Structure and Interpretation of Computer Programs
October 2000

Problem Set 9 **Metacircular Evaluator**

Issued: Wednesday, 25 October 2000
Due: Friday, 27 October 2000
Reading: Text (SICP 2nd Edition by Abelson & Sussman): Section 4.1, through 4.1.6

You will be working with the metacircular evaluator described in Section 4.1 of the textbook. 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, the procedures that implement the evaluator itself, and Scheme procedures called by the evaluator. You will need to have carefully studied Chapter 4 through subsection 4.1.6 in order to do this assignment.[1]

Using the Metacircular Evaluator

The code for this problem set is in the file `mc-eval.scm`. This is the file that was distributed in class on Wednesday, 25 October 2000. To start up, `M-x load-problem-set 9`. Then evaluate `(mc-eval-loop)` in Scheme; this starts the read-eval-print loop for the metacircular evaluator with a freshly initialized global environment.

- To evaluate an expression, type the expression into the `*scheme*` buffer followed by `C-x C-e`.
- Use a separate file to keep any procedure definitions you want to install in the evaluator.
- The evaluator can be interrupted by typing `C-c C-c`. To restart it with the recent definitions still intact, evaluate `(driver-loop)`.
- To start a read-eval-print loop for the evaluator in a fresh global environment, return to Scheme and evaluate `(mc-eval-loop)`.
- The metacircular evaluator you are working with does not include error systems. If you hit an error you will bounce back into ordinary Scheme. You can restart the current evaluator, with its global environment still intact, by evaluating `(driver-loop)`.
- It can be instructive to trace `mc-apply` and/or `mc-eval` during evaluator executions. (You will also probably need to do this while debugging your code for this assignment.) To trace the `mc-eval` procedure, evaluate `(trace mc-eval)` in the Scheme environment (not in the metacircular evaluator). Then when you evaluate an expression in the metacircular environment, information about the processing of the expression will be displayed. See the `info` entry on `trace` to understand the output it generates.

[1]This problem set is a substantially modified version of the Scheme Evaluators problem set on the MIT Press web site.

- Since environments are generally 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. This was done by

  ```
  (set! *unparser-list-depth-limit* 7)
  (set! *unparser-list-breadth-limit* 10)
  ```

  at the end of the file `mceval.scm`. You may want to alter the values of these limits to vary how much list structure will be printed as output.

Exercises

Exercise 1: Start the `mceval` evaluator and evaluate a few simple expressions and definitions. It's a good idea to make an intentional error and practice restarting the read-eval-print loop. There is nothing to turn in for this exercise.

Exercise 2: Trace evaluation of the application of some simple procedure to some argument such as `(+ 2 3)`. There is nothing to turn in for this exercise. Make sure, however, that you have a good understanding of the result.

Exercise 3: Add the special form `adu:or` to the metacircular evaluator. (Hint: it might be similar to adding the special form `adu:and` to the metacircular evaluator, which we did in class on Wednesday, 25 October.) Add the appropriate clause to `mc-eval`[2] to handle `adu:or` expressions. Turn in your code and sample runs. Include expressions to fully test the functionality of your implementation.

Exercise 4: We say that `let` expressions are derived expressions, because

```
(let ((var1 exp1) ... (varN expN))
   body)
```

is equivalent to

```
((lambda (var1 ... varN)
    body)
 exp1
 .
 .
 .
 expN)
```

Implement a syntactic transformation `let->combination` that reduces evaluating `adu:let` expressions to evaluating combinations of the type shown above, and add the appropriate clause to `mc-eval` to handle `adu:let` expressions. Turn in your code and sample runs.

Exercise 5: If we were to evaluate the following expressions,

```
(adu:define (adu:foo adu:cond adu:else)
   (adu:cond ((adu:= adu:cond 2) 0)
             (adu:else (adu:else adu:cond))))
(adu:define adu:cond 3)
```

[2]Make a copy of the `mc-eval` procedure in your solutions buffer and make your changes to that copy. If you need to restart Edwin, remember to evaluate your new `mc-eval` once you load the problem set code.

```
(adu:define (adu:else adu:x) (adu:* adu:x adu:x))

(adu:define (adu:square adu:x) (adu:* adu:x adu:x))
```

what value will the metacircular interpreter give for each of the following expressions?

```
(adu:foo 2 adu:square)

(adu:foo 4 adu:square)

(adu:cond ((adu:= adu:cond 2) 0)
          (adu:else (adu:else 5)))
```

Try to figure out what will be returned before typing the expressions into the metacircular evaluator.

Exercise 6: You might be disturbed by the results of the previous exercise. It might be better to make the evaluation of the above expressions signal an error rather than to let it give the results it does. You could put `adu:else` and the names of the special forms into a list of `reserved-words`. Then signal an error if a program tries to `adu:define` a name that appears in this list or if an `adu:lambda` parameter appears in this list. Modify the evaluator to implement this change. (Hint: You should not have to modify any procedures outside Section 4.1.3 of the textbook.) To test your changes, evaluate the expressions from the previous exercise in the metacircular evaluator both with and without your modifications. Turn in your code and your tests.

Exercise 7: Change something about ADU Scheme in the metacircular evaluator. This can be as simple or complex as you'd like to make it. Describe your change(s) and turn in the relevant code with tests.

As a suggestion, implement a crude debugger by writing a replacement for `error` called `mc-error` which you will call every time the metacircular evaluator encounters a problem with an ADU Scheme expression. The suggested operation of `mc-error` will be to start a recursive session of the `mc-eval` read-eval-print-loop, allowing the user to examine the state of local variables and the like. There are a number of user-interface issues that will need addressing.