MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 1997

Problem Set 8

Issued: Thursday, 13 November

Due: Friday, 21 November, in recitation.

Tutorial preparation for Week of 17 November:
Exercises 4.7, 4.13, 4.14, 4.27, and 4.32.

Reading: *SICP* Sections 4.1–4.1.6, 4.2, and attached code files `meval.scm`, `evdata.scm`, and `syntax.scm`.

Evaluators for Extended Scheme

You will be working with an evaluator for an extended version of Scheme as 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.

1 Using the Evaluator

The code for this Problem Set includes these files:

- `meval.scm` is the so-called "metacircular" evaluator described in Section 4.1.1 of *SICP*. It has been extended to allow declarations of lazy procedure parameters, as described in Section 4.2.2 and *SICP* 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 the evaluator procedures `meval` and `m-apply`.

- `evdata.scm` contains the procedures that define the evaluator's data structures. Note that although the definitions in this file are very similar to those Section 4.1.3, they are not exactly the same.

- `syntax.scm` contains the procedures that define the syntax of expressions. Note that although the definitions in this file are very similar to those Section 4.1.2, they are not exactly the same.

Here's how to use the extended evaluator:

- To start up, do

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

  in Edwin. This loads the files `syntax.scm`, `evdata.scm`, and `meval.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.)

  For example,

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

  shows an interaction with the `meval` evaluator.

- It will be helpful to keep separate files of procedure definitions you want to run the evaluator on. When your Edwin Scheme buffer is running the `meval` read-eval-print loop, you can then visit this file and type `M-o` to have the definitions evaluated by `meval`.

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

- The `meval` evaluator does not include much of an error system. 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)`.

- It can be instructive to trace `m-eval` and/or `m-apply` 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.

2 Lab Work

Lab exercise 1:

1A: 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.

1B: Now trace evaluation of the application of some simple procedure to some argument.

Read the definition of `or` on page 19 of *SICP*. Note that the `or` special form is not implemented in `meval`.

Lab Exercise 2: Demonstrate how, using `lazy` parameters, you can define *three*-argument `or` as a *procedure*. Turn in the definition of your `or` procedure together with some test examples to show that it works as required.

Lab Exercise 3: Define an `or->desugared` procedure which transforms an arbitrary `or` expression, which may have a varying number of arguments as in real Scheme, into an equivalent expression which `meval` can handle. Using this procedure, extend `meval` so `or` is implemented as a derived expression, i.e., as "syntactic sugar."

Lab Exercise 4:

PreLab 4A: Notice that the `initial-environment` procedure in `evdata.scm` binds the evaluator's variable `pair?` to a special `mpair?` procedure. Give an example of what would go wrong if the evaluator's variable `pair?` was bound to Scheme's built-in `pair?` procedure.

4B: Temporarily modify initial-environment so the evaluator's variable pair? is bound to to Scheme's built-in pair? procedure, and run the evaluator on your example. Now restore the original definition of initial-environment and demonstrate that your example is handled correctly.

4C: Define a predicate mproc? which applies to any Scheme value, v, and returns #f iff v is not the meval evaluator's representation of a procedure.

4D: Add a procedure? predicate to the meval evaluator. A simple way to do this is to extend the evaluator's initial environment so that the variable procedure? is bound to the mproc? procedure of 4C, just as the evaluator's variable pair? is bound to the mpair? procedure.

Section 4.2.3 develops the idea of delaying evaluation of all list elements, so all lists wind up as "lazy" lists with stream-like behavior. Second 4.2.3 implements lazy lists as lazy procedures, which works, but has some undesirable properties (cf. Exercise 4.33). In the following exercise we develop an improved implementation.

Exercise 5:

Pre-Lab 5A: One proposal for implementing lazy lists uses a procedure lazy-cons rather than cons, where lazy-cons is defined as:

```
(define (lazy-cons (a lazy) (b lazy))
  (cons a b))
```

Explain why this does not work in the current evaluator.

5B: Modify the evaluator so that

```
(cons <expr1> <expr2>)
```

returns a cons-cell whose car and cdr are the *unforced* values of <expr1> and <expr2>. There are several ways to do this, for example, by modifying the meval procedure to handle cons as a new special form, or by modifying the way m-apply applies primitive procedures. Try to make the fewest and simplest modifications to accomplish the task. Try to have your modified evaluator continue treating car and cdr as the primitives built into underlying Scheme, thereby avoiding the problems noted in *SICP*, Exercise 4.33.

5C: Check that your lazy lists in meval work properly by trying out the definition of the prime stream defined on page 330 of *SICP*, using the list operations cons, car, cdr (which you made behave lazily in 5B) instead of stream operations.

Lab exercise 6: In Problem Set 7, `define-class` was a procedure which took a quoted class definition as an argument. Extend `meval` so it properly supports a `define-class` special form (whose body does not need to be quoted). Most of the necessary code can be yanked verbatim from the Problem Set 7 file `object.scm`. (Do `M-x load-problem-set: 7` to get this file in your Edwin buffer.)

Even with lazy parameters, not all language facilities can be defined as Scheme procedures. An example is the special form `(for-list <var> <l> <body>)` where `<var>` is a variable and the value of `<l>` is a list. Evaluation of this expression causes `<body>` to be repeatedly evaluated as `<var>` is successively assigned to be the elements of the value of `<l>`. More precisely,

```
(for-list <var> <l> <body>)
```

would be syntactic sugar for

```
(let (rest-l <l>))
  (define (iter)
    (if (null? rest-l)
        the-unspecified-value
        (begin
          (set! <var> (car rest-l))
          <body>
          (set! rest-l (cdr rest-l))
          (iter))))
  (iter))
```

assuming that the local variables `iter` and `rest-l` are "fresh," in particular, that neither is the same as `<var>` or equal to any variable free in `<body>`.

Lab exercise 7: Add the `for-list` construct as a special form. Try to do better than simply treating it as syntactic sugar as described above. If you do this properly, the special form will evaluate a good deal more efficiently than the desugared version, and no assumptions about "fresh" variables will be necessary. Include some tests showing that your implementation works properly.

Turn in this sheet along with your answers to the questions in the problem set:

How much time did you spend on this homework assignment?

Indicate all of the following facilities you in doing this problem set:

- The 6.001 lab in building 36.
- MIT Scheme on Project Athena.
- MIT Scheme on a Windows PC.
- EdScheme on a PC.
- Some other Scheme system (which one?).

We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).

If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the course text (such as some of the course archives), please indicate your consultants' names and your references in the space below. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.

Otherwise, write "I worked alone using only the course reference materials." and sign your statement.