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

Problem Set 8

Scheme Evaluators

Issued: Thursday, April 10, 1997

Due: Wednesday, April 23, 1997 for recitations held at 10 and 11; and on Friday, April 25, 1997 for recitations held at 9, 12, 1 and 2. Tutorial preparation for: Week of April 14, 1997

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

QUIZ REMINDER: Quiz 2 is on Wednesday, April 16, from 5–7 PM xor 7–9 PM. Information on the quiz was distributed in lecture on April 10.

This problem set requires only a small amount of actual programming. Yet it is more challenging than any of the previous ones this semester, and it will require more careful preparation. You will be working with the evaluator described in chapter 4 of the notes. If you don't have a good understanding of how the evaluator is structured, it is very easy to become confused between the programs that the evaluator is interpreting, and the procedures that implement the evaluator itself. For this problem set, therefore, we are asking you to do some careful tutorial preparation. Once you've done this, your work in the lab should be fairly straightforward.

Part 1: Adding new derived expressions to Scheme

Early in the semester, we introduced the idea of "syntactic sugar," that is, the notion that some of the special forms in our language can be expressed as simple syntactic transformations of other language elements. Examples are `cond`, which can be implemented in terms of `if`, and `let`, which can be implemented in terms of `lambda`. Such expressions are also call *derived expressions*. Your main job in this problem set will be to design and implement a new derived expression for Scheme.

Section 4.1.2 of the notes demonstrates how to implement `cond` as a derived expression: to evaluate a `cond` expression, we transform it to an equivalent `if` expression using the procedure `cond->if`; then we evaluate the transformed expression. `Let` can also be implemented as a derived expression, as explained in exercise 4.6. The code for this problem set already has `let` installed in just this way, using a transformer `let->combination` included in the file `syntax.scm`. You may wish to study this as an aid in implementing your own derived expression.

In doing your work, we'd like you to follow a step-by-step process. We'll illustrate this process with a new derived form called `until`.

Step 1: Give some sample uses of the new form that show how you would like it to be used. For `until`, a simple example is

```
(define (countup n)
  (let ((x 0))
    (until (> x n)
           (write-line x)
           (set! x (+ x 1)))))
```

Step 2: Give a careful description of what the form does. By examining your example in step 1 (and possibly some other examples), try to come up with a formal description of what your new construct does. For `until` the general form of the expression will be

(until *test*
exp$_1$
exp$_2$
...
exp$_n$)

The description is that `until` evaluates the *test* expression. If the result is true, the value returned by `until` is `#t`.[1] If the result is false, `until` evaluates each of the expressions `exp`$_i$ in sequence, then goes back to repeat the *test*, and keeps doing this over and over until the result of the test is true.

Step 3: Rewrite the example in step 1 in terms of more primitive Scheme expressions. For instance, in the example in step 1, the `until` subexpression, namely

```
(until (> x n)
       (write-line x)
       (set! x (+ x 1)))))
```

can be rewritten as

```
(let ()
  (define (loop)
    (if (> x n)
        #t
        (begin (write-line x)
               (set! x (+ x 1))
               (loop))))
  (loop))
```

Step 4: Generalize the process in step 3 to cover the general description of the form in step 2. For `until`, the general form

[1] What the `until` form returns is not clear from the example in step 1. This is the kind of detail we need to specify in giving a careful description of the new construct.

```
(until  test
        exp1
        exp2
        ...
        expn)
```

can be rewritten as

```
(let ()
  (define (loop)
    (if  test
        #t
        (begin  exp1
                exp2
                ...
                expn
                (loop))))
  (loop))
```

Step 5: Implement the transformation in step 4 as a procedure. The procedure takes the `until` expression and returns the transformed expression. This procedure can now be added to the syntax procedures of the evaluator, and an appropriate clause added to `eval` to handle the new type of expression. In our example, the transformation is

```
(define (until->transformed exp)
  (let ((test (cadr exp))
        (other-exps (cddr exp)))
    `(let ()
       (define (loop)
         (if ,test
             #t
             (begin ,@other-exps
                    (loop))))
       (loop))))
```

As this example illustrates, the special syntax characters backquote (`), comma, and at-sign (@) are extremely useful in writing syntactic transformations. Placing a backquote before an expression is similar to placing an ordinary quote before the expression, except that any subexpression preceded by a comma is evaluated. If a subexpression is preceded by comma at-sign, it is evaluated (and must produce a list) and the list is appended into the result being formed. For example:

```
(define x '(1 2 3))
(define y '(4 5 6))

`(a b c ,x ,@y 7 8)
;Value: (a b c (1 2 3) 4 5 6 7 8)
```

