CS61A Discussion Notes: Week 11: The Metacircular Evaluator

By Greg Krimer, with slight modifications by Phoebus Chen (using notes from Todd Segal)

What is the Metacircular Evaluator?

It is the best part of the best class at Cal. An evaluator—also called an *interpreter*—is a program written in some language *L1* that evaluates expressions in language *L2*. STk is an interpreter; it is written in C (*L1*) and it evaluates Scheme (*L2*). If *L1* and *L2* happen to be the same, the interpreter is *metacircular*. Hence, the metacircular evaluator is a Scheme program that evaluates Scheme expressions. Although our implementation language *L1* will stay Scheme, over the next few weeks *L2* will change. Chapter 4 of the book shows how to implement in Scheme first a Scheme interpreter, then an optimizing Scheme interpreter, a lazy Scheme interpreter, a nondeterminisitic interpreter and a query interpreter. In the fourth project, you will complete an implementation of a Logo interpreter in Scheme.

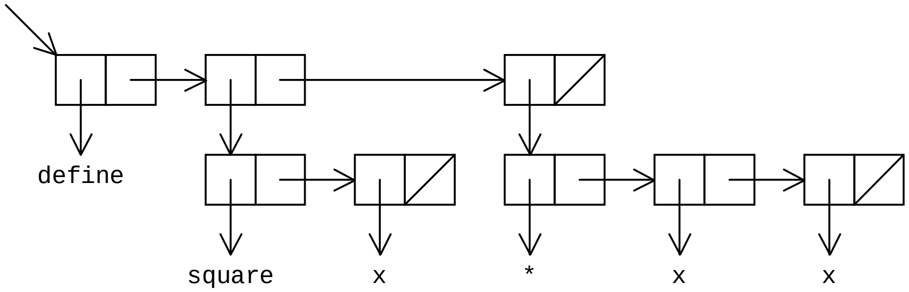
A few weeks ago, we learned that to evaluate a Scheme expression, we must draw an environment diagram. This is how STk evaluates Scheme. This is also how our evaluator will evaluate Scheme—by drawing the environment diagram in code. But first, we need to talk about the representation of a Scheme program used by the metacircular evaluator.

Programs as data

A Scheme program is a perfectly good list. Take the `square` function:

```
(define (square x) (* x x))
```

All you need to do is `quote` the definition and you've got a proper list of three elements:



The MCE uses the `read` primitive to read in Scheme expressions; `read` is the Scheme *parser*, a function that takes raw input and transforms it into a form that is easer to handle for the interpreter. If `read` sees a symbol, string or number, it just returns it. If it sees a combination—anything in parentheses—it constructs a list to represent the combination. The `read` function does not perform any evaluation[1]; it just returns a data structure—most often a list—that represents the expression.

[1] It does convert some Scheme expressions into their true form. For example, `read` is responsible for expanding any single-quotes into calls to the `quote` special form:

```
STk> (read)
'''h                          ;; user input
(quote (quote (quote h)))     ;; return value of READ
```

This means our evaluator can pick apart a Scheme program using regular list operators!

1. Draw the box and pointer diagram for the Scheme expression (do not *de-sugar* it):

```
(let ((a 3) (b 4))
  (set! a (+ a b))
  a)
```

`Mc-eval` and `Mc-Apply`: Overview

As mentioned earlier, the meta-circular evaluator merely implements the rules for drawing environment diagrams. Let's review those rules:

1. **Self-Evaluating** - Just return their value
2. **Symbol** - Return closest binding, if none error.
3. **Special forms:**
   - o *Define - bind var name to evaluation of rest in current frame*
   - o *Lambda - Make a procedure, write down params, and body - Do not evaluate*
   - o *Begin - Evaluate each expression, return value of last one*
   - o *set! - find var name, eval expression and set var to the return value*
   - o *if - eval predicate and then either the true-part or false-part.*
4. **Procedures**
   - o *Primitive's - Apply by magic...*
   - o *User-defined - Make a new frame, extend to proc's frame, bind arguments to formal parameters, evaluate the body of the procedure in the new frame.*
5. **Syntactic Sugar** - Get rid of it (untranslate)!

