MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945/6.S081 Spring 2014
Problem Set 4

Issued: Wed. 5 March 2014 **Due: Wed. 12 March 2014**

Reading:
SICP, From Chapter 4: 4.1 and 4.2; (pp. 359--411)
en.wikipedia.org/wiki/Evaluation_strategy
www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR44
http://en.wikipedia.org/wiki/Fexpr
http://web.cs.wpi.edu/~jshutt/kernel.html

Code: load.scm, utils.scm, ghelper.scm,
syntax.scm, rtdata.scm, interp.scm, repl.scm,
general-procedures.scm, kons.scm
code is on the class web page... no reason to kill more trees.

A Language For Each Problem

One of the best ways to attack a problem is to make up a language in which the solution is easily expressed. This strategy is especially effective because if the language you make up is powerful enough, many problems that are similar to the one you are attacking will have easy-to-express solutions in the your language. It is especially effective if you start with a flexible mechanism. Here we get some practice working with a souped-up interpreter for a Lisp-like language.

We start with an extended version of Scheme similar to the ones described in SICP, Chapter 4. Without a good understanding of how the evaluator is structured it is very easy to become confused between the programs that the evaluator is interpreting, the procedures that implement the evaluator itself, and Scheme procedures called by the evaluator. You will need to study Chapter 4 through subsection 4.2.2 carefully in order to do this assignment.

The interpreters in the code that we will work with in this problem set are built on the generic operations infrastructure we developed in previous problem sets. Indeed, in these interpreters, unlike the ones in SICP, EVAL and APPLY are generic operations! That means that we may easily extend the types of expressons (by adding new handlers to EVAL) and the types of procedures (by adding new handlers to APPLY).

Before beginning work on this problem set you should carefully read the code in interp.scm.

Using the generic interpreter

Download the supplied code to a fresh directory in your computer.
Get a fresh Scheme system, and load the file load.scm:

```
   (load "<your-code-directory>/load")
```

Initialize the evaluator:

```
   (init)
```

You will get a prompt:

```
   eval>
```

You can enter an expression at the prompt:

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

   eval> (cube 3)
   27
```

The evaluator code we supplied does not have an error system of its own, so it reverts to the underlying Scheme error system. (Maybe an interesting little project? It is worth understanding how to make exception-handling systems!) If you get an error, clear the error with two control Cs and then continue the evaluator with "(go)" at the Scheme. If you redo "(init)" you will lose the definition of cube, because a new embedded environment will be made.

```
   eval> (cube a)
   ;Unbound variable a
   ;Quit!

   (go)

   eval> (cube 4)
   64

   eval> (define (fib n)
            (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
   fib

   eval> (fib 10)
   55
```

You can always get out of the generic evaluator and get back to the underlying Scheme system by quitting (with two control Cs).

One may become confused when working with an interpreter embedded in the underlying Scheme system. The code that you load into the underlying Scheme system is MIT Scheme code. Your definitions define symbols in the underlying system with values that are defined with respect to the underlying system. However, when you execute (init) or (go) you enter the embedded system. The mapping of the symbols to values (the environment) is private to the embedded system. However, symbols, like car, that you did not define and have values in the underlying Scheme system have those values in the embedded system as well.

To make changes to the embedded interpreter you must leave the embedded system and reenter the underlying Scheme system, by quitting (with two control Cs). You can then reenter the embedded interpreter using either (go), if you want to retain the definitions you made in the embedded system, or (init), if you want to start from scratch.

Problem 4.1: Warmup

In mathematical text a common abuse of notation is to identify a tuple of functions with a function that returns a tuple of values. For example, (written in Lisp prefix form)

```
 If   (cos 0.6) ==> 0.8253356149096783
 and  (sin 0.6) ==> 0.5646424733950354

 then we expect

 ((vector cos sin) 0.6) ==> #(0.8253356149096783 0.5646424733950354)
```

This requires an extension to APPLY so it can handle Scheme vectors as a kind of function. Make this extension; demonstrate it; show that it interoperates with more conventional code.

Notice that

```
eval> (map (lambda (x) (* x x)) '(1 2 3))
```

DOES NOT WORK, if the map in this expression refers to the map procedure from the underlying Scheme system.

However, if we redefine map for our interpreter it does work:

```
eval> (define (map f l)
        (if (null? l)
            '()
            (cons (f (car l)) (map f (cdr l)))))
map

eval> (map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
```

Problem 4.2: Interoperation problem

Why does it not work to use the underlying procedures that take procedural arguments, such as map? Explain. Outline a strategy to fix the problem. Implement your solution, if you are so inspired, BUT this is subtle to get right, so don't spend infinite time trying to make it work.

In Lisps, including Scheme, attempting to evaluate an unbound symbol is an unbound-variable error. However, in some algebraic processes it is sensible to allow an unbound symbol to be a self-evaluating object. For example, if we generically extend arithmetic to build algebraic expressions with symbolic values, it is sometimes useful to allow the following:

