Brandeis University

Computer Science (1)21b (Spring Term, 2018)

Structure and Interpretation of Computer Programs

Problem Set 5: Metacircular Evaluator

Due Wednesday, April 11

Reading Assignment: Chapter 4, Sections 4.1–4.2.

1 Homework Exercises

This problem set asks you to modify the Scheme evaluator to allow for procedures that pass parameters in a lazy "call-by-name" style. The necessary modification is outlined in Section 4.2.2 of the text. Although this modification does not require writing a great deal of code, you will need to have a good understanding of the evaluator in order to do this problem set.

- Exercise 4.6: adding `let`.
- Exercise 4.7: adding `let*`.
- Exercise 4.13: binding and unbinding variables.
- Exercise 4.15: detecting halting procedures.
- Consider a metacircular evaluator which does not implement the `if` construct, but only a `cond`. Louis Reasoner claims that `if` is unnecessary, since we could evaluate the following procedure using the metacircular evaluator:

  ```
  (define (if predicate action alternative)
    (cond (predicate action)
          (else alternative)))
  ```

  Explain why this does not work. In particular, suppose you use this `if` procedure to define `factorial`:

  ```
  (define (factorial n)
    (if (= n 1)
        1
        (* n (factorial (- n 1)))))
  ```

  What happens when you attempt to evaluate `(factorial 3)`? Why?

  We realize that you have seen this problem before, but we want you to be thinking about this issue before going on to the rest of the assignment. Note also that you cannot try out the above definitions in an ordinary Scheme, since Scheme's own definition of `if` will interfere with the one you attempt to define. (Ordinarily, there is nothing to stop you from re-defining Scheme primitives, but `if` is implemented in Scheme as a special form.)

2 Building a *hybrid* evaluator: synthesizing applicative-order and normal-order evaluation

The file `mceval.scm`, accessible from the home page for the course, contains a copy of a simple Scheme interpreter written in Scheme, identical to that in Section 4.1, with the following very minor difference: the procedures `eval` and `apply` have been renamed `mc-eval` and `mc-apply`, so as not to interfere with Scheme's own `eval` and `apply` operators. *Because the code you need to modify is virtually identical to the metacircular evaluator explained in the textbook, it is not included here.*

If you load this file into Scheme and type `(driver-loop)`, you will find yourself typing at the driver loop. Please note that this Scheme running in Scheme contains no error system of its own. If you hit control keys to stop evaluation, you you will bounce back into "normal" Scheme. Also, you must type in procedure definitions directly to the driver loop, since in the metacircular evaluator there is no interface to the file system via `load`.[1] In order to help you keep from becoming confused, the driver loop uses a "prompt" for input that is different than the ordinary Scheme prompt.

Start up the interpreter and try a few simple expressions. If you bounce out into Scheme, you can re-enter the interpreter by typing `(driver-loop)`. If you get hopelessly fouled up, you can run `setup-environment`, but this initializes the global environment, and you will lose any definitions you have made.

Also, it is instructive to run the interpreter while tracing `mc-eval` and/or `mc-apply`, to see just how the evaluator works. (You will also probably need to do this while debugging code for this assignment.)

Problem 1. The basic metacircular evaluator and its *lazy* (normal-order) evaluation variation use two different procedure calling protocols. These protocols differ in when they evaluate arguments. In this exercise, you are to create a *hybrid* procedure-calling protocol, where the *definition* of each procedure specifies which arguments are evaluated immediately, and which are delayed. To understand the point of this facility, consider the Scheme code

```
(define (try a b)
  (cond ((= a 0) 1)
        (else b)))
```

Evaluating `(try 0 (/ 1 0))` will produce an error in Scheme (try it!), even though the second argument isn't needed to produce output—the procedure only checks if `a` is zero. Clearly the problem is the procedure-calling protocol in Scheme: in evaluating a combination, we must evaluate *all* the subexpressions. Instead, imagine we could use (this is *not* MIT Scheme!)

```
(define (try a (delayed b))
  (cond ((= a 0) 1)
        (else b)))
```

where `(delayed b)` is an annotation to *change* the procedure-calling protocol as follows: when evaluating a combination, first evaluate the subexpression in the "function" position, and if it is this procedure, evaluate the first argument (`a`), but *delay* the evaluation of the second argument (`b`).

