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

Lecture Notes – October 31, 1996

The Metacircular Evaluator

Today we shift perspective from that of a **user** of computer langugaes to that of a **designer** of computer languages.

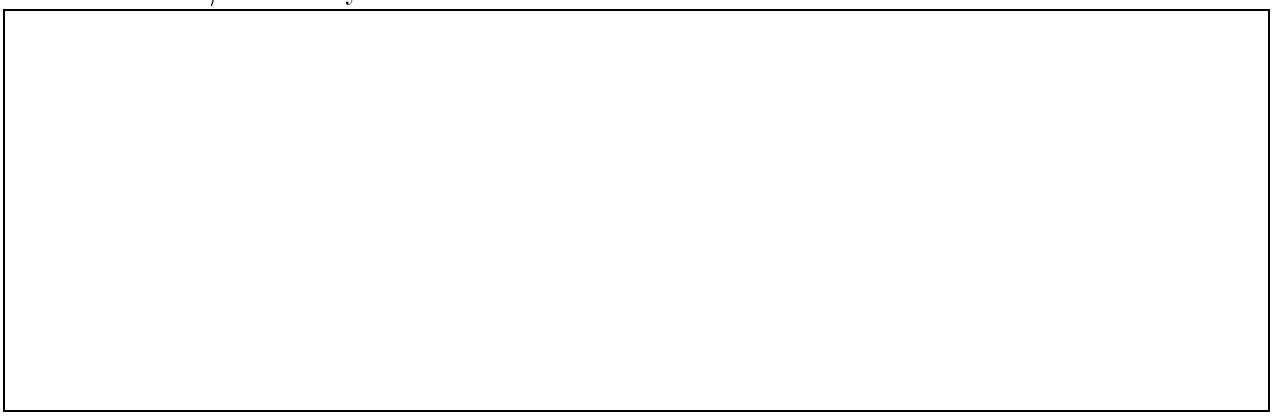
EVAL is an example of a UNIVERSAL machine. That is, it can simulate any other machine.

Remember the rules of the environment model:

1. To **EVALUATE** a combination (a compound expression other than a special form), evaluate the subexpressions and then **APPLY** the value of the operator subexpression to the values of the operand subexpressions.

2. To **APPLY** a compound procedure to a set of arguments, **EVALUATE** the body of the procedure in a new environment. To construct this environment, extend the environment part of the procedure object by a frame in which the formal parameters of the procedure are bound to the arguments to which the procedure is applied.

Draw the EVAL/APPLY cycle here:



Assume we have some way of representing environments together with the following operations:

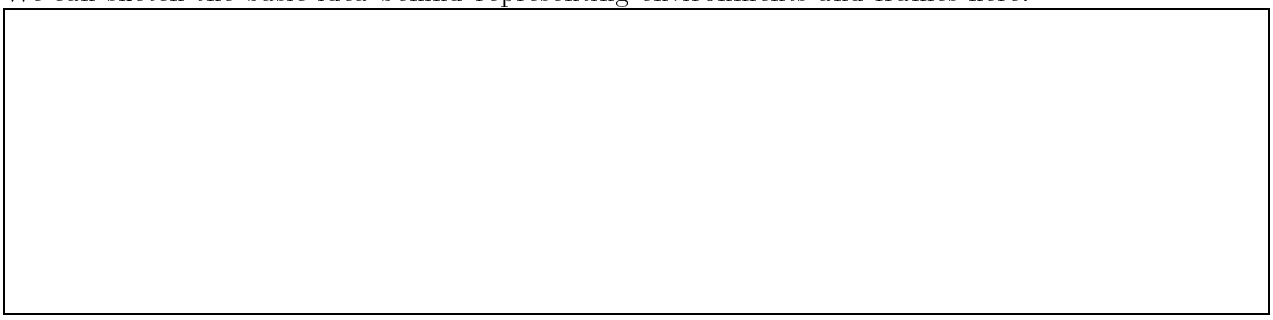
`(lookup-variable-value symbol env)`

–returns the value bound to the symbol in the environment.

