MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Problem Set 8

Scheme Evaluators

Issued: Tuesday, October 29, 1996

Due: Friday, November 8, 1996

Tutorial preparation for: Week of November 4, 1996

Reading: Chapter 4 through section 4.2.2; attached code files `meval.scm`, `analyze.scm`, `evdata.scm`, `syntax.scm`, `ps8defs.scm`.

You will be working with two of the evaluators as described in Chapter 4 of the notes. If you don't have a good understanding of how the evaluators are 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 have carefully studied Chapter 4 through subsection 4.2.2 in order to do this assignment.

Because the main meta-circular evaluator will be presented in lecture on October 31, and the analyze evaluator will be presented in lecture on November 5, we have designed this problem set in two parts, one on each evaluator. We strongly encourage you to start the problem set as soon as you can, and not to wait until the analyze evaluator is covered in lecture before starting work on the parts of the problem set dealing with meta-circular evaluator.

Tutorial exercises Do exercises 4.6, 4.7, and 4.11.

Using the Evaluators

The code for problem set 8 includes these files:

- `meval.scm` is the metacircular evaluator described in section 4.1.1 of the notes. It has been extended to allow declarations of lazy procedure parameters, as described in section 4.2 and exercise 4.31. In order to avoid confusing the `eval` and `apply` of this evaluator with the `eval` and `apply` of the underlying Scheme system, we have renamed these procedures `meval` and `m-apply`.

- `analyze.scm` is the `analyze` evaluator of section 4.1.7, extended to allow declarations of lazy procedure parameters, as described in section 4.2 and exercise 4.31.

- `evdata.scm` contains the procedures that define the evaluator's data structures, as in section 4.1.3. In order to interface the evaluator to the underlying Scheme system, we have made the modification outlined in the text, namely, any variable that is not found in the evaluator's global environment will be looked up in the underlying Scheme; Scheme procedures found in this way will treated by the evaluator as primitive procedures. Programs being executed by the metacircular evaluator can therefore make use of any procedure (primitive or compound) which may be defined in your Scheme environment.

- `syntax.scm` contains the procedures that define the syntax of expressions, as described in section 4.1.2.

You'll be switching back and forth between both of these evaluators. The two evaluators use the same implementation of environment data structures. This arrangement requires that you reinitialize the evaluator's environment when you switch from one to the other. Here's how to manage this:

- To start up, do

  ```
  M-x load-problem-set: 8
  ```

  in Edwin. This loads the files `syntax.scm`, `evdata.scm`, `meval.scm`, and `analyze.scm`. Then evaluate `(start-meval)` in Scheme; this starts the read-eval-print loop for the meta-circular evaluator with a freshly initialized global environment.

- To evaluate an expression, you may type the expression into the `*scheme*` buffer followed by `ctrl-x ctrl-e`. (Don't use `M-z` to evaluate the expression in the `*scheme*` buffer—the presence of the evaluator prompts confuses the `M-z` mechanism. But `M-z` works fine for sending expressions to the `*scheme*` buffer from other Scheme-mode buffers.)

  We've arranged that each read-eval-print loop identifies itself by its input and output prompts. For example,

  ```
  MEVAL=> (+ 3 4)
  ;;M-value: 7
  ```

  shows an interaction with the `meval` evaluator.

- You should keep in a separate file any procedure definitions you want to install in an evaluator. If your Edwin Scheme buffer is running the read-eval-print loop of an evaluator, you can then visit this definitions file and type `M-o` to enter the definitions into the evaluator. To get you started, we have included an alternative implementation of lazy lists in the buffer `ps8defs.scm`.

- You can interrupt an evaluator by typing `ctrl-c ctrl-c`. To restart it with the recent definitions still intact, evaluate `(eval-loop)`.

- To start a read-eval-print loop for the analyzing evaluator in a fresh global environment, return to Scheme and evaluate `(start-analyze)`.

- The evaluators you are working with do not include error systems. If you hit an error you will bounce back into ordinary Scheme. You can restart the current evaluator, with its global environment still intact, by evaluating `(eval-loop)`.
- The variable `current-evaluator` is defined as an alias for the current evaluator in each of the files `meval.scm` and `analyze.scm`. This should ensure that `current-evaluator` will be bound to the most recently started evaluator at all times during your lab session.
- It can be instructive to trace `current-evaluator`, `m-apply`, and/or `exapply` during evaluator executions. (You will also probably need to do this while debugging your code for this assignment.)
- Since environments are generally complex, circular list structures, we have set Scheme's printer so that it will not go into an infinite loop when asked to print a circular list. This was done by

  ```
  (set! *unparser-list-depth-limit* 7)
  (set! *unparser-list-breadth-limit* 10)
  ```

  at the end of the file `evdata.scm`. You may want to alter the values of these limits to vary how much list structure will be printed as output.

Lab exercises

As usual, for all the lab exercises below, you should turn in listings of any procedures you define in your solutions, as well as sample evaluations demonstrating their correct behavior.