Now all we have to do is translate this into a working Scheme evaluator. Notice how only two things ever happen: we evaluate something, or we apply a procedure to arguments. Because of this we only really need two procedures. The procedure that evaluates expressions (`eval`), and a procedure to apply an operator to arguments in a new environment (`apply`). THATS IT! Here's a simplified/easier-to-read version of the code in the book:

```
(define (scheme)
  (print '|> |)
  (print (eval (read) the-global-environment))
  (scheme) )

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)                        ;;Rule 1
        ((symbol? exp) (lookup exp env))                    ;;Rule 2
      ((special-form? exp) (do-something-special exp env))  ;;Rule 3
      (else (apply (eval (car exp) env)                     ;;Rule 4
                   (map (lambda (e) (eval e env)) (cdr exp))) ) ) )

(define (apply op args) ;;Rule 4... verbatim
  (if (primitive? op)
      (do-magic op args)
      (eval (body op)
            (extend-environment (formals op)
                                args
                          (op-env op) ))))
```

The book expands out the `special-form?` clause in `eval` to handle each special form. It also uses more data abstraction for the application of procedures (for example, using `list-of-values` instead of `map` in the `eval` clause handling procedure application). But the code above represents the fundamental components of the meta-circular evaluator. Let's now get into the details…

`mc-eval`: dispatching on the expression type

The heart of the MCE lies in the mutually recursive relationship between the `mc-eval` and `mc-apply` functions. The book defines these as `eval` and `apply`, but we renamed them so as not to overwrite the STk primitives with the same names. The `mc-eval` function takes a Scheme expression (a list!) and an environment and evaluates the expression in the environment.

In order to evaluate a Scheme expression, `mc-eval` has to first determine its type and then "do the right thing" for that type. Since Scheme uses prefix notation and a Scheme program is a list, the `car` of that list will contain the operator of the expression. For instance:

```
STk> (car '(+ 1 2 3))
+                                           ;; the operator is +
STk> (car '(define (square x) (* x x)))
define                                      ;; operator is define
```

The metacircular evaluator refers to the operator of a Scheme expression as a *tag* and defines a function `tagged-list?` that checks if a Scheme program begins with a certain operator (i.e. starts with a given tag):

```
STk> (tagged-list? '(define (square x) (* x x)) 'define)
#t
STk> (tagged-list? '(define (square x) (* x x)) 'lambda)
#f
STk> (tagged-list? '(lambda (x) x) 'lambda)
#t
```

Using `tagged-list?`, `mc-eval` is able to determine what kind of expression it must evaluate. I am being purposely vague about the meaning of "type of expression" since the following exercise will have you discover it.

2. Here is the `mc-eval` code. You'll notice that each of the predicate functions that are in **boldface** are defined in terms of `tagged-list?`. For each of these predicates, determine the corresponding *tag* and write it on the side.

```
(define (mc-eval exp env)
  (cond
    ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp env))
    ((quoted? exp) (text-of-quotation exp))          ;; tag is
    ((assignment? exp) (eval-assignment exp env))    ;; tag is
    ((definition? exp) (eval-definition exp env))    ;; tag is
    ((if? exp) (eval-if exp env))                    ;; tag is
    ((lambda? exp)                                   ;; tag is
     (make-procedure (lambda-parameters exp)
                     (lambda-body exp) env))
    ((begin? exp)                                   ;; tag is
      (eval-sequence (begin-actions exp) env))
    ((cond? exp) (mc-eval (cond->if exp) env))      ;; tag is
    ((application? exp)
      (mc-apply (mc-eval (operator exp) env)
              (list-of-values (operands exp) env)))
    (else
      (error "Unknown expression type -- EVAL" exp))))
```

What do all the tags have in common? What is s*pecial* about them?