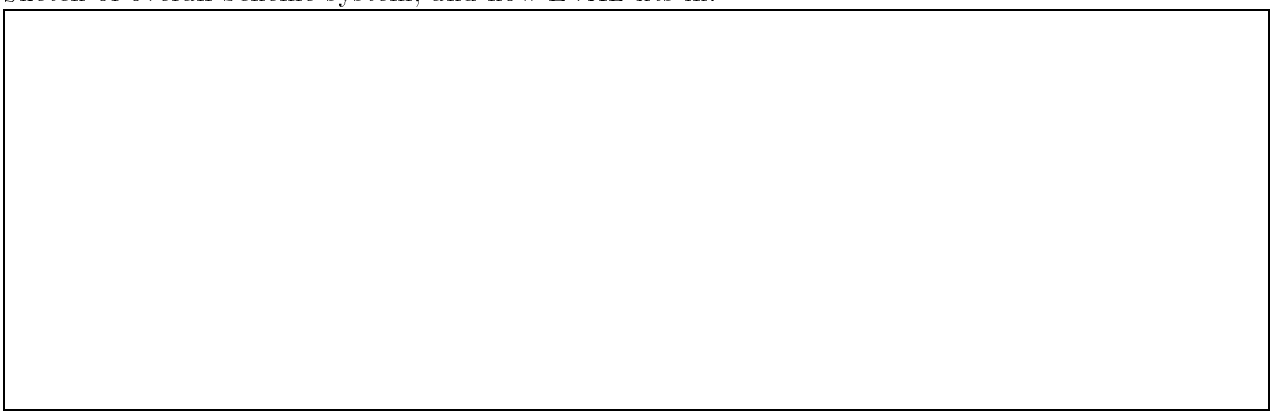
`(extend-environment vars vals base-env)`

– returns a new environment, which extends `base-env` by a frame in which the `vars` are bound to the corresponding `vals`.

We can sketch the basic idea behind representing environments and frames here:



Sketch of overall Scheme system, and how EVAL fits in:



When you type the expression: `(* x (+ 2 y))`, EVAL sees that as `(eval '(* x (+ 2 y)) the-global-env)`. In general, your programs are EVAL's data.

Here is the basic form for EVAL – note the primities, the special forms, and the reduction to `apply`.

```
(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))))
```

Examples:

`quoted?`

`lambda?`

`if?`

Evaluation of `if` expressions:

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

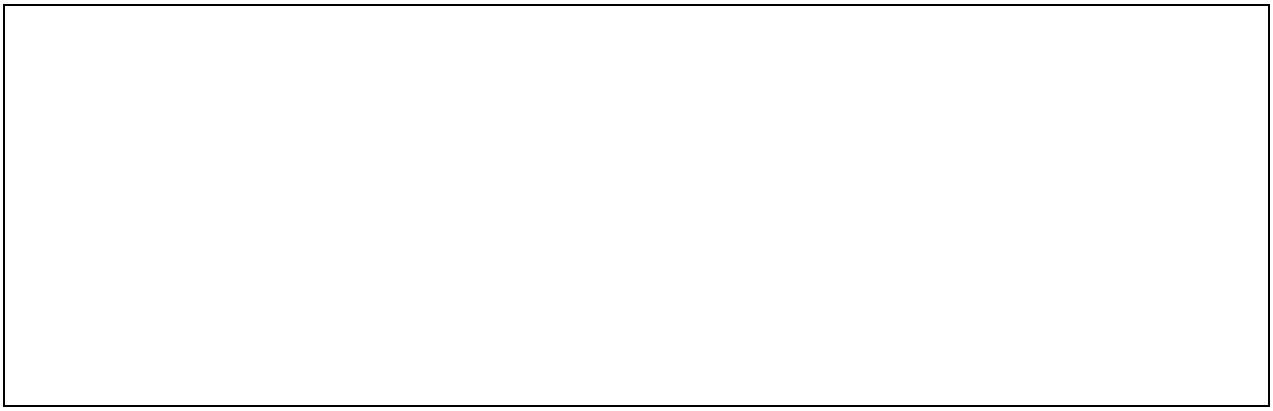
`Begin` expressions indicate sequences to be evaluated in order:

```
(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))))
```

