CS61A – Homework 4.2	**Kurt Meinz**
University of California, Berkeley	Summer 2005

Topic: Assignment, state, environments

Lectures: Wednesday July 16, Thursday July 17

Reading: Abelson & Sussman Sections 3.1, 3.2 (Pages 217-251) and
"Object-Oriented Programming—Below-the-line view" (in course reader)

This assignment gives you practice with procedures that have local state and with the environment model of evaluation. Do not use OOP; use regular Scheme. This assignment is due at **8 PM on Sunday, July 17**. Please put your answers to Questions 1-3 into a file `hw4-2.scm` and submit electronically. Question 4 asks you to draw an environment diagram. Please do this on a blank sheet of paper and turn it into the box labeled with your TA's name in 283 Soda before lecture on Monday. Don't forget to **write your name and login** on the paper.

Question 1. To *instrument* a procedure means to make it do something in addition to what it already does. For example, a procedure can be instrumented to keep statistics about itself, such as how many times it has been called.

A. Write a procedure `instrument` that takes a one-argument procedure f. It should return an *instrumented* version of f that keeps track of how many times it was called using a *local* counter. If the instrumented procedure is called with the special symbol `times-called`, it should return the number of times it has been invoked. If it's called with `reset`, the internal counter should be set to zero. Any other argument should be passed directly to f:

```
STk> (define i-square (instrument (lambda (x) (* x x))))
i-square
STk> (i-square 5)
25
STk> ((repeated i-square 3) 2)
256
STk> (i-square 'times-called)
4
STk> (i-square 'reset)
ok                                ;; return value up to you
STk> (i-square 'times-called)
0
```

B. We'd like to keep track of how many times `factorial` is called (including recursive calls). So we try:

```
STk> (define (factorial n)
       (if (= n 0)
           1
           (* n (factorial (- n 1)))))
factorial
STk> (define i-factorial (instrument factorial))
i-factorial
STk> (i-factorial 5)
120
STk> (i-factorial 'times-called)
1
```

Explain why `i-factorial` thinks it's only been called once, not five times. You may find it useful to draw an environment diagram, though you don't have to.

Question 2. Modify the `make-account` procedure on Page 223 to create password-protected accounts. You choose the password when you create an account; you must then supply that same password when you wish to withdraw or deposit:

```
STk> (define a1 (make-account 100 'this-is-my-password))
a1
STk> ((a1 'withdraw 'this-is-my-password) 40)
60
STk> ((a1 'withdraw 'this-is-not-my-password) 10)
Incorrect Password                              ;; print this and
ok                                              ;; return something
STk> ((a1 'deposit 'this-is-my-password) 10)
70
```

If an account is accessed three consecutive times with wrong password, display the following warning: "Do it again and I'll call the police". If an account is accessed more than three consecutive times with the wrong password, invoke this procedure (or your own variant):

```
(define (police)
   (display "Bad boys, bad boys\n")
   (display "Watcha gonna do whatcha gonna do when they come for you\n")
   (display "Bad boys, bad boys\n")
   (display "Watcha gonna do whatcha gonna do when they come for you\n")
   (display "...\n")
   (display "Nobody naw give you no break\n")
   (display "Police naw give you no break\n"))
```

(Lyrics from www.geocities.com/tvshowthemelyrics/CopsSong.html)

Question 3. Under the substitution model of evaluation, the *order* in which arguments were evaluated (e.g., left to right or right to left) didn't matter. With the introduction of assignment—or other side-effects, such as printing—the order in which expressions are evaluated matters a great deal; different results are possible if arguments are evaluated from left to right instead of right to left. Devise a way to test the order in which arguments are evaluated. Determine which way does `+`, `(lambda (x y z) (list x y z))` and `cons` evaluate their arguments in STk.

The fun continues on the next page.

Question 4. Draw an environment diagram for the following expressions. Also fill in the return value in each blank:

```
STk> (define make-counter
        (let ((total 0))
          (lambda ()
             (let ((count 0))
               (lambda ()
                 (set! count (+ 1 count))
                 (set! total (+ 1 total))
                 (list count total))))))
STk> (define c1 (make-counter))
STk> (c1)

_________________

STk> (c1)

_________________

STk> (define c2 (make-counter))
STk> (c2)

_________________

STk> (c1)

_________________
```

There is a program called EnvDraw that draws environment diagrams. There is no reason you shouldn't check your work using it. To run it, type `envdraw` at your shell (`%` is the shell prompt):

```
% envdraw
```

This will launch STk. From STk call the `envdraw` procedure:

```
STk> (envdraw)
okay
```

Now, any expression you evaluate at the STk prompt will be shown in the environment. To keep EnvDraw from drawing the entire diagram at once, turn on Stepping mode; when you want it to draw the next piece of the diagram, choose Step from the menu.

But wait! We did say *check* your work using EnvDraw, not let EnvDraw do the whole thing and copy. Environment diagrams are fundamental to what we do from now until the end of the semester. Putting in the time to understand them this week will make subsequent topics easier.

Lastly, it is useful to think of local state variables as either *class* variables or *instance* variables. If we treat `c1` and `c2` in the above expression as instances, which are the class and instance variables?