We can, of course, define `until->transformed` without using backquote:

```
(define (until->transformed exp)
  (let ((test (cadr exp))
        (other-exps (cddr exp)))
    (list
     'let
     '()
     (list 'define
           '(loop)
           (list 'if
                 test
                 #t
                 (cons 'begin
                       (append other-exps (list '(loop))))))
     '(loop))))
```

but this is much more awkward. Note that that the evaluated subexpressions of a backquote form can be actual expressions, not just symbols. Thus `until->transformed` can also be defined as

```
(define (until->transformed exp)
  `(let ()
     (define (loop)
       (if ,(cadr exp)
           #t
           (begin ,@(cddr exp)
                  (loop))))
     (loop)))
```

The `until` form is a very simple example of a derived expression that can be added to Scheme. Here are some other possibilities, with examples of how they are used. (We are purposely *not* saying precisely what these forms do, because we want you to fill in the details.)

`for`: This is a general iteration facility for one variable.

```
(define (count-to max)
  (for (i 0 (+ i 1))      ;variable
       ((> i max) "done")   ;end-test and result
       (write-line i)))   ;body of loop
```

`do`: This is like `for`, except that there may be multiple loop variables.

```
(define (factorial n)
  (do ((i n (- i 1))
       (prod 1 (* prod i)))
      ((<= i 0) prod)))        ;this loop has no body

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)
    (write-line (list sum "not done"))))

(0 "not done")
(1 "not done")
(4 "not done")
(9 "not done")
(16 "not done")
;Value: 25
```

`match`: This is similar to `cond`. However, there is an extra initial expression that is evaluated to produce a *key*, and each clause begins with a *match list* rather than a predicate. As with `cond`, `match` checks the clauses in sequence, and processes the actions of the first clause that passes a test. The test for a clause is not to check a predicate (as in `cond`) but rather to see if the key is `eq?` to an element of the clause's match list. For example:

```
(match (car '(c d))
  ((a e i o u) 'vowel)  ;note that the match list is not evaluated
  ((w y) 'semivowel)
  (else 'consonant))
```

extended `cond` syntax: This is described in exercise 4.5.

Part 2: Tutorial exercises

Tutorial exercise 1: In the expansion of `until`, why did we embed the loop in a `let`? What's wrong with expanding `until` as

```
(begin
  (define (loop)
    (if  test
        #t
        (begin  exp1
                exp2
               ...
                expn
               (loop))))
  (loop))
```

Tutorial exercise 2: Choose one of the new derived forms listed above at the end of part 1, or design a new (reasonable) form of your own, and carry out the first four steps of the implementation process described above. In tutorial, you should present a careful description of your form, one or two (reasonable) programs that use it, and you should describe the general transformation that implements your form as a derived expression. You should also start writing (but not necessarily debugging) the actual procedure that implements the transformation. If you have any questions (e.g., about backquote) this is the time to ask.

Tutorial exercise 3: Do exercises 4.12 and 4.13. Be prepared to give arguments on both sides in 4.13.

Part 3: Lab exercises

The code for problem set 8 includes these files:

1. `syntax.scm` are the procedures that define the syntax of expressions, as described in section 4.1.2
2. `evdata.scm` are the procedures that define the evaluator's data structures, as in section 4.1.3. Any variable that is not found in the metacircular evaluator's global environment will be looked up in the underlying Scheme, and any actual Scheme procedure will appear to the metacircular evaluator as a primitive procedure. You can therefore make use of any Scheme primitive (or other procedure) in running programs in the metacircular evaluator.
3. `meval.scm` is the metacircular evaluator described in section 4.1.1 of the notes. 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 `m-eval` and `m-apply`.
4. `analyze.scm` is the optimized evaluator, that separates analysis from execution, as described in section 4.1.7.
5. `lazy.scm` is the `analyze` evaluator modified for lazy evaluation, as described in section 4.2.

You'll be working with each of the three evaluators (one at a time). Be careful, because it is easy to get confused with all these different evaluators. Here are things to keep in mind:

- After loading an evaluator, you start it by typing `(init)`. In order to help you avoid confusion, we've arranged that each driver loop will print prompts on input and output to identify which evaluator you are typing at. For example,

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

  shows an interaction with the `m-eval` evaluator. To evaluate an expression, you type the expression and press `ctrl-x ctrl-e`; Don't use `M-z` to evaluate the exprssion; the presence of the prompt confuses the `M-z` mechanism.