`Cond` is implemented as a *derived expression*

```
(cond ((> x 0) x)
      ((= x 0) 0)
      (else (- x)))
```

can be converted into



Key clause in `EVAL` that handles procedure applications:

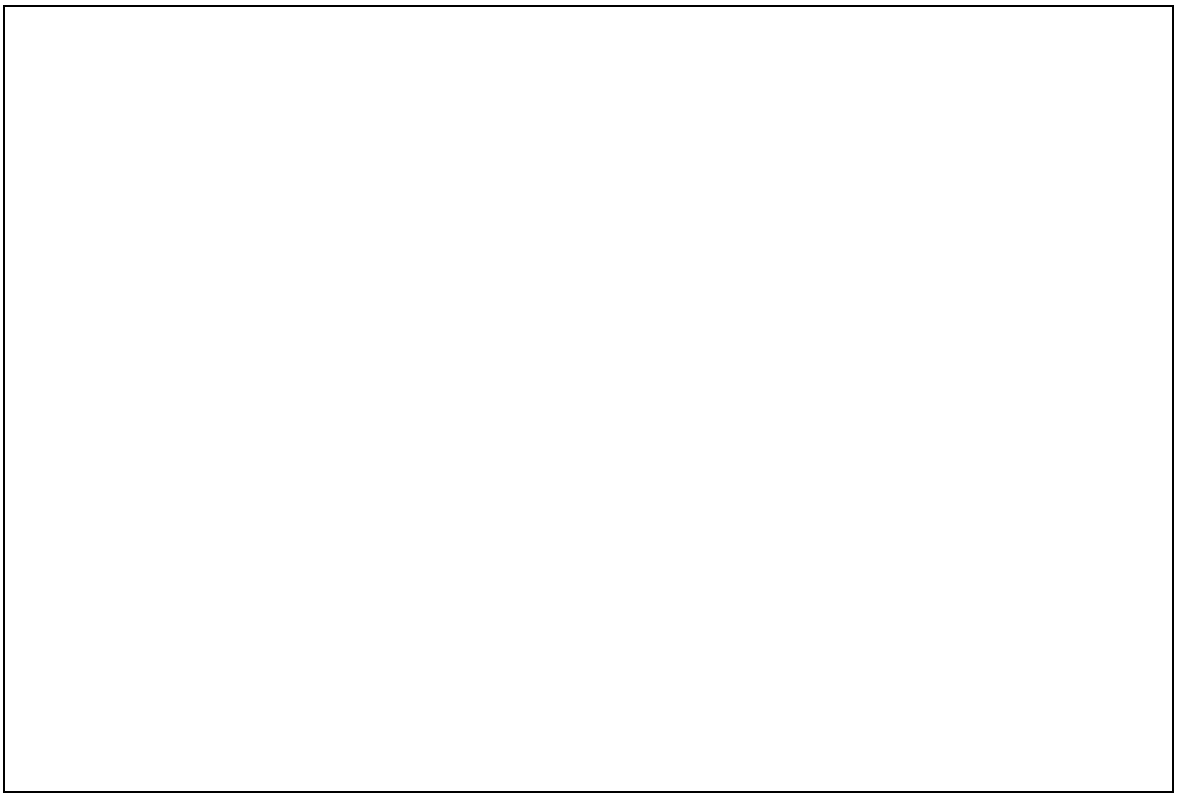
```
((application? exp)
 (apply (eval (operator exp) env)
        (list-of-values (operands exp)
                        env)))
```

