MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Lecture Notes – November 5, 1996

Variations on a Scheme

Eval from the perspective of a language implementor: We can implement the language by implementing the evaluator.

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp)
         (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp)
         (eval-assignment exp env))
        ((definition? exp)
         (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp)
                                env)))
        (else
         (error "Unknown exp -- EVAL" exp))))
```

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure
                                    arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure -- APPLY"
           procedure))))
```

Why is this so inefficient?

Redesign the evaluator to analyze each expression only once:

Analyze returns `(lambda (env) ....)`

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else
         (error "Unknown expression type -- ANALYZE" exp))))
```

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application (fproc env)
                           (map (lambda (aproc) (aproc env))
                                aprocs)))))
```

```
(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))))
        (else
         (error
          "Unknown procedure type -- EXECUTE-APPLICATION"
          proc))))
```

Eval from the perspective of a language designer: We can think about new features in the language, and try them out, by modifying the evaluator.

To implement normal order evaluation, we simply change the application clause of eval.

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

and we create a new `apply`

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env))) ; ***
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env) ; ***
           (procedure-environment procedure))))
        (else
         (error "Unknown procedure type -- APPLY" procedure))))
```

Representing thunks is easy:

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

We create delayed objects by:

```
(define (delay-it exp env)
  (list 'thunk exp env))

(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk) (cadr thunk))

(define (thunk-env thunk) (caddr thunk))
```