MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 26
Wrap-up: Lazy Eval, Dynamic Scoping

Applicative vs Normal Order evaluation

In applicative order execution (like regular Scheme), all procedure arguments are evaluated before applying the procedure. In normal order execution, procedure arguments are evaluated after applying the procedure, and then only if the result is needed to complete the evaluation of the procedure.

If there are no side effects or mutation, (another way of saying this would be to say that if all expressions were idempotent, meaning that you could evaluate the same expressions repeatedly without any other effects or different results), then the final returned value will be the same for either normal or applicative order application.

However, with mutation, the results will not be the same:

```
(define a 1)
(define b 1)

(define (foo x y)
  (+ x y y))

(define res (foo
              (begin (set! a (+ a 1))
                     a)
              (begin (set! b (* b 2))
                     b)))

(define v (cons a (cons b '())))
(cons res v)
```

In an applicative order Scheme, evaluating `(cons res v)` will return `(6 2 2)`. What will a normal order evaluation return? (assume no memoization). Draw diagrams for both.

A normal order (lazy) evaluator would make it easy to define procedures that would need to be special forms in standard Scheme:

```
(define (unless test a b)
   (if test b a))
```

In an applicative order evaluator `unless` would need to be a special form, because we don't want to evaluate both `a` and `b`.

Lazy Evaluator

Let's change meval to be fully lazy. Only a few changes are needed:

1. In m-apply, add an environment argument, and if the procedure is a primitive, force the argument values and apply the primitive procedure.
2. Also in m-apply, if the procedure is a compound, delay evaluating the arguments – extend the environment of the procedure with *thunks* – promises to evalue the arguments later in the environment passed to apply.
3. In m-eval, if the procedure is an application, force the value of the procedure (so that mapply can tell whether it is a primitive or compound), but pass the expressions for the arguments to apply.
4. Conditionals (`if`, `cond`, etc.) need to force their predicates to know which branch to take.
5. helper procedures for dealing with thunks. `actual-value` and `force-it`. Also, the driver loop needs to use `actual-value` to show the final answer.

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)
  (let ((pi 4))
    (circ 5)))

(test)
```

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 square) 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 ...)))
```

Lexical vs dynamic scope

```
(let ((x 2))
  (let ((f (lambda (y) (- y x))))
     (let ((x 1))
        (f 3))))
```

1. Value in a dynamically-scoped Scheme:

2. Value in a lexically-scoped Scheme: