CS61A Notes – Disc 12: Metacircular evaluator, Analyzing evaluator

Meta-metaevaluation

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? The metacircular evaluator is just about as powerful as real Scheme, and the primary reason for that is because we're using applicative order and the **environment model**.

Let's look at how environments are represented. The pair structure handles the environment model; 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 a frame look like? Well, it's a pair whose `car` contains all of the variables, and whose `cdr` contains all of the values. Here's an example environment created through a `let` call:

```
(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 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.

You should keep the constructors and selectors for frames and environments in mind:

Frames	**Environments**
`make-frame ⇔ cons`	`extend-environment ⇔ ??? (see q3)`
`frame-variables ⇔ car`	`first-frame ⇔ car`
`frame-values ⇔ cdr`	`enclosing-environment ⇔ cdr`

Also, `(enclosing-environment the-global-environment)` is eq? to `the-empty-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)                ;; (cons '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))))
```

A lot of code, but remember `mc-eval` is just an implementation of environment diagrams. 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?**

EXTRA PRACTICE:

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 – "This is where the magic happens"

The intuition is quite easy. We want to "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. Let's look at the sample call `(square 7)`:

1. `(square 7)`: not self-evaluating, not a symbol...
   application: `eval square`, `eval 7`
   `apply square` to operands: `(7)`
2. `apply`: not primitive, compound procedure
   extend environment, `eval (* x x)`
3. `(* x x)`: not self-evaluating, not a symbol...
   application: `eval *`, `lookup x`, `lookup x`
   `apply *`

Annoying, isn't it? 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.

What if we could analyze the `square` procedure once so that we know what type of expression the body of square is? That's what the analyzing evaluator does:

1. `(analyze (square 7)) => <analyzed-procedure>`
   returns analyzed procedure that takes in an environment
2. `(<analyzed-procedure> env) => value`
   applies the body of the procedure to the operands

And that's it! Well, sort of. *How* do we actually do this? First, we analyze the expression. After analysis, 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. (Disclaimer: This isn't exactly how it looks, because `analyzing-eval` has to handle general cases.)

So here's the model: `(define (analyzing-eval exp env) ((analyze exp) env))`. Analyze the expression, and when it's time for evaluation, plug in the environment.

```
(define (analyze-lambda exp)
  (let ((vars (cadr exp))
        (analyzed-body (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars analyzed-body env))))
```

Here's 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.

```
(define (analyze-application exp) ;; for lambda-created procs, not directly from code
  (lambda (env)
    (let ((analyzed-proc (analyze (operator exp)))
          (analyzed-operands (map (lambda (a) (a env)) (map analyze (operands exp)))))
      ((procedure-body proc)
        (extend-environment (procedure-parameters analyzed-proc)
                            analyzed-operands
                            (procedure-environment analyzed-proc))))
```

So in the example of `square`, it first goes through `analyze-lambda`, then every time we want the value of `square` called with some argument, we use `analyze-application`.

If you haven't noticed by now, all the analyzing evaluator does is **package procedures into regular Scheme procedures**. But the idea is still important – if we can compile procedures into regular Scheme procedures, it's not much harder to compile it into something else, like machine language. This is what compiling is!

(Note: When you see `analyze-application` in the code, it won't look like above. But it's just split between `analyze-application` and `execute-application`.)

QUESTIONS

Which of the following would have speed up in `analyzing-evaluator`?

1. **`(+ 1 2)`**

2. **`(((lambda (x) (lambda (y) (+ x y))) 5) 6)`**

3. **`(map (lambda (x) (* x x)) '(1 2 3 4 5 6 7 8 9 10))`**

4.
```
(define fib
  (lambda (n)
    (if (or (= n 0) (= n 1)) 1
        (+ (fib (- n 1)) (fib (- n 2))))))
(fib 5)
```

5.
```
(define fact
  (lambda (x) (if (= x 0) 1 (* x (fact (- x 1))))))
```

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