CS61A Notes – Week 12: Metacircular evaluator

Meta-metaevaluation

Remember Scheme-1? Of course you do. Well, now that you're 100% familiar and comfortable with a weak Scheme interpreter written in Scheme, let's move on to a (nearly) full-fledged Scheme interpreter in Scheme!

First, let's examine how the metacircular evaluator represents things in underlying Scheme. A primitive procedure is represented as list whose first element is the word `PRIMITIVE` and whose second element is the actual procedure:

```
(PRIMITIVE #[subr car]) ;; car in mceval
```

Non-primitive procedures are a bit more interesting. They're actually a list of four elements: the word `PROCEDURE`, a list of its parameters, a list of expressions in the body, and the environment it was *created in*:

```
(define (foo a b) (+ a b)) =>
  (PROCEDURE (a b) ((+ a b)) <the-global-environment>)
```

Does that last part sound familiar? If you know your environment diagrams, it should – the metacircular evaluator is more powerful than Scheme-1 by far, and the primary reason for that is because we've moved beyond the **substitution model** of Scheme-1 and are ready to handle the full **environment model**! Right guys? Hey, where'd you guys go?

Anyway. Let's look at how environments are represented. The pair structure handles the environment model perfectly; each environment corresponds to a pair whose its `car` points to the variable/value bindings (which we'll call a **frame** from now on) and whose `cdr` points to the next environment (just like in the environment model). The global environment, then, has a null `cdr`. What does the frame look like? Well, it's a pair whose car contains all of the variables, and whose cdr contains all of the values:

```
(let ((x 3) (y 5)) ...) =>
  (((x y) 3 5) <the-global-frame>) ;; the printout of environment 1
```

Of course, multiple environments can and will point to the same environment, so the entire environment diagram is not exactly a straight list structure. Notice, however, that it's simple to simulate evaluating a variable with this model! Simply check the current frame for a binding, and if it's not there, `cdr` to the next environment until we either find our variable or go past the global environment – in which case the next environment is null.

QUESTION

Write `lookup-variable-value`, which takes a variable and starting environment and returns the value associated with the variable or an error if it isn't found after the global environment.

