MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Department of Electrical Engineering and Computer Science

6.001--Structure and Interpretation of Computer Programs

Fall Semester, 2002

Project 0

1. Issued: On Week 1 / Day 3
2. To Be Completed: On Week 2 / Day 5
3. Reading:
    1. Read ``6.001 General Information" found in the syllabus. Be especially sure to read the 6.001 policy on collaborative work, and to read the detailed information on collaboration on the course web page.
    2. Scan the 6.001 course site on the web. All handouts (including this one) can be found there. The web page also contains announcements, Scheme software, projects, documentation, advice on where to get help, and other useful information. You should make a habit of looking at this page at least once a week during the semester.
    3. Textbook reading: In general, the lectures will assume that you've read the appropriate sections of the text **before** coming to or listening to lecture.

The purpose of Project 0 is to familiarize you with the Scheme programming environment and the resources available to you. The format of this project is different from the more substantive ones you will see later in the term. Those projects focus on extended experience in creating and testing code, and integrating such code with existing computational frameworks. As a consequence, those projects will take a significant amount of time to complete. This project is really just a warm up, and is intended to get you up to speed on the mechanics of using Scheme and the course resources. We expect that this project should not take much time, and while we understand that some of the things we ask you to do seem fairly obvious, we want to make sure you get the mechanics down right. Don't worry; the later projects and problem sets will involve much more extensive thinking and coding!

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 in the Tools section.

You'll see there that the 6.001 Scheme system runs in the 6.001 Lab on Linux, and on your own machine if you are running Athena-Linux, GNU/Linux, Windows 95, NT, 2000 or XP. We do not support Macintoshes.

Furthermore, if you use Athena, you can only run Scheme from an IBM or a Dell computer. Finally, you cannot run Scheme from a dialup Athena machine.. 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. The 6.001 lab is probably the best place to work if you want help, however, since that is staffed by knowledgeable and friendly Lab Assistants.

Starting Scheme

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

1. *In the 6.001 lab ...* A good way to get started learning Scheme is to do this project 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 in a similar fashion to that in the lab. The 6.001 machines are on the right side as you enter the lab, the left side belongs to 6.004. Both sets of machines use Athena logins:
    1. *Athena Linux:* Find a free lab computer running an Athena login screen, and log in with your Athena username and password. In this case, you will have access to your usual athena home directory and customizations. To start Scheme and Edwin type

    ```
    2.
    3.    add 6.001
    4.    6001-scheme
    ```
2. *On your home computer...* Follow the instructions on the Web page for downloading and installing Scheme (either Linux or Windows versions). *Important:* The version of Scheme we are using is Scheme 7.5.1. If you have an earlier version, it is important that you get the new one. Once you've installed the Scheme system you should be able to simply start it and work through this problem set. 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. (For this project, there is no code to download.)

Learning to use Edwin

When you start Scheme (either by using the commands listed above for Athena Linux, or by clicking on the appropriate 6.001 icon on your home machine if you installed Scheme on it) you are interacting with a text-edinting systems called *Edwin*, which is a Scheme implementation of the Emacs text editor. Edwin is virtually identical to Emacs. Even if you are familiar with Emacs, you will find it helpful to 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 then type `h`. Then release the CTRL key before typing `t`.) You will probably get bored before you finish the tutorial, 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 and projects. To get out of the tutorial, type `C-x k`, which will kill the buffer.

General Emacs commands

6.001 requires you to use Emacs and a few Edwin specific commands; you can see many of this by typing `C-h m` at the top level of Edwin. Below we discuss some of the commands you are likely to use as you interact with Scheme and Edwin.

Files and Buffers

Edwin allows you to read and edit existing files (`C-x C-f filename`), and create and write files (`C-x C-w filename` or `C-x C-s`). The text within these files are contained in Edwin objects called ``buffers''. The name of the current buffer can be found at the base of the Edwin window. Notice that Edwin starts up in a buffer called `*scheme*`. What distinguishes the Scheme buffer from other buffers is the fact that connected to this buffer is a Scheme interpreter (more about how to activate the interpreter later!)

