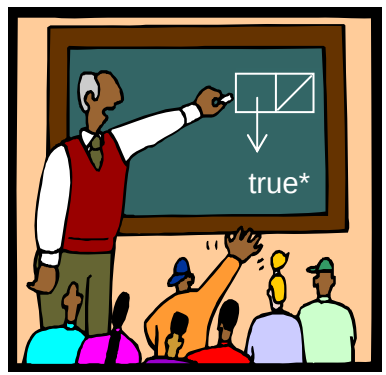
6.001 recitation 18 4/25/07

- interpretation
- our evaluator



Dr. Kimberle Koile

interpretation key ideas

. abstractions

. cycle

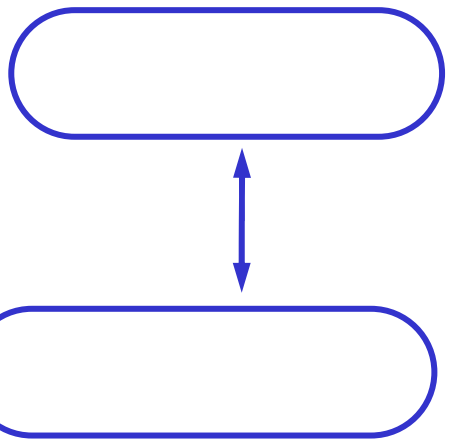
. environment

. parts

stages of an interpreter

input to each stage

```
"(average 40 (+ 5 5))"
```



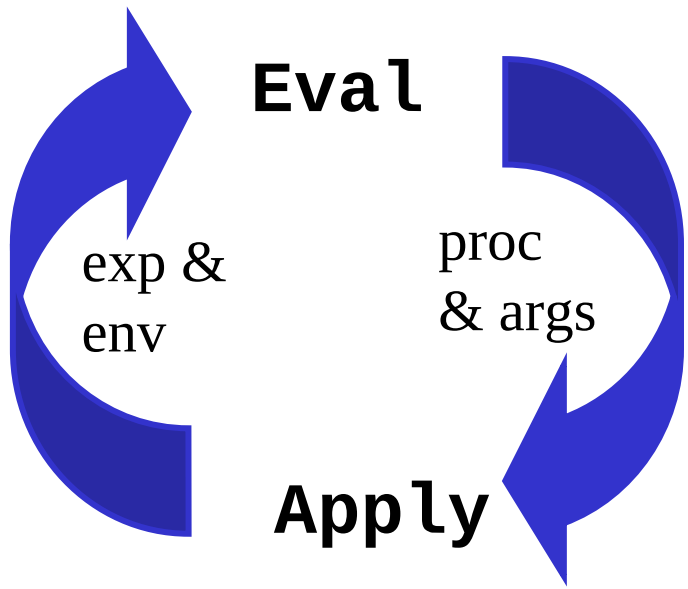
25

```
"25"
```

our evaluator

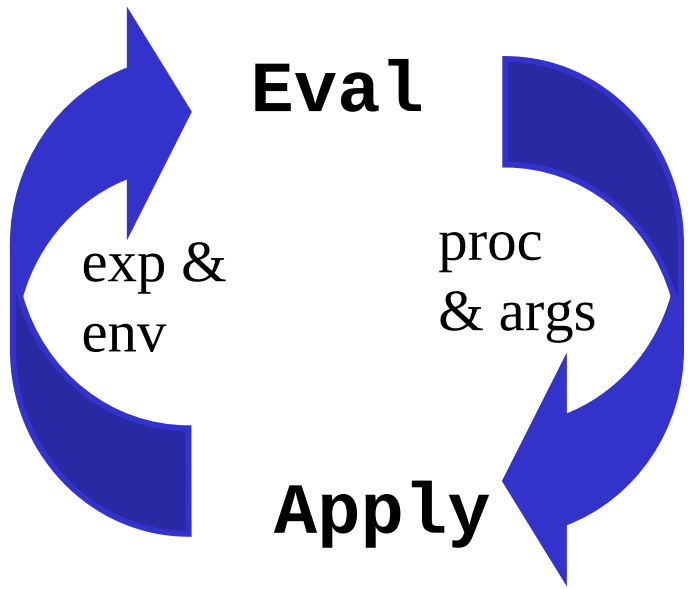
eval: dispatch on expression type

apply: eval args then apply operator



our evaluator

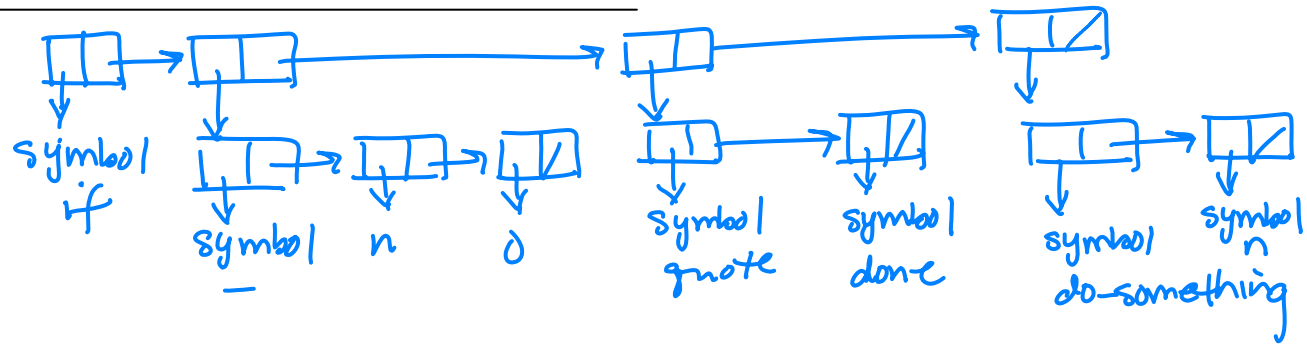
```
; the initial global environment
(define GE
  (extend-env-with-new-frame
   (list 'plus* 'greater*)
   (list (make-primitive +) (make-primitive >))
   '( )))

(define (eval exp env)
  (cond
   ((number? exp) exp)
   ((symbol? exp) (lookup exp env))
   ((define? exp) (eval-define exp env))
   ((if? exp) (eval-if exp env))
   ((lambda? exp) (eval-lambda exp env))
   ((let? exp) (eval-let exp env))
   ((application? exp)
            (apply* (eval (car exp) env)
                    (map (lambda (e) (eval e env))
                         (cdr exp))))
   (else
    (error "unknown expression " exp))))
```



our evaluator

example

exp: (if (= n 0)
'done
(do-something n))




```
(define (eval exp env)
  (cond
   …

   ((if? exp) (eval-if exp env))

   … ))
```

our evaluator

example

exp: (if (= n 0)
'done
(do-something n))

```
(define (eval exp env)
  (cond
   …

   ((if? exp) (eval-if exp env))
   … ))
```

```
(define (if? exp) (tag-check exp 'if*))

(define (eval-if exp env)
  (let ((predicate (cadr exp))
        (consequent (caddr exp))
        (alternative (cadddr exp)))
    (let ((test (eval predicate env)))
      (cond
       ((eq? test #t) (eval consequent env))
       ((eq? test #f) (eval alternative env))
       (else (error "predicate not a conditional: "
                    predicate))))))
```