- 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 driver-loop by running the procedure `driver-loop`. If you get hopelessly fouled up, you can run `init`, but this initializes the environment, and you will lose any definitions you have made.
- It is instructive to run the interpreter while tracing `m-eval` and/or `m-apply`. (You will also probably need to do this while debugging your code for this assignment.) Since environments are, in general, 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.[2]

Lab exercise 1: Load the files `syntax.scm`, `evdata.scm`, and `meval.scm`, and start the interpreter by running `init`. Evaluate a few simple expressions and definitions. It's a good idea to intentionally make an error and practice restarting the driver loop. Notice that we have already installed `let` in the evaluator, so you can use this in your programs. There's nothing to turn in for this exercise.

Lab exercise 2: Let's first get some experience with the evaluator by adding a new special form. To do this, do exercise 4.4 of the text. Do only one of `and` or `or`, your choice. Also, please do this by installing the form as a special form, not as a derived expression.

Lab exercise 3: Implement the new syntactic form that you prepared for tutorial. (You should probably add your transformation into the `syntax` file.) After it's implemented, write two or three (non-stupid) procedures that use your form, and demonstrate that they work. **Advice:** Debug your transformation procedure *before* installing it in the evaluator. In other words, check that your procedure takes an expression of the right form, and generates the transformed expression that you expect, before you hook this into `m-eval` and run the evaluator. Turn in your transformation procedure, a note on the modification you need to make to `m-eval` in order to install your new form, and examples of the evaluator interpreting procedures that use your form.

Speeding up the evaluator

This next problem asks you to demonstrate the gain in efficiency obtained by separating analysis from execution, as described in section 4.1.7. To install the second evaluator, load the file `analyze` and run `init`. You need not reload the `syntax` or `evdata` files. These are shared between the two evaluators. You can tell that you are typing at the second evaluator because the prompt will be `A-EVAL=>` rather than `M-EVAL=>`.

To help you time things, we've included the procedure `timed`. We've also supplied a procedure called `mini-eval`, which invokes the current embedded interpreter from Scheme. Thus, for example, you can find out how long it takes the evaluator to evaluate `(fib 7)` by quitting out of the evaluator (by typing `ctrl-c ctrl-c`) and running (in Scheme)

[2] See the descriptions of `*unparser-list-depth-limit*` and `*unparser-list-breadth-limit*` in the Scheme reference manual.

```
(timed mini-eval '(fib 7) the-global-environment)
```

Be careful to define the procedure you are timing (e.g., `fib`) in the embedded evaluator, not in ordinary Scheme!. Otherwise, you'll end up timing the underlying Scheme evaluator.[3]

You can switch back and forth between the two evaluators by alternately loading `meval` and `analyze`. You will have to rerun `init` each time, though, and redefine any procedures you are timing.

Lab exercise 4: Design and carry out an experiment to compares the speed of the `meval` and `analyze` interpreters on a simple procedure. It's best to use tests that run for a reasonable amount of time (say 10 or 20 seconds). It's also good to use something like `fib` that has an input that varies the running time in an exponential way, so you can easily increase the time by changing the input. Turn in a description of your experiment and the results and describe what you can conclude from this about the relative speeds of analysis and execution times for your test programs.[4]

Experimenting with lazy evaluation

You can start the normal-order evaluator described in section 4.2.2 by loading the file `lazy.scm` and running `init`. The prompt will now read `L-EVAL=>`.

Lab exercise 5: Write two procedures `test1` and `test2` that distinguish between ordinary evaluation and lazy evaluation, in the following way: Each procedure takes a single argument, and the procedures are *identical* except that one has its parameter declared to be lazy and the other does not. Then demonstrate a way to call the procedures that distinguishes between them. That is, give an expression *exp* such that calling `(test1` *exp*`)` always returns the same value, and calling `(test2` *exp*`)` always returns the same value, but the two values are different. (Hint: Think about side-effects.)

[3] This is because of the way we've linked the interpreter 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.

[4] Feel free to use a program that makes use of your new derived expression. If you do so, however, don't forget that you will have to install your new expression for the the `analyze` interpreter as well as the `meval` interpreter.