. When you are finished with this section write up and hand in the numbered computer exercises below. Take a look on the web site, in the problem sets section, at the information labeled by "How should I write up my problem set?" to see what we are looking for.

Shading patterns

Scheme comes with many built-in primitive procedures, including simple numeric procedures such as `sin` and `cos` which take as argument a number in radians, and return the sine and cosine of that argument respectively. One way to visualize numeric procedures is to graph them as functions of their arguments. For one-argument procedures, one can use ordinary graphs. For two-argument procedures, it is often convenient to display the values as pictures. That is, we can let $b(x, y)$ be interpreted as a brightness value, and then create a two-dimensional display showing the value of b as a brightness pattern.

In this problem set, we'll play with these two-dimensional graphs.[4] We have provided you with some utilities. When you load this problem set (using `M-x load-problem-set`), the system will create three graphics display windows on your screen and give them the names `g1, g2` and `g3`. Each window is 128 wide and 128 high: the points in the window have integer x and y coordinates that range between 0 and 127. (These are "logical coordinates"—the physical size of the window in pixels may be different from 128×128.)

You can display pictures with the `picture-display` procedure, which takes two arguments: a window and a picture. `Picture-display` finds the minimum and maximum values in the picture, scales those values so that the minimum value appears as black and the maximum as white, and then displays the result in the window. You can supply two additional optional arguments to `picture-display` that set the minimum and maximum display values: any value in the picture below the specified minimum will be displayed as black; any value in the picture above the specified maximum will be displayed as white.

One way to create a picture is as the "graph" of a two-argument function, using

```
 (procedure->picture x y b)
```

Here `x` and `y` specify the width and height of the picture to be created. `x` and `y` must be integers, and `b` must be a function of two arguments, whose values will be used to create the picture. For example, evaluating

```
(picture-display g1 (procedure->picture 128 128 +))
```

will fill window `g1` with a shaded gray pattern whose brightness at each point (x, y) is $x + y$. Thus, the brightness increases uniformly as you move diagonally from black near $(0, 0)$ to white near $(128, 128)$.

Computer exercise 1: Draw a shaded pattern (a) that looks like the one above, only upside down; (b) where the brightness increases uniformly as you move horizontally from left to right; (c) where the brightness increases uniformly as you move outwards from the center of the window. Turn in the expressions you evaluated to obtain each pattern. There are lots of different correct answers to this problem.

Sinusoidal gratings

Creating a simple cosine grating: Use a procedure of two arguments whose returned value is the cosine of the second argument, transform that to a picture, and display the result:

```
(define pic1 (procedure->picture 128 128 (lambda (x y) (cos y))))

(picture-display g1 pic1)
```

[4] This problem set was developed by Eric Grimson and modified by Hal Abelson. The image display code was designed and implemented by Daniel Coore.

Note on naming pictures: Whenever you create a picture, you use up memory space. If you give the picture a name using `define`, that space cannot be reused, and if you create and name too many pictures, you may fill up Scheme's memory. It is a good idea to only use a few names for pictures and reuse these names as needed. For example, we could just use `pic1, pic2` and `pic3`.

Computer exercise 2: The code above creates a cosine grating of a particular width. To change the width, use $\cos(fy)$ rather than $\cos y$, where f specifies the frequency of the cosine wave. Define a procedure called `cosine-y-grate` that takes f as an argument and returns the appropriate cosine grating. If your procedure is defined correctly, you ought to be able to evaluate:

```
(picture-display g2 (cosine-y-grate .3))
```

to draw a wide-spaced grating in window `g2`. Turn in your procedure definition.

Computer exercise 3: Define a procedure `cosine-x-grate` like the one for exercise 2, but where the bands run vertically.

Computer exercise 4: Define a procedure like the one for exercise 2, but where the bands in the grating run diagonally. (Hint: Instead of using the cosine of x or y, use the cosine of ….)

Operations on pictures

We've provided a procedure called `picture-map` that enables you to operate on pictures and to combine pictures. You use it as follows:

```
(picture-map proc picture1 picture2 ... picturen)
```

Here `proc` is some procedure of n arguments, and `picture1, picture2, ..., picturen` are n pictures, all of the same size. The result is a new picture, where each value in the picture is the result of applying `proc` to the corresponding values of the n pictures. For example, if `pic` is a picture, then

```
(picture-map (lambda (val) (- val)) pic)
```

returns a picture whose values are the negatives of the values of `pic`; and if `pic1` and `pic2` are pictures, then

```
(picture-map + pic1 pic2)
```

returns a picture whose value at each point (x, y) is the sum of the corresponding values of `pic1` and `pic2` (i.e., the superposition of the two pictures).

`Picture-map` illustrates an important concept called *operating on aggregates*, namely, the idea of expressing operations directly in terms of compound data structures, rather than in terms of the individual components of those structures. We'll look at this idea in detail later in the semester, starting with lecture on February 24.