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 working on. 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`). Choose a filename that ends in `.scm` for your code buffer, and then type `C-x C-f` *filename* (for example, `C-x C-f myproj0work.scm`). The half of the screen your cursor is in will now be a buffer for this new file. You can type in this buffer and evaluate expressions in this buffer(more on this in the next section!). The results of expression evaluation will appear in the `*scheme*` buffer. If an error occurs during evaluation you must go to the `*scheme*` buffer to deal with the error.

In addition to the `*scheme*` buffer, Edwin also provides a special read-only buffer called the `*transcript*` buffer which is automagically maintained and keeps a history of your interactions with the Scheme evaluator. You can extract pieces of this buffer to document examples of your work for projects (see the `C-space`, `M-w`, and `C-y` commands). To go to this buffer type `C-x b *transcript*`. You can go back to any buffer with the `C-x b` command.

Scheme Expression Evaluation

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.

While in the `*scheme*` buffer, 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 interpreter to read the expression immediately preceding the cursor, evaluate it (according to the kinds of rules described in lecture) and print out the result which is called the expression's ``value''. Try this.

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))
```

Scheme should respond with a message akin to the following:

```
;The object 7 is not applicable.
;Type D to debug error, Q to quit back to REP loop:
```

Type `q` for now. You will find that typing `d` invokes the debugger which is a useful tool. If you are interested in finding out more about the debugger now, look in the *Don't Panic* manual! There is also a section later on in this project that introduces the debugger.

We've already seen how to use `C-x C-e` to evaluate the expression preceding the cursor. There are also several other commands that allow you to evaluate expressions: `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.

2. Your Turn

There are several things that you need to do for this part: evaluate some simple Scheme expressions, use the provided tools to manipulate these expressions, and answer some documentation and administrative questions. The answers for all these parts should be **submitted electronically on the tutor**, using the `Submit Project Files` button. Remember that this is Project 0; when you are have completed all the work and saved it in a file, you should upload that file and submit it for Project 0.

2.1 Preparing your project handin

Create a new buffer (use the `C-x b` command, and give the prompt a new name). **At the top of the buffer, include your name, your TA's name, your section, the project number, and the part of the project you are answering** (yes, we know this seems silly, but it is amazing how many otherwise bright MIT students forget this!). Use this buffer as file (which you will need to save by using the `C-x C-s` command) into which you will insert your answers to the following parts.

Note that if you want to see your directories, you can use `(pwd)` inside a Scheme environment to list the name of the current directory to which you are connected, `(cd)` inside a Scheme environment to list the contents of the current directory, `(cd "foo")` to go to directory "foo", and `(cd "~foo")` to go to "foo" underneath the main directory.

Of course, since Edwin behaves like an Emacs, you can also use `M-x dired` to list the entries in your directory.

2.2 Expressions to Evaluate

Below is a sequence of Scheme expressions. Can you predict what the value of each expression would be when evaluated? Go ahead and type in and evaluate each expression in the order it is presented. **Extract the relevant parts out of the `*transcript*` buffer and copy this into your buffer that you made in part 2.1** (you will need to find the relevant Edwin/Emacs commands to do this; check out the `M-x set-mark` command, the `M-C-w` command, the `C-o` command and the `C-y` command).

```
(* 8 9)

(> 10 9.7)

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

(* (- 25 10)
   (+ 6 3))

+

(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)

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

In your handin, include an excerpt from the `*transcript*` buffer of these expressions and the resulting values of their evaluation. Also, include some comments that document your work. This is a good habit to get into NOW! For example:

```
;;;
;;; The following test cases explore the evaluation of simple
expressions.
;;;

(* 8 9)
;Value: 72
```

Note that in general, even when doing problem sets on the tutor, we **strongly suggest** that you use a separate Scheme environment to test your code, then cut and paste the expressions into the tutor window for submission.