The definition of the `application?` predicate does not use `tagged-list?`:

```
(define (application? exp) (pair? exp))
```

Why is the definition so simple?

Big idea! Since no one did the lab this week, we will revisit one of the lab questions: Louis Reasoner (oh oh!) realizes that most of the time evaluation of Scheme expressions will land in the `application?` clause of the `cond`. He proposes to move `application?` to the top of the `cond`. This way `mc-eval` will not need to search the entire `cond` to evaluate a procedure application. What is wrong with his idea?

3. With the metacircular interpreter, you have the power to change many, many aspects of the Scheme language. (Homework Exercise 4.10 asks you to make some changes to the syntax of Scheme. You can use your answers to this exercise as answers to 4.10.) Implement a new syntax for procedure calls (leave special forms alone). To invoke a procedure, you must first say `stanford-sucks`, then the procedure, then arguments:

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

```
(stanford-sucks + 1 2 3)

;;; M-Eval value:
6
```

Here is the definition of `factorial` using this new syntax:

```
(define (factorial n)
    (if (standord-sucks = n 0)
         1
         (stanford-sucks factorial (stanford-sucks – n 1))))
```

As we saw, it is imperative that `mc-eval` check for all special forms before the catchall `application?` clause because special forms have their own rules of evaluation. These rules of evaluation are captured in functions like `eval-definition`, `eval-if` or `eval-assignment`. **Big idea: to change the behavior of a special form, you need only change the procedure responsible for evaluating it.**

4. In most programming languages, the assignment operator(s) returns the new value of the variable. Currently in the metacircular evaluator, `define` and `set!` both return "ok". Make the necessary changes to the functions responsible for evaluating these special forms so that both `define` and `set!` return the new value of the variable. Examples:

```
;;; M-Eval input:
(define a (+ 1 (define b (+ 1 (define c 0)))))    ;; c = 0
                                                  ;; b = 1
;;; M-Eval output:                                ;; a = 2
2                               ;; returns new value of a
```

Midterm 1 revisited. What would this return in the MCE after your changes are made?

```
((define (cube x) (* x x x)) 3)
```

5. In STk, passing `if` more than three arguments causes an error. What about in the MCE? That is, what would the following return:

```
;;; M-Eval input:
(if (= 2 3) ‘yes ‘no ‘maybe ‘so)
```

What about:

```
;;; M-Eval input:
(if (= 2 3) ‘yes ‘no ‘maybe ‘so (/ 2 0))
```

Adding special forms

You can add your own special forms to the MCE. All you have to do is provide a way for `mc-eval` to recognize them (insert a predicate that checks for the tag of your special form) and create the special evaluation rules for your special form. When defining special forms, you have *complete* control over how you want the evaluation to proceed.

You also have two ways to implement the special evaluation rules you desire:

1. **As an evaluation function.** Write a function that knows how to evaluate your special form. That is, define `eval-`*foo*, where *foo* is the name of the special form you're adding; this evaluation function will call `mc-eval`, `mc-apply` and any of the other functions that make the MCE work to evaluate your special form from start to finish. For example, the `eval-if` function implements the special rules for evaluating `if` statements: evaluate the *predicate*, and depending on its value evaluate the *consequent* or *alternative*.