Regular Metaevaluation

So all that above was simple, right? Now let's look at some code (with helpful comments written by yours truly):

```
(define (mc-eval exp env)
  (cond
   ((self-evaluating? exp) exp)
   ((variable? exp) (lookup-variable-value exp env))      ;; you just did this above
   ((quoted? exp) (text-of-quotation exp))                ;; (cadr exp)
   ((assignment? exp) (eval-assignment exp env))          ;; question 1 below
   ((definition? exp) (eval-definition exp env))          ;; question 2 below
   ((if? exp) (eval-if exp env))                          ;; essentially uses Scheme if
   ((lambda? exp)
    (make-procedure (lambda-parameters exp)               ;; (list 'procedure args)
                    (lambda-body exp)
                    env))
   ((begin? exp)
    (eval-sequence (begin-actions exp) env))
   ((cond? exp) (mc-eval (cond->if exp) env))             ;; makes nested ifs
   ((application? exp)
    (mc-apply (mc-eval (operator exp) env)
              (list-of-values (operands exp) env)))       ;; map mc-eval onto exps
   (else
    (error "Unknown expression type -- EVAL" exp))))

(define (mc-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments)) ;; use underlying Scheme apply
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment                            ;; question 3 below
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

One of the mysteries not covered above is `eval-sequence`. This is how `begin` statements and, more importantly, compound procedures are handled:

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps)                          ;; last-exp? => (null? (cdr exps)))
         (mc-eval (first-exp exps) env))           ;; first-exp => car
        (else (mc-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

Fairly uninteresting for `begin` statements, but notice how it's called in `mc-apply` for compound procedures – the evaluating environment is a new environment created using `extend-environment`, which you'll code in about 2 questions.

QUESTIONS

1.
```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)            ;; (cadr exp)
                       (mc-eval (assignment-value exp) env) ;; (caddr exp)
                       env)
  'okay)
```

Modify your `lookup-variable-value` code above to create `set-variable-value!` (which takes an additional value argument).

2.
```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)            ;; (cadr exp)
                    (mc-eval (definition-value exp) env) ;; (caddr exp)
                    env)
  'okay)
```

Modify your `set-variable-value!` code above to create `define-variable!`. You should write a helper `add-binding-to-frame!` that takes a variable, value, and frame, and adds the binding into the given frame.

3. **Write `(extend-environment vars vals base-env)` that takes in a list of variables, a list of values, and an environment to extend, and creates the new environment (as when you call a procedure in the environment model).**

4. **Scheme's `map` won't work in `mc-eval`. Why?**

5. **Write `(mc-map fn ls)` to work with `mc-eval`. It will be installed as the primitive procedure associated with `map`. `fn` is defined in our new representation.**

Dynamic Scope

The major difference between lexical and dynamic scope's `apply`: In lexical scope, we extend the *procedure environment* (right bubble) of the procedure we're invoking, whereas in dynamic scope, we extend the *current environment*. (Review: Which one does Scheme use?)

Note that in dynamic scope, the right bubble is entirely unnecessary. Dynamic scope tends to be much easier to implement and model, but lexical scope gives us a nice way to do **local state**. It is important to understand dynamic scope though, and it may prove to be of some relevance to you in the near future (*cough* `proj4`).

There are various advantages that one has over the other, and I'll let you read about those in the lecture notes.

Analyzing Evaluator

The intuition is quite easy. Just remember, we "compile" expressions into procedures that take in an environment. This is mainly for speeding up procedure calls (and note, NOT for just recursive procedures). For instance, in mc-eval, let's suppose I use the square procedure a lot.

Here is a sample call to square: (define (square x) (* x x)) (square 7)

[1] not self-evaluating, not a symbol ....

application: eval square, eval 7

apply square to operands: (7)

[2] Not primitive... It's a compound procedure:

extend environment, evaluate (* x x)

[3] eval (* x x): not self-evaluating, not a symbol ...

application: eval *, lookup x, lookup x,

apply *

Now, every time we call square, we have to go through mc-eval's cond clause, checking for what type of expression the body of square is (in step [3]). What if we could analyze the square procedure once, so that we KNOW what type of expression the body of square is? Save ourselves some trouble! Well, that's what analyzing evaluator does.

So, what we'd like to see, is that when calling square, in step [3], we "know" it's an application, so we jump straight into action:

[new step 1] lookup *, lookup x, lookup x, apply *.

How do we do this? First we analyze the expression. After analysis, then we package the information into a procedure:

(lambda (env) (apply (lookup * env) (list (lookup x env) (lookup x env))))

Then, every time we call square, we just call the above procedure with the appropriate environment. (i.e. for (square 8), give it an environment where x is 8). In analyzing-eval, the compiled expression won't look exactly like this, because we have to handle general cases.

So here's the model:

(define (analyzing-eval exp env) ((analyze exp) env))

Analyze the expression, then when it's time for evaluation, plug in the environment.

Analyze-lambda:

(lambda <param> <body>)

<body> =analyze=> <analyzed-body>

=======================================>

(lambda (env) (make-procedure <param> <analyzed-body> env))

Here is where most of the benefits of analysis will come. The difference is that we analyze the body BEFORE we make the procedure, so when it comes to calling this procedure, all we have to do is take the analyzed-body and pass in the appropriate environment (as we mentioned earlier about square)

Analyze-application: (for lambda-created procedures)

(<proc> <operands>)

<proc> =analyze=> <analyzed-proc>

<operands> =analyze=> <analyzed-operands>

===>

```
(lambda (env)
        (let ((proc (<analyzed-proc> env))
              (operands (map (lambda (a) (a env)) <analyzed-operands>)))
                  ((procedure-body proc)
                          (extend-environment (procedure-parameters proc)
                                              operands
                                              (procedure-environment proc)) ))
```

From the above, you can deduce the following rule:

In a given transcript, you will see speedup if some non-primitive procedure is called more than once!

Question about analyzing eval:

1. Which of the following would have speedup in analyzing

eval?

a. (+ 1 2) b. (((lambda (x) (lambda (y) (+ x y))) 5) 6) c. (map (lambda (x) (* x x)) '(1 2 3 4 5 6 7 8 9 10))

d. (define fib (lambda (n) (if (or (= n 0) (= n 1)) 1 (+ (fib (- n 1)) (fib (- n 2))))) (fib 5)

e. (define fact (lambda (x) (if (= x 0) 1 (* x (fact (- x 1))))))

f. (accumulate cons nil '(1 2 3 4 5 6 7 8 9 10