Lab exercise 1: Start the `meval` evaluator and evaluate a few simple expressions and definitions. It's a good idea to make an intentional error and practice restarting the read-eval-print loop. Notice that we have already installed `let` in the evaluator, so you can use this in your programs. Turn in a transcript of your examples.

Adding Language Constructs

Pre-Lab exercise 2A: Lazy parameter declarations are already supported by both the `meval` and `analyze` evaluators supplied to you. Suppose we are given some *fixed* $n \geq 0$. With lazy parameters, it is possible to define within an evaluator an n-argument *procedure*, `or-proc`, that yields the same results as the Scheme special form `or` when it is used with n subforms. That is, `(or` E_1 ...E_n`)` in Scheme, and `(or-proc` E_1 ...E_n`)` in an evaluator, should produce exactly the same results (including side-effects) for any expressions $E_1 \ldots E_n$. Outline how to define the n-argument `or-proc` within our evaluators.

Lab exercise 2B: Evaluate the definition of a three-argument `or-proc` in the `meval`.

Lab exercise 3: Since our evaluators do not yet have the ability to define procedures with varying numbers of arguments, we can't add the general `or` construct with the approach of Exercise 2. So instead, add `or` as a special form to the meta-circular evaluator.

We can also extend the evaluators to handle Scheme's convention for defining procedures with varying numbers of arguments. For example, an abstraction of the form `(lambda (x y .  z) ...)` defines a procedure of two or more arguments, in which the value of the first argument would be bound to `x`, the value of the second to `y`, and a list consisting of the values of any further arguments would be bound to `z`. If, instead of `z`, we had `(w lazy)`, then a list consisting of the *delayed* values of any further arguments would be bound to `w`. Similarly, if `(w lazy-memo)` appears instead of z, this indicates that a list consisting of the memoized delayed values of any further arguments would be bound to `w`.

We accomplish the extension by modifying the procedure `process-arg-procs` in the file `evdata.scm`[1]. As written, `(process-arg-procs params aprocs env)` expects `params` and `aprocs` to be *lists* of equal length, and it returns a list of parameter values of the same length. If `params` is not a list of parameter declarations, but instead ends with a dotted pair of parameters, then we want `process-arg-procs` to return a list of values in which the last value is itself a list of all the "extra" values, possibly all delayed or all memoized depending on whether the final parameter is declared to be lazy or memoized. For example,

```
((lambda (x y . (w lazy)) (list x y (car w)))
 1  2  (if #t 3 4)  (print "this should not print"))
```

will return `(1 2 3)`, without printing anything. This is because in the application, the variable `x` will be bound to 1, variable `y` will be bound to 2, and variable `w` will be bound to the list consisting of the delayed values of

```
(if #t 3 4)
```

and

```
(print "this should not print").
```

The first delayed value gets forced when the primitive procedure `list` is applied to it. The second delayed value is not ever forced, so no printing occurs.

So we revise `process-arg-procs` to reformat its arguments if necessary, after which it proceeds as before:

```
(define (process-arg-procs params aprocs env)
  (let ((params (undot params))
        (aprocs (matchup-args params aprocs)))
    (map (lambda (param aproc)
           (cond ((variable? param) (aproc env))
                 ((lazy? param) (delay-it aproc env))
                 ((memo? param) (delay-it-memo aproc env))
                 (else (error "Unknown declaration" param))))
         params
         aprocs)))
```

[1] The procedure `parameter-names` in `syntax.scm` must also be written to handle the dotted-pair form of parameters, but we have done this already.

The procedure `matchup-args` has the task of taking the list of arguments and returning the list with all the "extra" arguments put into a single list at the end. But at this point each argument in the input list is actually packaged as an "argument procedure." The argument procedure will yield the argument value when it is applied to the environment. So `matchup-args` actually has to create a similar package which yields the list of extra argument values when it is applied to the environment.

```
(define (matchup-args params aprocs)
  (cond
   ((null? params)
    (if (null? aprocs)
        '()
        (error "matchup-args: too many args" aprocs)))
   ((or (variable? params) (declaration? params))
    (list (make-list-package params aprocs)))
   ((null? aprocs)
    (error "matchup-args: too few args" params))
   (else (cons
          (car aprocs)
          (matchup-args (cdr params) (cdr aprocs))))))
(define (make-list-package formal aprocs)
 (lambda (env)
       (map
        (cond ((variable? formal)
               (lambda (aproc) (aproc env)))
              ((lazy? formal)
               (lambda (aproc) (delay-it aproc env)))
              ((memo? formal)
               (lambda (aproc) (delay-it-memo aproc env))))
        aprocs)))

(define (undot params)
  <blob6a>)
```

Lab exercise 4A: Complete the definition of `<blob6a>`, insert these definitions in the file `evdata.scm` and reload the file into Scheme. You will find a copy of the above definitions in buffer `ps8-ans.scm`.