   The corresponding clause in `mc-eval` is:

   ```
    ((if? exp) (eval-if exp env))  ;; EVAL-IF calls MC-EVAL as needed
   ```

2. **As a derived expression.** With this approach, you're writing the function *foo*`->`*bar*, where *foo* is the special form you're adding and *bar* is a Scheme expression that the MCE already knows how to handle. Instead of writing an evaluation function, you're transforming the special form into an expression that the interpreter already knows how to handle. For example, the `cond->if` function takes a `cond` expression and returns the equivalent `if` expression:

   ```
   STk> (cond->if '(cond ((= a b) 'yes)
                         ((= c d) 'no)
                         (else 'maybe)))
   (if (= a b) (quote yes) (if (= c d) (quote no) (quote maybe)))
   ```

   The corresponding clause in `mc-eval` is:

   ```
   ((cond? exp) (mc-eval (cond->if exp) env))  ;; COND->IF does not
                                               ;; call MC-EVAL
   ```

   A crucial point to keep in mind whenever you are reading code for the meta-circular evaluator is which language you are working in: 1.The Language You Are Implementing (in this class Scheme or Logo), or the 2. Language Used by the Implementation (in this class Scheme).

1. Language You Are Implementing (Source Language)

---

2. Languaged Used by the Implementation (Implementation Language)

- The metacircular evaluator generally translates 1. → 2.
- However, in the case of derived expressions, we translate 1. → 1.

- Basically, we are removing “syntactic sugar” when we translate derived expressions.

6. Add the `and` special form to the interpreter as a derived expression. That is, define the `and?` function to spot `and` expressions. Then define the function `and->if` to perform the syntactic transformation of an `and` expression to an `if` expression. To simplify things, `and->if` will be called with the initial `and` tag stripped off. Here is what we'll put into the `cond` in `mc-eval` to make this work:

```
((and? exp) (mc-eval (and->if (cdr expr) env)))
```

Also, unlike STk's `and`, ours should return `#t` if all the arguments are true; it need not return the value of the last true expression. Here is how `and->if` should work:

```
STk> (and->if ’(cs61a is cool))
(if cs61a (if is (if cool #t #f) #f) #f)
```

7. Now add `or` to the interpreter, but do not add this special form as a derived expression. Instead, write a function to evaluate `or` expressions, `eval-or`.

Two important procedures: `eval-sequence` and `list-of-values`

Open your textbooks and find out what each does and when they're called:

- **`eval-sequence`**

- **`list-of-values`**

8. Add the `while` expression to the evaluator. The `while` is a “looping construct” found in most programming languages (including Scheme—`while` is an STk primitive; try it!). It is an alternative to recursion. The while expression looks like this:

```
(while <predicate>
     <body>)
```

If *predicate* is true, *body* (which is a bunch of Scheme expressions) is evaluated. The *predicate* is then checked again, and if it’s true, we do *body* again, and check *predicate*, etc. So *body* is evaluated over and over until *predicate* is no longer true. Here is an example:

```
(define x 2)

(while (not (equal? x 10))       ;; need to import not
    (set! x (+ x 1))
    (display x)                  ;; will need to import display
    (newline))                   ;; also need to import newline
```

The evaluation of this `while` expression causes the numbers 3, 4 … 10 to be displayed on the screen. At the conclusion of the evaluation, `x` is 10.

First of all, ask yourself why does `while` need to be a special form?

Next, write the selectors `while-test` and `while-body` that extract the corresponding parts of a `while` expression. Then, do whatever is necessary to add `while` as a special form. Make sure to use the abstract selectors you’ve created. The `while` expression *can* be added as a derived expression, though I think it is easier to write an `eval-while` procedure. Try to do it both ways. The return value of `while` is up to you.

Once you’ve added `while`, try to define the `factorial` procedure using `while`. (Hints: you will need a local variable; `while` goes well with `set!`.)

`mc-apply`: applying primitive and user-defined procedures

Just like `mc-eval`, `mc-apply` is a cond statement. `mc-apply` decides whether a procedure is 1) primitive or 2) user-defined, and dispatches them differently.