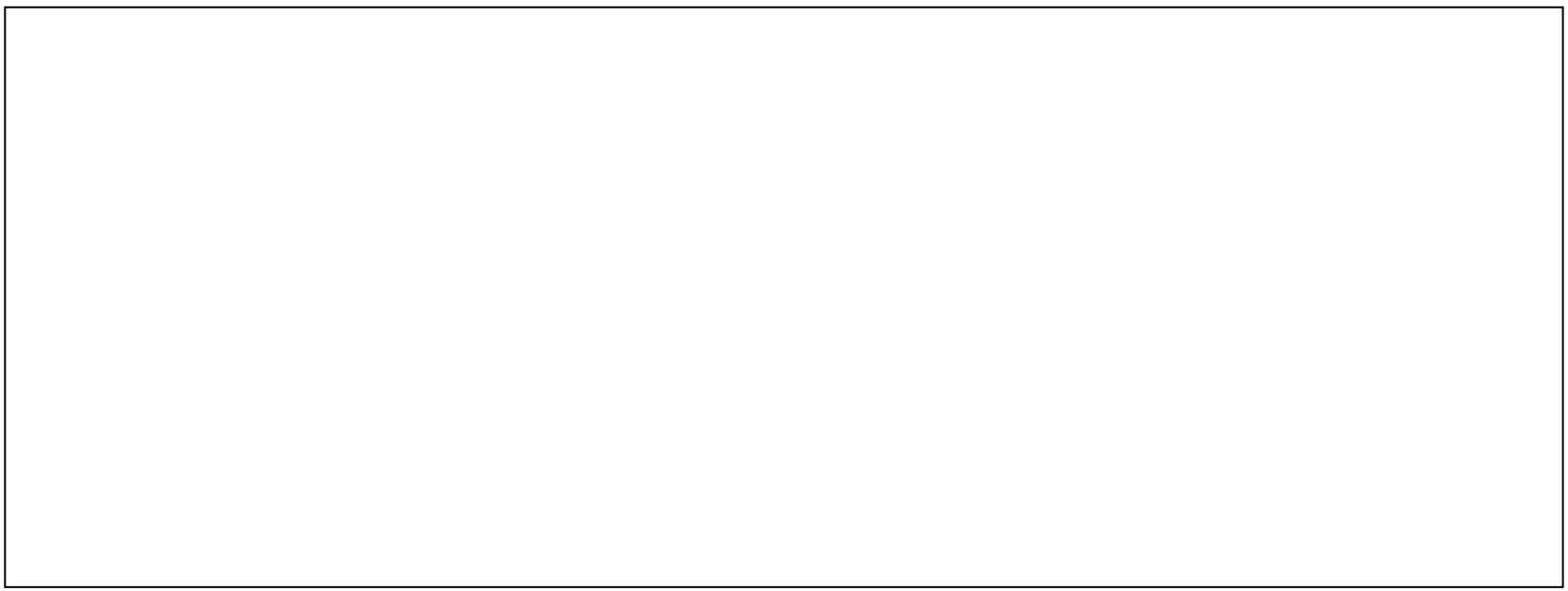
when

e.g. (when (= x 0) (print "zero"))

semantics is same as if without alternate clause

quote*

1. (**quote*** expr) returns expr without evaluating it. Assume eval calls eval-quote if the procedure quote? is true for a given quote* statement. Write **eval-quote**, which takes one argument.



eval-sequence

2. (**eval-sequence** exps env) evaluates each expression is a list of expressions, and returns the value of the last one. Assume eval calls eval-sequence if the procedure sequence? is true for a given expression. Write **eval-sequence**, which takes two arguments, expr and env.

(Hint: You'll need to call begin.)

case*

3. (**case*** expr
   ((val1 val2 …) consequent)
   ((vali valj …) consequent)
   …
   (else* alternative))

Case* evaluates expr and compares its value (using eqv?) against each of the listed values, which are not evaluated. When a match is found, the corresponding consequent expression is evaluated and returned as the result of the case*. If no matches are found, the alternate expression is evaluated and returned instead. You can assume the else* clause is required if you like.

Assume eval calls eval-case if the procedure case? is true for a given case* statement.

```
(define (eval-case exp env)
   (let ((target-value (eval (second exp) env)))
     (eval-case-clauses target-value (cddr exp) env)))
```

On the next slide, write **eval-case-clauses**, which takes three arguments: a target-value, a list of clauses, and env.

assume that you already have these procs

```
(define (else-clause? clause)
        (tag-check clause 'else*))

(define (value-found? target values)
        (not (null? (memv target values))))
```

case* (cont'd)

3. (**case*** expr
((val1 val2 …) consequent)
((vali valj …) consequent)
…
(else* alternative))

```
(define (eval-case exp env)
  (let ((target-value (eval (second exp) env)))
    (eval-case-clauses target-value (cddr exp) env)))
```

Write **eval-case-clauses.**

begin*

4. (**begin*** expr1 expr2 … exprn) evaluates each expression in the sequence, returning the value of exprn as its final result. Assume eval calls eval-begin if the procedure begin? returns true for a given begin* statement.

```
(define (begin? exp)  (tag-check exp 'begin*))

(define (eval-begin exp env)
   (eval-begin-body (cdr exp) env))
```

Write eval-begin-body, which takes two arguments, body and env.