6.001 recitation 18         4/25/07

- interpretation
- our evaluator

Dr. Kimberle Koile
interpretation  key ideas

- abstractions
  - unwind them at execution time

- cycle
  - between eval + apply
  - eval calls apply with operator + arg
  - apply calls eval with expression + environment
  - eval = cond statement that dispatches on type

- environment
  - represented as a table
  - application = extend environment with a new frame

- parts
  - lexical analyzer
  - parser
  - evaluator
  - printer
Stages of an interpreter

- Lexical analyzer
- Parser
- Evaluator
- Environment
- Printer

Input to each stage:

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

Evaluator steps:

- Symbol `average`
- Value `40`
- Symbol `+`
- Value `5`
- Value `5`

Output:

```
25
```
our evaluator

eval: dispatch on expression type

apply: eval args then apply operator
; 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))))
(define (eval exp env)
  (cond ...
  ...
  ((if? exp) (eval-if exp env))
  ...
))
our evaluator

Example

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

\[
\text{semantic: if predicate is true eval consequent otherwise eval alternate}
\]

\[
\text{(define (eval exp env)}
\text{ (cond ...)}
\text{((if? exp) (eval-if exp env)) ... ))}
\]

```
(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

```
(define (eval-when exp env)
  (let ((test-expr (cadr exp))
        (consequent (caddr exp)))
    (if (eval test-expr env)
        (eval consequent env)
        #f)))
```
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.

```
(define (eval-quote exp)
  (read exp))
```

```racket
  or cash
```
2. **(eval-sequence exps env)** evaluates each expression in 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.)

```lisp
(define (eval-sequence exps env)
  (if ((null? (cdr exps))
       (eval (first-exp exps) env))
    (eval (first-exp exps) env)
    (begin (eval (first-exp exps) env)
            (eval-sequence (rest-exps exps) env))))
```
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.
3. (case* expr
    ((val1 val2 ...) consequent)
    ((vali valj ...) consequent)
    ...
     (else* alternative))

Write eval-case-clauses.
4. \((\textbf{begin*} \ expr1 \ expr2 \ldots \ exprn)\) evaluates each expression in the sequence, returning the value of \(\text{exprn}\) as its final result. Assume \texttt{eval} calls \texttt{eval-begin} if the procedure \texttt{begin?} returns true for a given \texttt{begin*} statement.

\[(\text{define (begin? exp) (tag-check exp 'begin*)})\]

\[(\text{define (eval-begin exp env)}\]
\[\quad (\text{eval-begin-body (cdr exp) env})\]
\[\text{Write \texttt{eval-begin-body}, which takes two arguments, body and env.}\]

\[(\text{define (eval-begin-body body env)}\]
\[\quad (\text{if (null? body) 'undefined}\]
\[\quad \quad (\text{let (value (eval (car body) env))}\]
\[\quad \quad \quad (\text{if (null? (cdr body)) value}\]
\[\quad \quad \quad \quad (\text{eval-begin-body (cdr body) env})))\]