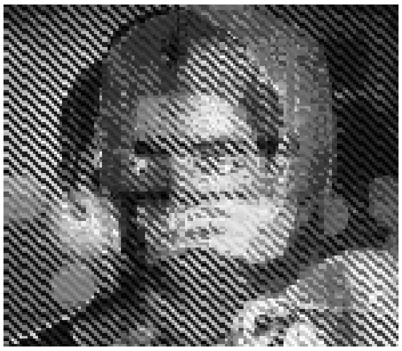

Figure 1: MIT's ideal Director of Freshman Housing

Computer exercise 5: Define a procedure that takes two arguments f_1 and f_2 and returns a picture whose value at (x, y) is $\cos f_1 x \cos f_2 y$. Define the procedure two different ways: (a) directly in terms of the point values, using `procedure->picture`; (b) using `picture-map` together with your answers to exercises 2 and 3. Be sure to draw the pictures produced by the two methods to confirm that they give the same result.

Predefined pictures

A second way of getting pictures into your system, rather than constructing them from numeric functions, is to load some previously stored pictures. If you evaluate:

```
(define pic1 (pgm-file->picture "fovnder.pgm"))
```

you'll get a picture of William Barton Rogers, the beloved FOVNDER and first president of MIT. He looks pretty dour, doesn't he? Maybe you can improve his disposition by combining him with some of your sinusoidal gratings from the previous exercises. Try this.

All mixed up

We're sure you've heard the rumors floating around campus that, despite repeated denials, the MIT administration has a secret plan to require all freshmen to live on campus beginning next fall, which they will announce later this spring. The 6.001 staff has a mole in the office of the Dean of Undergraduate Education, and—the rumors may very well be true! According to our information, Dean Rozz Williams is about to appoint a new Director of Freshman Housing. We've even obtained a picture that her office has constructed of the ideal candidate. The picture appears in figure 1, and you can get a copy of it by evaluating

```
(define pic1 (pgm-file->picture "mix-sp98.pgm"))
```

The word is that the new Freshman Housing Director must be a perfect blend—someone who can really empathize with the incoming students, and yet be tough enough to handle all the flack. You can see, however, that the picture is still foggy (perhaps a reflection of the state of the housing policy?). To decode the picture you need to understand how it was made.

Consider two functions $a(x, y)$ and $b(x, y)$. We can create a combined function known as a *convex blend* by using

$$c(x, y) = ra(x, y) + (1 - r)b(x, y).$$

Typically r is a constant, but we can make it a function of x and y as well:

$$c(x, y) = r(x, y)a(x, y) + (1 - r(x, y))b(x, y).$$

In this way, we can blend two pictures a and b to create a new picture c.

In general, knowing the values of $r(x, y)$ and $c(x, y)$ is not enough to allow us to recover $a(x, y)$ or $b(x, y)$. But there are particular choices for r for which we can recover the original unblended functions. Suppose we choose r to be a sinusoidal grating

$$r(x, y) = \frac{1}{2}(1 + \cos ft(x, y))$$

where f is a frequency and t is some function of x and y. We can take advantage of the structure of r to approximately recover the original unblended functions.

In particular, suppose we want to recover as much of a as we can. Notice that r takes on values between 0 and 1. At places where r is close to 0, there is not much of a in c (assuming that in general a and b have roughly similar ranges of values). On the other hand, when r is close to 1, then c is mostly a. So if we could find a way to multiply points in c (which is what we have our hands on) with values near 1 when r is near 1 and with values near 0 when r is near zero, then we could get back a partial approximation to a. But of course, this suggests that we just multiply the values of c by r itself.

To verify this, write a procedure that prints the values of

$$\begin{aligned} r &= \frac{1}{2}(1 + \cos u) \\ r^2 &= \frac{1}{4}(1 + \cos u)^2 \\ r(1 - r) &= \frac{1}{4}\sin^2 u \end{aligned}$$

for some sampled values of u between 0 and 2π. You should observe that when r is near 1, then r^2 is near 1 and $r(1 - r)$ is near 0, and when r is near 0, both r^2 and $r(1 - r)$ are near 0.

As a consequence, if we can find the underlying sinusoidal grating used to mix the two pictures, then by multiplying the mixture by the grating, we should get back a result that is approximately one of the original pictures: at points where c is mostly a we will get approximately a as a result, and in places where c is mostly b, we will get values near 0 as a result (i.e. a black picture). If we subtract that approximation from c we should get the other picture b.

Computer exercise 6: Extract the two personalities who blend to form the ideal freshman housing director, and see if you can recognize who they are. In order to do the extraction, you'll need to identify the sinusoidal grating that was used to construct the mixture. As hint, we'll tell you that it really does use a grating $r(x, y)$ of the form described above, where $t(x, y)$ is an appropriate function (which you ought to be able to guess) and the frequency f is a small integer multiple of 1/2. (Note that this grating is not exactly the gratings you created in the exercises above, since you need to add one to the cosine and divide by 2.) Turn in a two or three sentence description of how you performed the extraction together with any procedures you wrote to help you in the course of doing so.[5]

[5] If you're working in the 6.001 lab: The screens in the lab can be a bit dark on these pictures. To brighten things up, try using `picture-display` with the extra optional arguments 0 and 100, which will scale the brightness so that 100 is white. Also try using 100 to 200.