`APPLY`: the other half of the cycle.

```
(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))))
```

Trace through the evaluation of

```
((lambda (x y) (* x y)) 5 7)
```



Finally, here is stripped down version of the evaluator:

```
(define mc-eval
  (lambda (exp env)
    (cond ((number? exp) exp)                    ;base-case
          ((string? exp) exp)                    ;base-case
          ((symbol? exp) (lookup exp env))       ;base case
          ((eq? (car exp) 'quote) (car (cdr exp)));special forms
          ((eq? (car exp) 'cond) (evcond (cdr exp) env))
          ((eq? (car exp) 'sequence) (evseq (cdr exp) env))
          ((eq? (car exp) 'lambda) (list 'proc (cdr exp) env))
          ((eq? (car exp) 'define) (evdefine (cdr exp) env))
          (else (mc-apply (mc-eval (car exp) env)
                          (evlist (cdr exp) env))))))

(define mc-apply
  (lambda (fun args)
    (cond ((atom? fun) (apply fun args))         ;ground out
          ((eq? (car fun) 'proc)
           (mc-eval (car (cdr (car (cdr fun))))  ;procedure body
                    (bind (car (car (cdr fun)))  ;formal params
                          args                   ;supplied args
                          (car (cdr (cdr fun))))));saved env
          (else (error '"Unknown function")))))

(define evlist
  (lambda (lst env)        ;map evaluator over list
    (cond ((null? lst) nil)
          (else (cons (mc-eval (car lst) env)
                      (evlist (cdr lst) env))))))

(define evcond
  (lambda (clauses env)
    (cond ((null? clauses) nil)
          ((eq? 'else (car (car clauses))) (evseq (cdr (car clauses)) env))
          ((mc-eval (car (car clauses)) env) (evseq (cdr (car clauses)) env))
          (else (evcond (cdr clauses) env)))))

(define evseq
  (lambda (clauses env)
    (cond ((null? (cdr clauses)) (mc-eval (car clauses) env))
          (else (mc-eval (car clauses) env)
                (evseq (cdr clauses) env))))))

(define evdefine
  (lambda (body env)     ;mutate the first frame
    (sequence (set-cdr! (car env)
                        (cons (cons (car body) (mc-eval (car (cdr body)) env))
                              (cdr (car env))))
              (car body))))
```

```
(define bind
  (lambda (params values env) ;add a new frame
    (cons (cons 'frame (make-frame-body params values)) env)))

(define make-frame-body
  (lambda (params values) ;frame body is an association list
    (cond ((null? params)
           (cond ((null? values) nil)
                 (else (error '"Too many values supplied"))))
          ((null? values) (error '"Too few values supplied"))
          (else (cons (cons (car params) (car values))
                      (make-frame-body (cdr params) (cdr values)))))))

(define lookup
  (lambda (var env)
    (cond ((null? env) (error '"Unbound variable" var)) ;not in any frames
          (else ((lambda (binding)
                   (cond ((null? binding) ;check each frame
                          (lookup var (cdr env))) ;in turn
                         (else (cdr binding)))) ;(<varname> . <value>)
                 (find-binding var (cdr (car env))))))))

(define find-binding
  (lambda (var frame-body) ;this is just assq
    (cond ((null? frame-body) nil)
          ((eq? var (car (car frame-body))) (car frame-body))
          (else (find-binding var (cdr frame-body))))))

(define global-env
  (list (list 'frame
              (cons 'nil '()) (cons '+ +) (cons '- -) (cons '= =) (cons '* *)
              (cons 'car car) (cons 'cdr cdr) (cons 'cons cons)
              (cons 'list list)
              (cons 'set-car! set-car!) (cons 'set-cdr! set-cdr!)
              (cons 'null? null?) (cons 'eq? eq?) (cons 'atom? atom?)
              (cons 'number? number?) (cons 'string? string?)
              (cons 'symbol? symbol?)
              (cons 'error error) (cons 'apply apply))))
```