MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2006

Recitation 22 — 5/5/2006
Interpretation – Dynamic Scoping

Missing Pieces

Sequences:

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (m-eval (first-exp exps) env))
        (else (m-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

Assignments:

```
(define (set-variable-value! var val env)
  (if (eq? env the-empty-environment)
      (error "Unbound variable -- LOOKUP" var)
      (let* ((frame (first-frame env))
             (binding (find-in-frame var frame)))
        (if binding
            (set-binding-value! binding val)
            (set-variable-value! var val (enclosing-environment env))))))
```

Syntactic Transformations

Look at the evaluation rules for `let` and `cond` in the meval text:

```
        ((cond? exp) (m-eval (cond->if exp) env))
        ((let? exp) (m-eval (let->application exp) env))
```

`cond->if` and `let->application` rewrite the expression as another expression and immediately pass it back to meval.

For example:

```
(let ((a 1)            =>      ((lambda (a b) (* a b))
      (b (+ 4 5)))              1
   (* a b))                     (+ 4 5))
```

Other special forms can be written the same way (`and`, `or`, `case`).

Dynamic vs Lexical Scoping

What we've seen so far in Scheme is *lexical scoping*: we bundle up the environment and store it with the double-bubble procedure object we create. When we call m-apply in the metacircular evaluator, we don't need to pass in an environment – it's already there in the procedure object. The rule is "When you apply a procedure, attach your new frame to the environment *in which the procedure was created.*"

Dynamic scoping works a bit differently. The rule here is now "When you apply a procedure, attach your new frame to the environment *of the procedure that called you.*"

1. What does this mean in terms of the environment diagram? What happens to our double-bubbles?

   Double bubbles (procedure objects) no longer need to remember where they were created, so they become single bubbles.

2. Draw environemnt diagrams for both lexical and dynamic scoping evaluations of the following:

```
(define pi 3)

(define (circ r) (* 2 pi r))

(define (test)
    (define pi 4)
    (circ 5))

(test)
```

   In the lexically scoped version, the outermost definition for pi (3) is used in the body of `circ`*, so the result is 30. In the dynamically scoped version,* `pi` *is 4 in the evaluation of the body of* `circ`*, so the result is 40.*

3. Time to be creatively destructive: come up with a series of function definitions and calls that works in lexical scoping, but breaks in dynamic scoping.

   How about:

```
(define (foo proc) (lambda (x) (proc x)))
((foo inc) 3)
```

   Something odd is going on: look at
   `(lambda (x) (proc x))`
   inside the first define. Ordinarily, with lexical scoping, you'd be able to tell that proc comes from the parameter just outside. With dynamic scoping, you have no such guarantee, and you won't even know whether or not proc is even defined at all! This kind of mysterious dynamic binding problem makes for unreadable, very confusing code. This was one of the biggest reasons for a move away from dynamic scoping and toward lexical scoping.

4. Let's change the metacircular evaluator to have dynamic scoping. What parts need to change in meval?

   Most things don't need to change. However, since we now have single bubbles instead of double bubbles for our procedure objects, we need to change the lambdas and applications. Procedure objects no longer have the environment attached, and applications need to know the environment from which they were called.

```
...
((lambda? exp)
 (make-procedure (lambda-parameters exp) (lambda-body exp))) ; No env!
...
((application? exp)
 (m-apply (meval (operator exp) env)
          (list-of-valuesss (operands exp) env)
          env))   ; New!
```

5. What needs to change in m-apply?

   m-apply now takes an extra argument: the environment of the calling procedure. And since the procedure object no longer has the environment attached, we need to attach the new frame onto the environment we were called from.

```
(define (m-apply procedure arguments env) ; Added env!
  (cond ...
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure)
                              arguments
                              env)))  ; Change!
        (else ...)))
```

Loop

Now, back to our normal lexical evaluation, let's add a new special form called `loop` to meval:

```
(loop (i 1 inc) (= i 4)
     (newline)
     (display (list i (fact i))))

(1 1)
(2 2)
(3 6)
(4 24)
; Value: done

(define start-list '(1 3 5))

(loop (lst start-list cdr) (null? lst)
  (newline)
  (display (fact (car lst))))

1
6
120
; Value: done
```

The syntax of loop is a follows. The first clause includes a loop variable (`i` in the first example), an expression whose value is the initial value of the variable (1 in the first example), and an increment procedure to apply to the loop variable on each iteration to create a new value for the loop variable (the value associated with inc in the first example). The next clause is an end test, an expression that will evaluate to true or false. The remaining expressions are the body of the loop.

The semantics of loop is as follows. The loop variable is initially set to the value of its initialization expression. The end test is then evaluated. If the value is true, the loop exits, and the symbol done is returned. If not, the expressions in the body of the loop are evaluated. The incrementation procedure is then applied to the loop variable, and that variable is then bound to the returned value. The process then repeats.

Each of the following procedures extracts elements of a loop. Complete the definitions (assume that each would be applied to a full loop expression).

1. `(define (loop-variable exp) YOUR-ANSWER)`

   `(first (second exp))`

2. `(define (loop-initial-value exp) YOUR-ANSWER)`

   `(second (second exp))`

3. `(define (loop-increment exp) YOUR-ANSWER)`

   `(third (second exp))`

4. `(define (loop-end-test exp) YOUR-ANSWER)`

   `(third exp)`

5. `(define (loop-body exp) YOUR-ANSWER)`

   `(cdddr exp)`

To implement the special form, we add a dispatch to eval, and create a new evaluation procedure:

```
(define (eval exp env)
   (cond ...
        ((loop? exp)  (eval-loop exp env))
         ...
         (application? exp) ...)
         (else ....)))

(define (eval-loop exp env)
    (eval-loop-doit (loop-variable exp)
                    (loop-initial-value exp)
                    (loop-increment exp)
                    (loop-end-test exp)
                    (loop-body exp)
                    env))

(define (eval-loop-doit var init next end bod env)
   (let ((new-env  (extend-environment
                       ANSWER-1
                       ANSWER-2
                       env)))
     (if  ANSWER-3               ; test to see if done
          ANSWER-4               ; value to return
          (begin  ANSWER-5       ; evaluate body
                  ANSWER-6))))   ; go to next iteration
```

1. Provide an expression for ANSWER-1. (Together with Question 2, this should create a new environment with the loop variable bound to a new value.)

   `(list var)`

2. Provide an expression for ANSWER-2.

   `(list (m-eval init env))`

3. Provide an expression for ANSWER-3 to determine if the loop has satisfied the end condition.

   `(m-eval end new-env)`

4. Provide an expression for ANSWER-4 to return the correct value from the loop.

   `'done`

5. Provide an expression for ANSWER-5 to evaluate the body of the loop.

```
(eval-sequence bod new-env)
```

6. Provide an expression for ANSWER-6 to handle the next loop iteration.

```
(eval-loop-doit var (m-eval (next var) new-env) next end bod new-env)
```