The convention is therefore: (1) if just a variable name is given, the operand should be passed using applicative-order (call-by-value); (2) if the variable name is *tagged* with the *keyword* `delayed` (which is *not* a procedure!)—that is, the parameter `x` is specified as `(delayed x)`—the operand should be passed using normal-order (call-by-name).

For example, consider the procedure call `(foo 2 (/ 1 0))` with the following two different definitions of `foo`: either (1) `(define (foo (delayed a) b) a)`, or (2) `(define (foo a (delayed b)) a)`. In (1), the call results in a run-time error because, the second argument to foo (that is, `b`) is evaluated call-by-value, causing a division by zero. But in (2), the call succeeds returning the value 2 passed in for `a`—and the delayed second argument is not evaluated.

[1] However, you can use the Emacs editor to incrementally load code.

This procedure-calling protocol is actually harder to *describe* than to *program*—it does not take many lines of code. You should be able derive such a hybrid evaluator by modifying about a dozen lines of code, starting from the normal-order evaluator. The difficulty in constructing the hybrid evaluator is not so much *writing* the code, but rather the *understanding* of the details of the metacircular evaluator.

Section 4.2.2 of the text sketches the modification you need to change the evaluator from one that does *eager evaluation* (arguments are evaluated before procedure is called) to one that does *lazy (normal order) evaluation* (we procrastinate such evaluation as much as possible). Your implementation may either "scale up" from the eager evaluator, or "scale down" from the lazy one—the web page contains pointers to both.

Filling in the details will require modifying a few of the procedures already given, and also writing new procedures (for example, to handle *thunks*—look it up!). Be sure to test your implementation. For all of these lab problems, turn in listings of the procedures that you write and modify.

As part of your testing process, try the following example—notice when `try` is called, it evaluates its first argument, and delays evaluation of the second:

```
;;; M-Eval input:
(define (try a (delayed b))
  (cond ((= a 0) 1)
        (else b)))
;;; M-Eval value:
ok
;;; M-Eval input:
(try (- (* 10 10 10) 1000) (/ 1 0))
;;; M-Eval value:
1
```

Some hints in designing the implementation:

- Thunks are our implementation of laziness—but when is it that thunks need to get evaluated? (Any form of laziness, in real or computational life, needs to be complemented by an innate sense of when stuff has to *really* get done.)

  When a thunk is evaluated, it's because "someone" really needs the answer. This situation occurs in the read-eval-print loop (you don't want to return a thunk to the user as the result of an evaluation), when a primitive procedure is applied (how can you multiply two thunks?), when a conditional expression is evaluated (you need to evaluate the predicate), and when a procedure has an undelayed argument (you've been *told* to evaluate the argument.)

- The expressions like `(delayed a)` only occur in definition of a procedure. Thus instead of `(lambda (x y z) ...)` you might have `(lambda (x (delayed y) z) ...)`. But recall that `delayed` is *not* a procedure, or even a special form—it is a *tag* in a list of formal parameters in a procedure definition. The tag says that when the procedure is applied, the parameter is not to be evaluated, but "thunked." *When* are you going to need to read this tag?

- Consider the following:

  ```
  (define (foo (delayed x))
    (cond (x 0)
          (else 1)))
  (foo #f)
  ```

  If the interpreter you implemented responds with a 0, you probably forgot a place where it is necessary to force thunks. Find it.

Problem 2. When you think your implementation is working, try defining `if` as a procedure. Use `if` in defining `factorial`, as in the pre-lab exercise. Make sure they both work.

Problem 3. In our discussion of streams, we said they are like lists, except that the second argument to `cons-stream` is delayed. However, with automatically delayed arguments, streams can be identical to lists. To make lists behave like streams, define `cons` as a (nonprimitive) procedure with delayed arguments. `Cons` will use a new primitive `internal-cons` that does not force its arguments, as do the other primitive procedures. Modify the evaluator to handle `internal-cons`. (If you prefer, you can do what Scheme `cons-stream` does and delay only the second argument to `cons`.)

Test your implementation by trying the following examples:

```
;;; M-Eval input:
(define (integers-from n)
  (cons n (integers-from (+ n 1)))
;;; M-Eval value:
ok
;;; M-Eval input:
(define integers (integers-from 1))
;;; M-Eval value:
ok
;;; M-Eval input:
(car integers)
;;; M-Eval value:
1
;;; M-Eval input:
(car (cdr integers))
;;; M-Eval value:
2
```