MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.5150/6.5151 Spring 2024
Problem Set 6

Issued: Wed. 20 March 2024 **Due: Fri. 5 April 2024**

Reading:

SDF Chapter 5 -- Evaluation: Sections 5.1 and 5.2

Some, perhaps useful background material
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

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. It may help to review SICP 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 sdf/generic-interpreter/interp.scm.

Using the generic interpreter

Get a fresh Scheme system, and load up the generic interpreter from SDF section 5.1.

```
(load "<yd>/sdf/manager/load")  ;<yd> = <your class directory>
(manage 'new 'generic-interpreter)
```

Initialize the evaluator:

```
(init)
```

You will get a prompt that looks like "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 (C-c) 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.

To Do

Exercise 5.3: Vectors of procedures (SDF pp. 248–249)

Exercise 5a: Macros (Not in book)

Making both eval and apply generic procedures (as g:eval and g:apply) gives us enormous power. We can add novel types of expression (as in COND and LET on SDF pages 241 and 242) and novel types of procedures (as in Exercise 5.3, above) with little work. But these methods of extending the language are a bit too powerful: changing the interpreter requires getting into the underlying system. If we have a compiler to some other language, like the native code of the machine, we also have to change it compatibly.

But most changes we might want are simple expression transformations, such as COND, which turns into a nest of IF expressions. The usual way to do this is to provide a "macro feature" that makes it easy for a programmer to write these transformations without going into the underlying system.

What we really want is a single new expression type, a "macro" that allows a user to write an expression-transformation at the source level easily. For example, on SDF pages 166 and 167 we show a macro for implementing a special syntax for a rule.

If the interpreter sees an expression that looks like:

```
(rule '(* (? b) (? a)) (and (expr<? a b) `(* ,a ,b)))
```

it is transformed ("expanded") into an expression that looks like:

```
(make-rule '(* (? b) (? a))
  (lambda (b a)
    (and (expr<? a b) `(* ,a ,b))))
```

This expression is evaluated in place of the one beginning "(rule".

We used a mechanism that defines this kind of source-level transformation in a "hygienic" way with heavy machinery that we do not want to explain. In this exercise you should just implement a "define-macro" special form for the simple generic interpreter that allows a user to write:

```
(define-macro (rule pattern handler-body)
  `(make-rule ,pattern
              (lambda ,(match:pattern-names pattern)
                ,handler-body)))
```

that will produce the required expansion and other similar source-level transformations. Do not try to make it "hygienic".

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", by:

```
(load "<your-directory>/sdf/non-strict-arguments/general-procedures")
```

in the Scheme that has the section 5.1 evaluator loaded, you will get the extensions for section 5.2 that allow you 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, as described in Section 5.2.

We can also reload the whole generic interpreter, with these extensions, by

```
(manage 'new 'non-strict-arguments)

(init)
```

The init will lose any definitions you have made in the previous exercises, so you will have to reload them. If you put them into a file named "<your directory>/foo.scm" you can reload them at the "eval> " prompt using load-library:

```
eval> (load-library "<yd>/foo.scm")
```

(We define this loader in "<yd>/sdf/generic-interpreter/repl.scm".)

But given a loaded, initialized, embedded interpreter we can

```
(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 consequent or the alternative: neither 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. It may be instructive to read an ancient paper by Friedman and Wise:

www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR44

The procedure KONS immediately gives us the power of infinite streams. We have supplied the stream library in the file kons-extra.scm. You may load this into the interpreted generalized evaluation system. (We provided a loader in repl.scm.)

```
eval> (load-library "<yd>/sdf/non-strict-arguments/kons.scm")

;;; Note: this will run for quite a while, because kons.scm
;;;  is a test program.  It should after a while put out:

;;;     2.716923932235896
;;;     done

eval> (define fibs
        (kons 0
              (kons 1
                    (add-streams (kdr 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
```

To Do

Exercise 5.9: Why not kons? (SDF p. 257)

Exercise 5.10: Restricted parameters (SDF pp. 257–258)