Lab exercise 4B: Now we can add, as a procedure, a genuine `or` construct to the evaluators. Namely,

```
(define or (lambda (l lazy) (or-lproc l)))
```

where `or-lproc` is a procedure that takes as its argument a single list[2] of delayed values. Give the

[2] Both `lazy` and `lazy-memo` are now specified to be keywords. This prevents misreading `(lambda (l lazy) ...)` as a procedure with two formals, `l` and `lazy`. Rather, it is a procedure which may be applied to zero or more arguments—all of which will be delayed.

definition of or-lproc.

Even with lazy and varying numbers of parameters, not all language facilities can be defined as Scheme procedures. An example is the special form (while <var> <init> <test> <inc> <body>) where <var> is a variable, the value of <init> is an initial value for the variable, <test> is an expression that will be evaluated to determine termination of the loop, and <inc> is anb expression that will be evaluated to determine the next value of the variable. Evaluation of this expression causes <body> to be repeatedly evaluated as <var> is successively incremented until the test no longer holds true. More precisely, (while <var> <init> <test> <inc> <body>) is syntactic sugar for[3]

```
(let ()
  (define (loop <var>)
    (if <test>
(begin
  <body>
  (loop <inc>))
the-unspecified-value))
  (loop <init>))
```

For example, we might evaluate something like (while i 0 (< i 10) (+ i 1) <body>).

Lab exercise 5A: Add the while construct as a special form (*not* as syntactic sugar like let or cond) to the meta-circular evaluator.

Lab exercise 5B: Show an example of evaluation of a while loop.

The Analyzing Evaluator

Lab exercise 6A: Start the meval evaluator and evaluate a few simple expressions and definitions.

Lab exercise 6B: Now start the analyzing evaluator. You can tell that you are typing at the analyzing evaluator because the prompt will be AEVAL=> rather than MEVAL=>. Evaluate the same simple expressions and definitions you did in Exercise 6A.

Lab exercise 6C: Start the meta-circular evaluator again. Now trace evaluation of the application of some simple procedure to some argument.

Ben Bitdiddle thinks it is silly to reinitialize the environment every time he switches from one evaluator to the other. So, after having run the meta-circular evaluator, he revises the start-analyze procedure so it does not reinitialize the global environment:

[3] We're assuming that the local variable loop is not eq? to <var>, <inc>, <init> or any variable free in <body>.

```
(define (start-analyze)
  (set! current-evaluator (lambda (exp env) ((analyze exp) env)))
  (set! current-prompt "AEVAL=> ")
  (set! current-value-label ";;A-value: ")
;  (init-env)              ;Ben comments-out this line
  (eval-loop))
```

He reasons that now, if he types `(start-analyze)`, the definitions he already evaluated in the meta-circular evaluator will be available to the analyzing evaluator. Eva Lu Ator agrees that the definitions of the meta-circular evaluator will indeed be available to the analyzing evaluator, but that this will usually crash the system rather then being helpful.

Lab exercise 7A: Try Ben's suggestion and observe what Eva Lu Ator meant.

Post-Lab exercise 7B: Briefly, but clearly, explain what goes wrong with Ben's suggestion. Are there any definitions it would be safe to preserve when switching between evaluators?

Comparing evaluation speeds

This next problem asks you to demonstrate the improved efficiency of the analyzing evaluator over the meta-circular one.

To time things, you can use the procedure `show-time`, which you used in Problem Set 2. Thus, for example, you can find out how long it takes the evaluator to evaluate `(fib 10)` by quitting out of the evaluator and evaluating

```
(show-time (lambda() (current-evaluator '(fib 10) the-global-environment)))
```

in Scheme. Be careful to define the procedure you are timing, e.g., `fib`, in the evaluator, not in ordinary Scheme! Otherwise, you'll end up timing the underlying Scheme interpreter. This is because of the way we've linked the evaluator into Scheme: if you define `fib` in Scheme, and not in the evaluator's global environment, `lookup-variable-value` will find the Scheme procedure `fib` and `m-apply` will treat this as a primitive.

Lab exercise 8A: Design and carry out an experiment to compare the speeds of the `meval` and `analyze` evaluators on some procedures. Use tests that run for a reasonable amount of time (say 10 or 20 seconds). It may be helpful to use test procedures for which small changes in the input cause large changes in running time, so you can rapidly increase the time by increasing the input.

Summarize what you observe about the relative speeds of the two evaluators on your test programs.

Lab exercise 8B (Optional—Extra Credit): See if you can find an example where the interpreters run for at least five seconds and the `analyze` evaluator runs *no more* than 1.5 times the speed of `meval`.

Lab exercise 9 Repeat the process of adding `or` as a special form to the `analyze` evaluator (that is, do Exercises 2, 3 and 4 again but for the `analyze` evaluator).

Lab and Post-Lab exercise 10 (Optional): The evaluators are not quite powerful enough to evaluate themselves. That is, it doesn't quite work to evaluate all the `meval` definition files within the `MEVAL` read-eval-print loop, and then to apply `meval` as a procedure defined *within* the `MEVAL` evaluator. You may want to try this to see what happens.

Discuss what further features or definitions would be required in order to successfully run the evaluators within themselves and/or within each other.