MC28 Lab 6
Making the Meta-Circular Evaluator Talkative

Max Hailperin

Due November 22, 1995

You may do this lab in pairs. You may find your own partner, or you may ask for help finding one.

In this lab, your goal is to make the meta-circular evaluator explain what it is doing all the while it is doing it. This could turn it into a useful pedagogical tool, in that it would make the evaluation process explicit.

This lab should be done on the Nexts. You'll start with a slightly modified version of the text's meta-circular evaluator. This version is in `~max/MC28/mceval.scm`; a printout of it is also attached to this lab handout. Note that we have added some tracing utilities which allow for indentation according to level, described below. Make sure you understand the evaluator and have experimented with using it before you begin the lab proper. When you are experimenting with it, make sure you keep straight when you are evaluating things using the normal Scheme and when you are using the meta-circular "mini-scheme." The level indicator at the top of the Schematik Interaction Window can help with this.

Now you are ready to make the changes so that the meta-circular evaluator describes what it is doing. There is a sample interaction attached to this lab handout, illustrating the sort of output you should strive for. Read through it carefully enough to understand what it all means.

Notice that the indentation structure of the descriptive output reflects sub-problem levels. This can be achieved by using a global state variable called `level` is increased when a subproblem is encountered, and is set back when the subproblem evaluation is completed. This is set up as follows:

```
(define level 0)

(define (indent-line)
  (newline)
  (display-times "  " level))

(define (display-times value repetitions)
  (if (= repetitions 0)
      'done
      (begin (display value)
             (display-times value (- repetitions 1)))))
```

`Indent-line` goes to the next line and indents 2·`level` spaces.

Using `indent-line`, you can then indent according to level by using two procedures which couch a call to `mini-eval` inside appropriate modifications of the global state variable `level`. The first of these, `main-eval`, is called for the top-level problem:

```
(define (main-eval exp env) ; like mini-eval but for main (top level) problems
  (let ((old-level level))
    (set! level 0)
    (let ((value (mini-eval exp env)))
      (indent-line)
      (display "Result is ")
      (write (ready-value-for-printing value))
      (set! level old-level)
      value)))
```

The second procedure, `subp-eval` is called when a sub-problem evaluation is encountered.

```
(define (subp-eval exp env) ; like mini-eval but for sub-problems
  (let ((old-level level))
    (set! level (+ level 1))
    (let ((value (mini-eval exp env)))
      (indent-line)
      (display "Result is ")
      (write (ready-value-for-printing value))
      (set! level old-level)
      value)))
```

For example, each operand evaluation in an application is a sub-problem. Evaluating the predicate expression at the beginning of a `cond` clause is similarly a sub-problem of evaluating the overall `cond` expression. On the other hand, when an evaluation just represents a later stage in solving the original problem, you should just call `mini-eval` (which must be modified to be talkative).

Using these two procedures, as well as the modified version of `mini-eval`, you should then have the meta-circular evaluator print out the appropriate messages, as illustrated in the sample interaction included with this lab. The following list enumerates the types of messages, and you can use it as a checklist for adding the various messages to the metacircular evaluator. The order you add them isn't critical, however.

- Evaluating *expression* in environment *environment-id-number*.
- Applying *procedure* to *list-of-argument-values*.
- Creating new frame *frame-id-number* as global environment binding *list-of-names* to *list-of-values*.
- Creating new frame *frame-id-number* with parent *environment-id-number* binding *list-of-names* to *list-of-values*.

- Found binding for *variable-name* in frame *frame-id-number*.
- No binding found for *variable-name* in frame *frame-id-number*. [One of these messages is produced for every unsuccessfully searched frame, even if a binding is eventually located in an ancestor frame. That way the full search process is made visible.]
- Changing binding to *new-value*. [This is produced both when a `set!` is evaluated and also when a `define` happens to change an existing binding rather than adding a new one. In either case, the binding being changed is found first.]
- Adding binding to *new-value*. [This is produced when evaluating a `define` in the case that an existing local binding wasn't found.]
- Result is *value*. [This message is produced by the `main-eval` and `subp-eval` procedures above.]