Turn in answers to the following questions along with your answers to the questions in the problem set:

1. About how much time did you spend on this homework assignment? (Reading and preparing the assignment plus computer work.)
2. Which scheme system(s) did you use to do this assignment (for example: 6.001 lab, your own NT machine, your own Win95 machine, your own Linux machine)?
3. We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).
   - If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the text (such as some of the archives), please indicate your consultants' names and your references. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.
   - Otherwise, write "I worked alone using only the reference materials," and sign your statement.

4. The Debugging Tools

This next part of the problem set is a self-teaching exercise. There is nothing to turn in, and you do not need to do it right after finishing part 3. We suggest that you take a half hour to go through this before you work on the next problem set.

During the semester, you will often need to debug programs. This section contains an exercise to acquaint you with some of the features of Scheme to aid in debugging. Learning to use the debugging features will save you much grief on later problem sets. Additional information about the debugger can be found in *Don't Panic*, and by typing `?` in the debugger.

The debugger

Use the Edwin `M-x load-problem-set` command to again load the code for problem set 1 (and ignore the graphics display windows). 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))))
```

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 hit an error, to quickly type `Q`, often without even reading the error message. Then they stare at their code in the editor trying to see what the bug is. Indeed, the example here 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 allows you to grovel around examining 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 bottom buffer right now is empty, and 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 b)
    R0  (+ (p2 a) (p2 b))
    R1  (p3 x y)
 S2  (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 February 10. 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.[6]

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  (p3 x y)
```

The bottom buffer should now look like this:

```
                         SUBPROBLEM LEVEL: 2
Expression (from stack):
 Subproblem being executed highlighted.
    (+ (p2 x y) (p3 x y))
----------------------------------------------------------------------
ENVIRONMENT named: (student)
    has 79 bindings

==> 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 highlighted. 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 semester, 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). For example, try evaluating `(+ x y)`, and notice that you can do this, even though these values of `x` and `y` are local to this activation of `P1`.

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 contain various bits of compiled code. 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.

[6]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.

Experiment 1: More debugging The code you loaded for problem set 1 also defined three other procedures, called `fold`, `spindle`, and `mutilate`. One of these procedures contains an error. Evaluate the expression `(fold 1 2)`. What is the error? How should the procedure be defined? Notice that you can examine the code for a procedure by using the `pp` command. For example, evaluating the expression `(pp fold)` will print the definition of `fold`.

Experiment 2: Still more debugging The code you loaded also contains a buggy definition of a procedure meant to compute the factorials of positive integers: $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$. Evaluate the expression `(fact 5)` (which is supposed to return 120). Use the debugger to find the bug, and correct the definition.

The stepper

The stepper is another useful debugging tool that you should become acquainted with. Go into the Scheme buffer and type the expression `(+ (* 3 4) (* 5 6))`, but instead of evaluating it with `c-X c-E`, type `M-s`. The screen will split to show two windows. The top is your Scheme buffer; the bottom is the *Stepper buffer*, which right now should read,

```
(+ (* 3 4) (* 5 6)) => ;waiting
```

"Waiting" means that it's waiting for you to tell it to go on, which you do by pressing the space bar. You'll see Scheme start to work on the subexpression `(* 5 6)`. Continuing to press the space bar will show each element in the evaluation. The various "waiting" tokens will be replaced by the values as the evaluation proceeds. At the end, you should be up at the top of the window again, with 42 shown as the value of the call to the stepper. At this point, you can get out of the stepper by moving back to the Scheme buffer.

Try stepping this same expression a few times until you see clearly what is going on. Rather than always hitting the space bar, there are a couple of other stepper commands you can try (press `?` to see them listed):

- `o` — step over the current expression: Get the value of the current expression without stepping through the details.
- `c` — contract: After you've stepped through an expression, hide the details, showing only the result.
- `e` — expand: Undo the contraction.

One more experiment: Try stepping through the evaluation of `(p1 1 2)`, which is the expression you used above to learn about the debugger. You will find that when you get to the evaluation of the expression that produces the error, you'll see `#[unspecified-return-value]` as the result—this should give you a hint that something has gone wrong.

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.

Debugging can be frustrating. The debugging tools are your friends. Call on them regularly. People often wish that they never encountered bugs, and you'll sometimes hear "theorists" of computer science teaching to claim that if you plan your programs well, you won't have bugs. We wonder if these "theorists" have ever written any programs:

Without erring, and transcending our error, we, as sometimes suggested by the Socratic irony, simply cannot become wise....Error is not a mere accident of untrained intellect, but a necessary stage or feature or moment ...—Josiah Royce *Lectures on Modern Idealism*

What is most harmful is trying to preserve oneself from errors.— Hegel