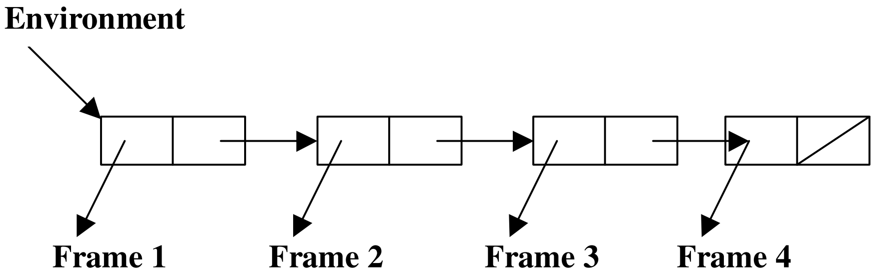
```
(define (mc-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

- Note that all of scheme's primitive procedures do not need environments to execute, but user-defined procedures need environments.
- Hey, I thought `mc-apply` also calls `mc-eval`? Where are the calls? Well, when you run `eval-sequence`, you will need to evaluate the expressions in the procedure. `mc-eval` is called there.

To better understand mc-apply, we need to see how procedures and environments are represented in the underlying implementation.

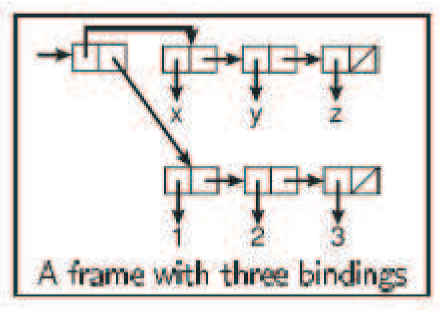
The representation of environments

An **environment** is essentially a list of frames, just like when we draw environment diagrams. The order of the frames in the list specifies the order in which we look up variables.



The global environment is represented by a list containing one frame, the global frame.

A **frame** is a list of variable names with their respective values. We chose to represent a frame as a pair pointing to two lists: a list of variable names and a list of variable values.



A frame with three bindings

Whenever we lookup a variable in an environment using `(lookup-variable-value var env)`, we start looking from the first frame, and proceed to the second frame if the variable is not found. If the variable is not found in any of the frames, it is said to be **unbound**.

Whenever we extend an environment using `(extend-environment vars vals base-env)`, we create a new frame with the new list of variables and values and cons it to the front of the list of environments.

Q: Describe what`(define-variable! var val env)` does.

A: It searches through the first frame for the variable. If the variable if found, it changes the value bound to the variable. If the variable is not found, it adds a new binding to the current frame (our implementation adds the binding to the front of the list of values and variables, though where you add it in the list does not matter as long the index of the value in the list is the same as the index of the variable in the list).

Q: Describe what `(set-variable-value! var val env)` does.

A: It searches through the first frame for the variable. If the variable if found, it changes the value bound to the variable. If the variable is not found, searches through the second frame for the variable, binding a new value to the variable if it is found in the second frame. It continues this process until it finds the variable or it reaches the last frame. If the variable is not found in any of the frames, it signals an error.

What must we have in our initial global environment when we load up the meta-circular evaluator?

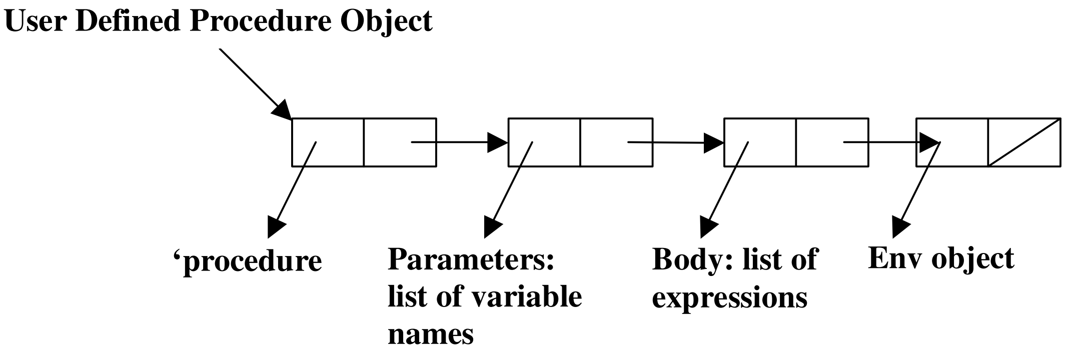
- We need to have primitive procedures! Variables like `+ - car cdr` … need to be bound to the procedures that represent them in our underlying implementation language. In the case of the meta-circular evaluator, the symbol `'+` is bound to the procedure `+` in the underlying scheme.
- We may also want to define default variables for convenience. For example, we can define `'true` and `'false` to represent `#t` and `#f` respectively.

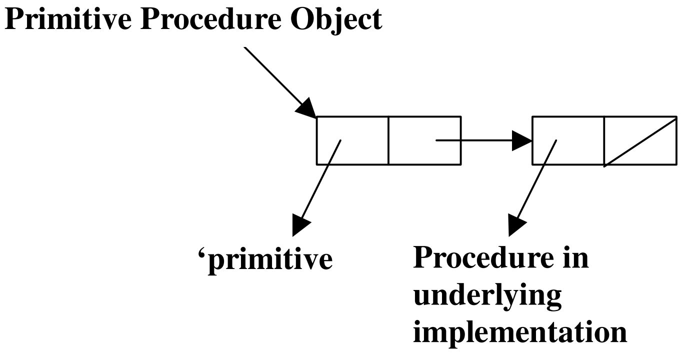
Below is the code that sets up the environment when you first initialize the meta-circular evaluator:

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))
```

