Variations on a Scheme

Lecturer:
Prof. Albert R. Meyer

November 18, 1997

Where We Left Off

- Built a *Meta-Circular Interpreter* FOR Scheme IN Scheme
- evaluate an IF

```
(define (eval-if exp env)
  (if (true? (meval (if-predicate exp) env))
      (meval (if-consequent exp) env)
      (meval (if-alternative exp) env)))
```

- As in environment model, Compound Procedures, say

```
(lambda (x y) (* x x y))
```

is represented with three parts:

1. the PARAMETERS `(x y)`
2. the TEXT `(* x x y)`
3. the ENVIRONMENT in which procedure was **created**

evaluate an APPLICATION

```
((application? exp)
 (MAPPLY (meval (operator exp) env)
         (list-of-values (operands exp) env)))

(define (list-of-values exps env)
  (map (lambda (rand) (meval rand env))
       (operands exps)))

(define (mapply procedure arguments)
  (cond ...
        ((compound-procedure? procedure)
         (meval
          (procedure-text procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        ...))
```

What was the Point?

After all it's CIRCULAR!

We do happen to learn something about Scheme

The evaluator is FIRST ORDER:
no procedure parameters or returned procedures.

(That's why text does not use **`MAP`** as we did above.) So the interpreter itself can have

STACK IMPLEMENTATION of ENVIRONMENTS

The environment frame of a procedure call can be discarded when the call returns.

But that's NOT THE POINT!

Interpreter Technology

The meta-circular interpreter illustrates how to build interpreters *in general*, not necessarily for Scheme.

We implemented Scheme first because you need to know the language you're implementing, and we all know Scheme by now.

For example, things get interesting, and not circular, if we implement LAZY SCHEME.

Lazy Scheme

Lazy (aka Normal Order) evaluation means
"Do not evaluate the arguments of a procedure until you have to"

```
(define (pick-one sym x y z)
  (cond
   ((eq? sym 'x) x)
   ((eq? sym 'y) y)
   ((eq? sym 'z) z)
   (else 'who-cares)))

;;; L-Eval input:
(pick-one
 (begin (newline) 'y)
 (display 'did-x)
 (display 'did-y)
 (display 'did-z))
                         ;TWO NEWLINES

did-y                    ;***displayed symbol
;;; L-Eval value:
#[undefined-value]      ;value from DISPLAY
```

Evaluate a LAZY APPLICATION

```
((application? exp)
 (lapply (actual-value (operator exp) env)
         (operands exp)  ;*** not their values
         env))

(define (lapply procedure exps env)
  (cond ((primitive-procedure? procedure)
                 ;;;;;FOR PRIM-OPS EVAL THE ARGS:
          (apply-primitive-procedure
           procedure
            (map
             (lambda (exp) (actual-value exp env))
             exps)))
         ...))
```

For Compound-Procedures, Delay the Args

```
((compound-procedure? procedure)
 (leval
  (procedure-text procedure)
  (extend-environment
   (procedure-parameters procedure)
   (list-of-delayed-args arguments env) ; CHANGED
   (procedure-environment procedure))))

(define (list-of-delayed-args exps env)
  (map (lambda (exp) (delay-it exp env)) exps))
```

Delayed Values

```
(define (actual-value exp env)
  (force-it (leval exp env)))

(define (delay-it exp env)  ;makes a THUNK
  (list 'thunk exp env))

(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj)
                    (thunk-env obj))
      obj))
```

Call By Need

Once a value has been forced, remember it and don't force again: MEMOIZATION as for streams.

```
(define (force-it obj)
  (cond ((thunk? obj)
          (let ((result (actual-value
                          (thunk-exp obj)
                          (thunk-env obj))))
            (set-car! obj 'evaluated-thunk)
                     ; replace exp with its value:
            (set-car! (cdr obj) result)
                              ; forget unneeded env:
            (set-cdr! (cdr obj) '())
            result))
         ((evaluated-thunk? obj)
          (cadr obj))
         (else obj)))
```

Small Change in Code, Large Significance

CORRECTION NOTED 11/19: The following example claims to illustrate how Call-by-Need evaluation can change exponential time to linear time behavior. But LAZY evaluation does NOT speed up this example. (The example correctly describes the effect of memoizing the **`fib`** *procedure* as in SICP, Exercise 3.27, but Call-by-Need memoizes the *arguments* of the procedure, which is a different matter.)

A proper example will be placed here sometime.

```
(define (fib n)
  (if (< n 1)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

With Call-by-Need:

```
Time(n) =  4     + Time(n-1) +        1
       (<,+,-,1)          MEMOIZED COST of fib (n-2)
```

SO

$$\text{Time}(n) = \Theta(n)$$

EXPONENTIAL becomes **LINEAR** – WITHOUT CHANGING THE CODE FOR **`fib`**!

Small Change in Code, Large Significance

DYNAMIC SCOPE:

```
((application? exp)
 (DAPPLY (meval (operator exp) env)
         (list-of-values (operands exp) env)
         env))        ;;;*****

(define (dapply procedure arguments env)
  (cond ...
        ((compound-procedure? procedure)
         (meval
          (procedure-text procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           ENV)))  ;;** NOT PROCEDURE-ENVIRONMENT
        ...))
```

DYNAMIC SCOPE:

```
(define (times x)
  (lambda (y) (* x y)))

(define (mul x y)
  ((times x) y))

(mul 3 4) ==> 12
```

REVERSE PARAMETER NAMES:

```
(define (mul x y)
  ((times y) x))

(mul 3 4) ==> 16 !!
```

Lots of Variations

- PS8 Applicative order (Scheme), AND Lazy AND Lazy-memo
- New procedures: ”reflective” **`EVAL`** within Scheme
- New Special forms, RETURN, GOTO
- NEW SYNTAX
- ANALYZE evaluator
- AMB evaluator
- ...

With this technology, you tailor your language to your application