CS61A – Homework 6.2 | **Kurt Meinz**
University of California, Berkeley | Summer 2005

Topic: Metacircular evaluator, Lazy evaluator

Lectures: Wednesday July 30, Thursday July 31

Reading: Abelson & Sussman, Sections 4.2.2–3 (Pages 401–411)

This assignment gives you practice making substantial modifications to the metacircular evaluator, as well as introduces you to the lazy evaluator. **It is long!** As before, make a copy of `~cs61a/lib/mceval.scm` and rename it `hw6-2.scm`. Alternatively, you may use your modified metacircular evaluator from the last homework. Include test cases in this file or a separate file called `tests`. Please do Question 4 in a file called `question4.scm`. Submit all files electronically. The homework is due at **8 PM on Sunday, July 31**.

Question 1. We can create new bindings with `define` in Scheme, but there is no way to get rid of old ones. Add the `undefine` special form to the metacircular evaluator which should remove the *most local* binding of a given symbol (the same binding that would be retrieved if the symbol was to be looked up):

```
;;; M-Eval input:
(define color 'yellow)
;;; M-Eval value:
ok
;;; M-Eval input:
((lambda (color) (undefine color) color) 'green)    ;; removes local "color"
;;; M-Eval value:
yellow
;;; M-Eval input:
(undefine color)             ;; now global "color" is gone too
;;; M-Eval value:
ok                           ;; return value up to you
;;; M-Eval input:
color
*** Error: Unbound variable color
```

This problem requires you to understand the representation of environments in the interpreter. Since environments are made of pairs (surprise, surprise) you'll need `set-car!` and `set-cdr!` to change them. Make sure to test your code outside the interpreter first. This will help you isolate bugs. Assuming the underlying Scheme procedure that implements `undefine` is called `eval-undefine`, here is how you might test it:

```
STk> (define my-environment                    ;; make simple environment
            (extend-environment                ;; with one frame that
                '(a b) '(1 2)                  ;; contains two bindings
                the-empty-environment))        ;; a=1 and b=2
STk> my-environment
(((a b) 1 2))                                  ;; peek at its representation as a list
STk> (eval-undefine 'b my-environment)
ok
STk> my-environment
(((a) 1))                                      ;; no more b
```

Lastly, briefly explain why `undefine` *must* be a special form.

The homework continues on the next page.

Question 2. We're going to borrow a neat looping construct from Emacs Lisp called `do-list`. It has this syntax:

```
(do-list (<variable> <list> <return value>)
    <body>)
```

The evaluation rules are: for every element of <*list*>, bind <*variable*> to the element and evaluate <*body*>. It's important that <*variable*> be made local to the evaluation of <*body*>. It should not exist after `do-list` is done. When the list is empty, evaluate and return <*return value*>. Below are some examples (you will need to add `display` and `newline` to the list of primitives):

```
;;; M-Eval input:
(do-list (num (list 1 2 3) 'foo)
   (display num)
   (newline))
1
2
3
;;; M-Eval value:
foo
;;; M-Eval input:
(define (reverse seq)
  (let ((result '()))
    (do-list (e seq result)
       (set! result (cons e result)))))
;;; M-Eval input:
(reverse (list 'a 'b 'c))
;;; M-Eval value:
(c b a)
```

Add `do-list` to the MCE, but **not as a derived expression!** That's less interesting. Write an evaluation function for it instead.

As you work on this problem, you may notice a slight ambiguity in the specs. Should <*return value*> be evaluated in the original environment, or in the environment where <*variable*> is bound? It's up to you.

The fun continues on the next page.

Question 3. Add `trace` and `untrace` to the metacircular evaluator. Both primitive and compound procedures should be traceable, just like in STk. Don't worry about proper indentation of trace output; we just want the basic printing of arguments and return value of a traced procedure, like this:

```
;;; M-Eval input:
(trace *)                              ;; return value omitted

;;; M-Eval input:
(* (* 2 3) (+ 4 1))

* with args (2 3)
returns 6
* with args (6 5)
returns 30

;;; M-Eval value:
30

;;; M-Eval input:
(untrace *)                             ;; * is no longer traced

;;; M-Eval input:
(define (factorial n)
   (if (= n 0)
       1
       (* n (factorial (- n 1)))))

;;; M-Eval input:
(trace factorial)

;;; M-Eval input:
(factorial 5)
factorial with args (5)
factorial with args (4)
factorial with args (3)
factorial with args (2)
factorial with args (1)
factorial with args (0)
returns 1
returns 1
returns 2
returns 6
returns 24
returns 120

;;; M-Eval value:
120
```

The return values of `trace` and `untrace` are up to you. There are several ways to do this problem, but the easiest is to stick a boolean inside the list that represents a procedure that will be true if the procedure is being traced and false otherwise. All that's left then is to figure out where procedures are called and do some printing. Depending on your implementation, `trace` and `untrace` may or may not need to be special forms. Did you make them special forms? Briefly explain.

The excitement continues on the next page.

Question 4. This question explores the difference between normal-order evaluation (as implemented by the lazy interpreter) and applicative-order evaluation (as done by STk or our own metacircular evaluator). Please put your answers to this question into a file `question4.scm`.

A. The "Hanoi stream" is an infinite stream of the form:

`1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 6 1 2 1 ...`

As you can see, every other element in the stream is a one:

$\underline{1}$ 2 $\underline{1}$ 3 $\underline{1}$ 2 $\underline{1}$ 4 $\underline{1}$ 2 $\underline{1}$ 3 $\underline{1}$ 2 $\underline{1}$ 5 $\underline{1}$ 2 $\underline{1}$ 3 $\underline{1}$ 2 $\underline{1}$ 4 $\underline{1}$ 2 $\underline{1}$ 3 $\underline{1}$ 2 $\underline{1}$ 6 $\underline{1}$ 2 $\underline{1}$ ...

If we take out all the ones, we get a stream where every other element is a two:

$\underline{2}$ 3 $\underline{2}$ 4 $\underline{2}$ 3 $\underline{2}$ 5 $\underline{2}$ 3 $\underline{2}$ 4 $\underline{2}$ 3 $\underline{2}$ 6 $\underline{2}$ ...

And so on. This leads to the following rather intuitive generator procedure:

```
(define (make-hanoi-stream n)
   (interleave (stream-of n)
               (make-hanoi-stream (+ n 1))))

(define (stream-of x) (cons-stream x (stream-of x)))
```

The Hanoi stream can then be made by evaluating:

```
(define hanoi (make-hanoi-stream 1))
```

Try this in STk and explain the results. **Hint:** You have seen this question before.

B. Now let's try a similar approach in the lazy evaluator. Although the lazy evaluator does not have streams, it has something better: non-strict compound procedures, which allow us to implement *lazy lists.* To quote SICP (Page 409), "With lazy evaluation, streams and lists can be identical, so there is no need for special forms or for separate list and stream operations." As shown on Page 409, implement pairs as procedures in the lazy evaluator, thereby eliminating the need to have `cons`, `car` and `cdr` as primitives. Adapt `make-hanoi-stream` and `stream-of` to create lazy lists instead of streams. You may use this definition of `interleave`:

```
(define (interleave list1 list2)
   (cons (car list1) (interleave list2 (cdr list1))))
```

Use the following procedure to print the first n elements of a lazy list (you will need to add `display` and `newline` as primitives):

```
(define (show-lazy lst n)
   (if (= n 0)
       (begin (display "...") (newline))
       (begin (display (car lst))
              (display " ")
              (show-lazy (cdr lst) (- n 1))))
   'ok)
```

Include a short session with the lazy evaluator demonstrating that the generator function works.