The representation of procedures

Procedure objects are represented in Scheme as a list. There are two representations of procedures: one for user defined procedures and one for primitive procedures.


User Defined Procedure Object
‘procedure
Parameters: list of variable names
Body: list of expressions
Env object


Primitive Procedure Object
‘primitive
Procedure in underlying implementation

Now, we can look back at mc-apply and see how these procedure objects are used.

Confusion with `make-procedure`
A common point for confusion is thinking that `(make-procedure parameters body env)` does the same thing as

```
(make-begin seq)
(make-lambda parameters body)
(make-if predicate consequent alternative)
```

`make-procedure` creates a *procedure object* to be manipulated by `mc-eval` implementation, whereas the other make-x procedures listed above are used for making derived expressions.

Confusion with procedures used by `eval-define`
Another place where people often get confused is when they try to read `(definition-variable exp)` and `(definition-value exp)`, which are both used by `(eval-definition exp env)`. The reason for the if-statement is to decide whether they are evaluating
`(define var-name exp)`
or they are evaluating
`(define (proc args …) body)`
which will be translated to a derived expression

Driver Loop: the user interactive prompt

Now that we understand the fundamentals of mc-eval and mc-apply, we just need loop that reads in expressions and evaluates what the user types in to the interpreter.

```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (mc-eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

Printing Procedure Objects

You'll note that procedure objects have a pointer to their defining environments. Their defining environments may also have a variable name bound to the procedure object, and hence a pointer to the procedure. We have a cycle! If we tried to print the defining environment of a procedure, we may go into an infinite loop. As such, we need to check if a return value is a procedure before printing it (and not print it's environment if it's a procedure). This is done in (user-print).

Those are all the core parts of the meta-circular evaluator. Now, all you need to do is familarize yourself with it by doing more exercises from the book and figuring out which parts you need to modify if you want to add/take away a feature (or chance the evaluation model).

More Exercises

9. You will be adding the `let` special form to the interpreter as a derived expression for homework. We will do something harder. We will add write an evaluation procedure for it, `eval-let`. This function should take a `let` expression and an environment as argument, and perform the evaluation by hand.

10. Do Exercise 4.8 on Page 376 of *SICP* : implementing “named let”. This was the MCE
Hard! question on this summer’s final. Make sure you test it.