2.3 Pretty printing

As you begin to write more complicated code, being able to read it becomes very important. To start building good habits, type the following simple expression into Scheme (with interspersed Edwin commands as shown):

```
(define abs C-j (lambda (a) C-j (if (> a 0) C-j a C-j (- a))))
```

Show a copy of how this actually appears in your buffer. Do the same thing, but in place of each `C-j`, use the `Enter` key followed by the `Tab` key. What difference is there in the result?

2.4 Real printing

If you want to print your a hardcopy of your work (at the end of this project), you will need commands to do this. These commands are issued from an interface to the operating system. When you logged in, an `xterm` window opened up. Inside that window, you can issue commands for printing. To see what the printer queue looks like, use `lpq` (this automatically selects the first available printer among those physically in the lab).

To print, assume that you have saved your work in `~/u6001/work/project01.scm`. Then go to `xterm` and type the following

```
cd ~/u6001/work   (to connect to this directory)
ls                (make sure the file you want is in fact there)
lpr project0.scm
lpq               (to chck the status of your print job)
```

To remove a job, use `lprm job-number`.

If you are working on Athena (as opposed to in the 6.001 lab), you may use `cview printers` to get a list of available printers. This can be done from an `xterm` window, or you can invoke `M-x shell` inside Edwin/Emacs, which will create a shell window within the editor, to which you can type Linux commands. To queue a print job, use

```
lpr -P printername filename
```

or if you prefer

```
lpr -P printername2 filename
```

which will print things two-sided rather than single-sided.

2.5 Documentation and Administrative Questions

Explore the 6.001 webpages to find the answers to the following questions:

1. According to the *Don't Panic* manual, how do you invoke the stepper? What is the difference between the stepper and the debugger?

2. According to the *Guide to MIT Scheme*, which of the words, in the scheme expressions you evaluated as part of section 2.2 above, are "special forms"?

3. After referring to the course policy on collaboration, describe the course policy on the use of "bibles".

4. List three people with whom you could collaborate on projects this term. You are not required to actively collaborate with these people, we just want you to start thinking about what collaboration means in this course.

5. Locate the list of announcements for the class. What does a particular entry say about recitation attendance?

6. What are the three methods for controlling complexity described in the learning objectives section of the course objectives and outcomes? List one example from each category.

7. What does the MIT Scheme Reference Manual say about treatment of upper and lower case in expressions?

8. What are the Edwin commands for creating a new file, and for saving a file? What is the difference between the *scheme* buffer and the `*transcript*` buffer?

3. Debugging Tools

During the semester, you will often need to debug programs. This section will acquaint you with the debugger. Additional information can be found in *Don't Panic*, and by typing `?` in the debugger.

The Debugger

Type the following two *define* statements into your `*scheme*` buffer, and evaluate each of them.

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

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

Next, type and 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 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 some helpful information about the error can be produced.

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 examine 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 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 6 p2] has been called with 1
argument;
  it requires exactly 2 arguments.



>S0  (#[compound-procedure 6 p2] 1)
    R0  (p2 x)
 S1  (p2 x)
    R0  (+ (p2 x y) (p2 x))
    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. 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) (p2 x))`. Above that, we see that in order to evaluate this expression the interpreter chose to work on the subproblem `(p2 x)` which produces the error because two arguments are required.

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

```
>S1  (p2 x)
```

The bottom buffer should now look like this:

```
                         SUBPROBLEM LEVEL: 1
Expression (from stack):
 Subproblem being executed highlighted.
    (+ (p2 x y) (p2 x))
------------------------------------------------------------------------
ENVIRONMENT named: (student)
    has 68 bindings (see editor variable environment-package-limit)

==> 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 to 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.

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):

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

If you were to 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 `#[some-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!

Submission

Once you have completed all the parts of this introductory project, your file should be **submitted electronically on the tutor**, using the `Submit Project Files` button. Remember that this is Project 0; when you are have completed all the work and saved it in a file, you should upload that file and submit it for Project 0.

Congratulations! You have reached the end of Project 0!