

# ***Scheme in Scheme: The Metacircular Evaluator*** ***Eval and Apply***

*CS21b: Structure and Interpretation of Computer Programs*  
Brandeis University  
Spring Term, 2015



# The *metacircular evaluator* is

A rendition of Scheme, *in Scheme*, where expressions and environments are just *list structures* that get modified according to fixed rules.

A concise language description (as opposed to, say, English) -- which, of course, you need to know Scheme already to understand.

A kind of *fixed point* of the Scheme interpreter.  
(**factorial** is not a fixed point.)

A straightforward medium to discuss language modifications among language designers

No more complicated than many of the programs we have discussed in class

*What you are about to learn is powerful, secular wizardry.  
Your life will never be the same again.*

# Evaluator -- top level

```
(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 expression type – EVAL" exp))))
```

# Apply (coroutines with Eval)

```
(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 type – APPLY" procedure))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

# Syntax-directed evaluation (examples)

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

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

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                        (eval (assignment-value exp) env)
                        env)
  'ok)

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))
```

# Environments

```
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

(define (make-frame variables values)
  (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

# Environments (variable lookup)

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

# Environments (variable lookup)

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
              (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
              (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable – SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame))))))
  (env-loop env))
```

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
              (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
              (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
      (scan (frame-variables frame)
            (frame-values frame))))
```



# Environments (variable lookup)

```
(define (setup-environment)
  (let ((initial-env
        (extend-environment
         (primitive-procedure-names)
         (primitive-procedure-objects)
         the-empty-environment)))
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    initial-env))
```

# Driver loop

```
(define input-prompt ";;; M-Eval input:")
(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
    (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))

(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                    (procedure-parameters object)
                    (procedure-body object)
                    '<procedure-env>))
      (display object)))
```

```
;;; M-Eval input:  
5  
;;; M-Eval value:  
5
```

```
;;; M-Eval input:  
(+ 5 7)  
;;; M-Eval value:  
12
```

```
;;; M-Eval input:  
((lambda (x) (* x x)) 5)  
;;; M-Eval value:  
25
```

```
;;; M-Eval input:  
(define square (lambda (x) (* x x)))  
;;; M-Eval value:  
ok
```

```
;;; M-Eval input:  
(square 5)  
;;; M-Eval value:  
25
```

```
;;; M-Eval input:  
(define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))  
;;; M-Eval value:  
ok
```

```
;;; M-Eval input:  
(fact 5)  
;;; M-Eval value:  
120
```

## Using the metacircular evaluator...