CS61A – Homework 6.1 **Kurt Meinz**
University of California, Berkeley Summer 2003

Topic: Metacircular evaluator

Lectures: Monday July 28, Tuesday July 29

Reading: Abelson & Sussman, Section 4.1.1–6 (Pages 359–393)

This is the first of two homeworks on the metacircular evaluator. This assignment focuses on adding simple special forms as derived expressions and modifying the behavior of existing special forms. A version of the metacircular evaluator is available in `~cs61a/lib/mceval.scm`. Please copy it to your homework directory and rename is `hw6-1.scm`. Answer all questions by adding to or modifying the code in this file. Clearly mark the parts you changed. You may include test cases in this file (just be sure to comment them out) or in a separate file called `tests`. When you are done, you will have a Scheme interpreter that supports `let`, `let*` and an extended version of `define`, as well as have a built-in `map` higher-order procedure. This assignment is due at **8 PM on Sunday, August 3**.

To keep your sanity **test any new code in isolation** before testing it through the interpreter. Get in the habit of testing incrementally: test the smallest nontrivial piece of code first, and work your way up. This way, any errors you encounter will be closer to the code that produced them.

Lastly, remember to use `mce` to start the interpreter for the first time, since `mce` initializes the global environment. When you wish to get back to the REPL and preserve the state of the environment, use `driver-loop`.

Question 1. This question concerns adding derived expressions to the metacircular evaluator.

A. Add `let` as a special form to the metacircular evaluator by implementing a syntactic translation `let->lambda` that transforms a `let` expression into the equivalent procedure call:

```
STk> (let->lambda '(let ((a 1) (b (+ 2 3))) (* a b))))
((lambda (a b) (* a b)) 1 (+ 2 3))                  ;; returns a list!
```

Remember, `let->lambda` takes a *list* that represents a `let` expression and returns another *list* that represents the equivalent procedure call. Do not be intimidated by this problem simply because it appears in the context of the MCE. This is a simple list-manipulation problem; the only thing that is new is that the list happens to look like Scheme code. Make sure your `let->lambda` function works correctly before proceeding; test it in isolation, at the STk (not MCE!) prompt. After you have written `let->lambda`, install `let` into the interpreter by adding the following clause to `mc-eval`:

```
((let? exp) (mc-eval (let->lambda exp) env))
```

Don't forget to define the predicate `let?` in the obvious way. You should now be able to use the `let` form in your metacircular interpreter, like this:

```
;;; M-Eval input:
(let ((cadr (lambda (x) (car (cdr x)))))
  (cadr '(one two three)))

;;; M-Eval value:
two
```

The question continues on the next page.

B. The `let*` special form is similar to `let` except that the bindings are preformed sequentially (from left to right), allowing you to refer to previous `let` variables in defining later ones:

```
STk> (let* ((a 10) (b (* a a)) (c (+ a b)))
       (list a b c))
(10 100 110)
```

One way to implement `let*` is by transforming it into nested `let` expressions. That is, the expression

```
(let* ((a 10) (b (* a a)) (c (+ a b)))
  (list a b c))
```

is just syntactic sugar for

```
(let ((a 10))
  (let ((b (* a a)))
    (let ((c (+ a b)))
      (list a b c))))
```

Add `let*` to the MCE by implementing this syntactic transformation. Write the function `let*->lets` which takes a *list* that looks like a `let*` expression and returns nested lets. Before going further, test your function in isolation:

```
STk> (let*->lets '(let* ((a 10) (b (* a a)) (c (+ a b)))
       (list a b c)))
(let ((a 10)) (let ((b (* a a))) (let ((c (+ a b))) (list a b c))))
```

Then do everything else necessary to allow `let*` to be used in metacircular Scheme.

Question 2. In lab (Exercise 4.4) you added `and` and `or` to the MCE. An important detail of these two special forms is that `and` returns #f or the *last* true value. For example:

```
STk> (and 1 2 3 4)
4
```

Similarly, `or` returns #f or the *first* true value:

```
STk> (or 1 2 3 4)
1
```

Here is a naïve implementation of `or` that is intended to behave as above:

```
(define (eval-or exp env)
    (if (null? exp)
        #f
        (if (true? (mc-eval (car exp) env))
            (mc-eval (car exp) env)
            (eval-or (cdr exp) env))))
```

Please define `or?` in the standard way and add the following clause to `mc-eval`:

```
((or? exp) (eval-or (cdr exp) env))    ;; cdr to strip off the "and" tag
```

Show a sample interaction with the MCE that reveals a bug in this `eval-or`. You can use STk to see what the "right answer" is for any given `or` expression. How would you fix this bug? (You don't actually need to fix it if you don't want to.)

The action continues on the next page.

Question 3. Sometimes it's convenient to initialize a whole slew of variables with a single `define`. Modify the `eval-definition` function to cope with the definition of any number of variables. For example:

```
;;; M-Eval input:
(define a (+ 2 3)
        b (* 2 5)
        c (+ a b))
;;; M-Eval value:
ok
;;; M-Eval input:
(list a b c)
;;; M-Eval value:
(5 10 15)
```

Like `let*` in the previous problem, the bindings should be performed sequentially in a left-to-right order, allowing later bindings to refer to earlier ones. Do not implement this feature as a derived expression by, say, turning

```
(define a (+ 2 3) b (* 2 5) c (+ a b))
```

into

```
(begin (define a (+ 2 3)) (define b (* 2 5)) (define c (+ a b)))
```

Change `eval-definition` instead. Remember to always test in isolation first:

```
STk> (eval-definition '(define a (+ 2 3) b (* 2 5) c (+ a b))
                      the-global-environment)
ok
STk> (lookup-variable-value 'c the-global-environment)
15
```

Hint: You may find it convenient to change the `definition?` clause in `mc-eval` to strip off the "define" tag, like this:

```
((definition? exp) (eval-definition (cdr exp) env))
```

The learning continues on the next page.

Question 4. The MCE is missing quite a few primitive procedures. Evaluate `primitive-procedures` in STk to see which ones are available. The goal of this question is to make the higher-order function `map` available on startup in the metacircular evaluator:

```
STk> (mce)   ;; initializes interpreter

;;; M-Eval input:
(map (lambda (x) (* x x)) '(1 2 3))

;;; M-Eval value:
(1 4 9)
```

Depending on how you do this, `map` may end up a primitive procedure:

```
;;; M-Eval input:
map

;;; M-Eval value:
(primitive #[closure arglist=(func lst) 9d3c10])
```

or a compound procedure that is pre-defined in the MCE:

```
;;; M-Eval input:
map

;;; M-Eval value:
(compound-procedure (func lst) (...) <procedure-env>)
```

A. Why can't we just import STk's `map` into the MCE by adding it to the list of known primitives:

```
(define primitive-procedures
   (list (list 'car car)
         (list 'cdr cdr)
         (list 'map map)     ;; new!
               ...
```

Explore what happens when you attempt to use `map` in metacircular Scheme. **Hint:** STk's `map` is designed to be used with STk procedures, which look like `#[closure arglist=(x) d3afbc]`. What do MCE procedures look like?

B. Find a way to add `map` to the MCE. You may add it as a primitive or compound procedure, **but not as a special form**. There is no reason to make `map` a special form because `map` obeys the normal rules of evaluation.

You know you've done this right when you can use `map` immediately after initializing the interpreter (as in the example above). You may modify *any* functions or definitions you need to.