```
(+ (* 2 3) (* 4 5)) ==> 26

(+ (* a 3) (* 4 5)) ==> (+ (* a 3) 20)
```

In this case, the symbol "a" is unbound and self-evaluating. The operators "*" and "+" are extended to just build expressions when their arguments are not reducible to numbers.

Problem 4.3: Unbound-variable handling

A. Make a generic extension to EVAL to allow this kind of behavior. In order to make this work with the numerical primitives (+, *, -, /) it is necessary to extend their behavior as well. Note that these operators should be changed in the underlying Scheme environment. As we found in problem 4.2, the generic operator mechanism must be given handlers that work in the underlying Scheme system.

B. Also augment APPLY to allow literal functions, known only by their names: (+ (f 3) (* 4 5)) ==> (+ (f 3) 20)

These extensions to EVAL and APPLY are generally dangerous, because they hide real unbound-variable errors. Make them contingent on the value of a user-settable variable: ALLOW-SELF-EVALUATING-SYMBOLS.

Much more powerful extensions are available once we accept generic operations at this level. For example, we can allow procedures to have both strict and non-strict arguments.

   If you don't know what we are talking about here please read the article: http://en.wikipedia.org/wiki/Evaluation_strategy.

If you load the file general-procedures.scm into the underlying Scheme, after loading (with '(load "load")') the generic interpreter, you will find that there are extensions that allow us to define procedures with some formal parameters asking for the matching arguments to be lazy (or both lazy and memoized). Other undecorated parameters take their arguments strictly. These extensions make it relatively easy to play otherwise painful games. For example, we may define the UNLESS conditional as an ordinary procedure:

```
   ;Quit!

   (load "general-procedures" generic-evaluation-environment)
   ;Loading "general-procedures.scm"...
   ;Warning: Replacing top-level handler
   ;... done
   ;Value: #[compound-procedure 17 operator]


   (go)

   eval> (define unless
           (lambda (condition (usual lazy) (exception lazy))
             (if condition exception usual)))
```

We may use the usual define abbreviations (see syntax.scm):

```
   eval> (define (unless condition (usual lazy) (exception lazy))
           (if condition exception usual))
   unless

   eval> (define (ffib n)
           (unless (< n 2)
                   (+ (ffib (- n 1)) (ffib (- n 2)))
                   n))
   ffib

   eval> (ffib 10)
   55
```

Notice that UNLESS is declared to be strict in its first argument (the predicate) but nonstrict in the alternatives: neither alternative will be evaluated until it is necessary.

Additionally, if we include the file kons.scm we get a special form that is the non-strict memoized version of CONS. This immediately gives us the power of infinite streams:

```
;Quit!

(load "kons" generic-evaluation-environment)
;Loading "kons.scm"... done
;Value: #[compound-procedure 19 operator]

(go)

eval> (define (add-streams s1 s2)
        (kons (+ (car s1) (car s2))
              (add-streams (cdr s1) (cdr s2))))
add-streams

eval> (define (ref-stream stream n)
        (if (= n 0)
            (car stream)
            (ref-stream (cdr stream) (- n 1))))
ref-stream

eval> (define fibs
        (kons 0
              (kons 1
                    (add-streams (cdr fibs) fibs))))
fibs

eval> (ref-stream fibs 10)
55

eval> (ref-stream fibs 20)
6765

eval> (ref-stream fibs 30)
832040

eval> (ref-stream fibs 40)
102334155
```

Problem 4.4: Streams

a. The non-strict procedure KONS adds great power to the system. Notice that there is no need to make CAR or CDR different to obtain the use of streams. In a short paragraph explain why KONS is almost sufficient. It may be instructive to read an ancient paper by Friedman and Wise:
www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR44

b. Unfortunately, the addition of KONS does not, in itself, solve all stream problems. For example, the difficulty alluded to in SICP section 4.2.3 (p. 411) does not automatically dissipate. If we make the following definitions:

```
   (define (map-stream proc items)
     (kons (proc (car items))
           (map-stream proc (cdr items))))

   (define (scale-stream items factor)
     (map-stream (lambda (x) (* x factor))
                 items))

   (define (integral integrand initial-value dt)
     (define int
       (kons initial-value
             (add-streams (scale-stream integrand dt)
                          int)))
     int)

   (define (solve f y0 dt)
     (define y (integral dy y0 dt))
     (define dy (map-stream f y))
     y)
```

and then we try:

```
   (ref-stream (solve (lambda (x) x) 1 0.001) 1000)
```

we will get an error (try it!). Why? Explain the error. What other declarations should be made in these stream-procedure definitions to fix all such errors?

```
-------------
Problem 4.5: Why not?

a. The KONS special form is equivalent to a CONS with both arguments
lazy and memoized.  If the arguments were not memoized the computation
(ref-stream fibs 40) in Problem 4.3a above would take a very long
time.  Is there ever any advantage to not memoizing?  When might it
matter?

b. Why, given that CONS is a strict procedure imported from Scheme,
could we not have defined KONS simply as:

   (define (kons (a lazy memo) (d lazy memo))
     (cons a d))
?

c. More generally, the Lisp community has avoided changing CONS to be
KONS, as recommended by Friedman and Wise.  What potentially serious
problems are avoided by using CONS rather than KONS?  Assume that we
do not care about small constant factors in performance.
-------------


-------------
Problem 4.6: Your turn

Invent some fun, interesting construct that can easily be implemented
using generic EVAL/APPLY that would be rather painful without that
kind of generic support.  Show us the construct, the implementation,
and some illustrative examples.  Enjoy!
-------------
```