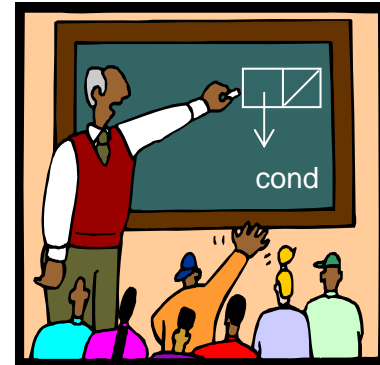


6.001 recitation 19

4/27/07

- our evaluator (cont'd)



Dr. Kimberle Koile

extending our evaluator: 2 ways

- > direct evaluation: add as new clause in toplevel `eval` procedure
- > syntactic translation: translate the expression into a known type

See `eval-if`, `eval-sequence`

See `cond->if`

our evaluator: direct

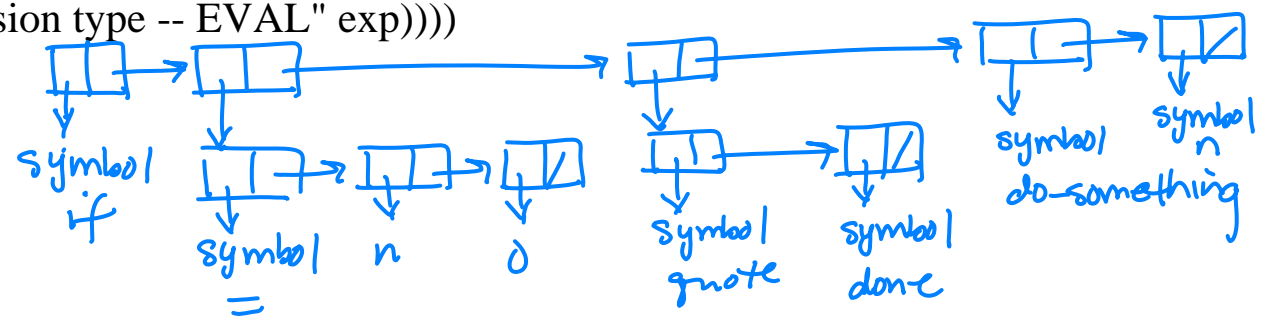
```
(define (eval exp env)
  (cond
    ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp env))
    ((quoted? exp) (text-of-quotation exp))
    ((assignment? exp) (eval-assignment exp env))
    ((definition? exp) (eval-definition exp env))
    ((if? exp) (eval-if exp env))
    ((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))
    ((and? exp) (eval (and->if exp) env))
    ((until? exp) (eval-until exp env))
    ((application? exp)
     (mapply (eval (operator exp) env)
              (list-of-values (operands exp) env)))
    (else (error "Unknown expression type -- EVAL" exp))))
```

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

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

example

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



our evaluator: syntactic

```
(define (m-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) (m-eval-sequence (begin-actions exp) env))
    ((cond? exp) (m-eval (cond->if exp) env))
    ((and? exp) (m-eval (and->if exp) env))
    ((until? exp) (eval-until exp env))
    ((application? exp)
     (mapply (m-eval (operator exp) env)
              (list-of-values (operands exp) env)))
    (else (error "Unknown expression type -- EVAL" exp))))
```

```
(define (cond? exp) (tagged-list? exp 'cond))

(define (cond->if exp)
  (cond-clauses->if (cond-clauses exp)))

(define (cond-clauses->if clauses)
  (if (null? clauses)
      #f
      (list 'if (first (car clauses))
            (second (car clauses))
            (cond-clauses->if (cdr clauses)))))
```

example: UNTIL, direct

Suppose that we want to add a new kind of expression called `until`. Its syntax is as follows:

```
(until test exp1 exp2 ... expn)
```

and its behavior is to first evaluate the `test` expression. If the value is `true`, the `until` expression returns the symbol `done`. Otherwise it evaluates each of the expressions `exp1 exp2 ... expn` in turn, then repeats this entire process. (Note: The problem in the online tutor returns `#t` instead of `'done`.)

Assume that we have the following abstractions:

```
(define (until? exp) (tagged-list? exp 'until))  
(define (until-test exp) (cadr exp))  
(define (until-body exp) (caddr exp))
```

example: `(until (= x 0) (set! x (- x 1))
 (print x))`

example: UNTIL, direct

Suppose that we want to add a new kind of expression called `until`. Its syntax is as follows:

```
(until test exp1 exp2 ... expn)
```

and its behavior is to first evaluate the `test` expression. If the value is `true`, the `until` expression returns the symbol `done`. Otherwise it evaluates each of the expressions `exp1 exp2 ... expn` in turn, then repeats this entire process. (Note: The problem in the online tutor returns `#t` instead of `'done`.)

Assume that we have the following abstractions:

```
(define (until? exp) (tagged-list? exp 'until))  
(define (until-test exp) (cadr exp))  
(define (until-body exp) (caddr exp))
```

a. We add a new expression to the evaluator by means of direct evaluation: we add the following clause to `m-eval` `((until? exp) (eval-until exp env))`

Complete the definition of `eval-until`.

```
(define (eval-until exp env)  
  (if
```

example: UNTIL, direct

b. Add UNTIL to the evaluator by using a syntactic transformation: Write a definition for until->if.

e.g. `(until (= x 0) (set! x (- x 1)) (print x))`

\Rightarrow `(if (= x 0) 'done
 (begin (set! x (- x 1)) (print x)
 (until ...)))`

(define (until->if exp)

IF-THEN-ELSE direct

1. IF-THEN-ELSE (from Spring 2006 final)

We are going to add a new special form called **if-then-else** to our meta-circular evaluator. It has the same semantics as our regular Scheme **if**, but the syntax differs in that we explicitly write **then** and **else** in an expression. For example:

```
(if (> x 0) then (decrement x) else (stop))
```

a. We add the following clause to the evaluator: `((if-then-else? exp) (eval-if-then-else exp env))`

Write the procedure `if-then-else?` by completing the following definition. Your answer should ensure that both the **then** and **else** clauses are present in the expression.

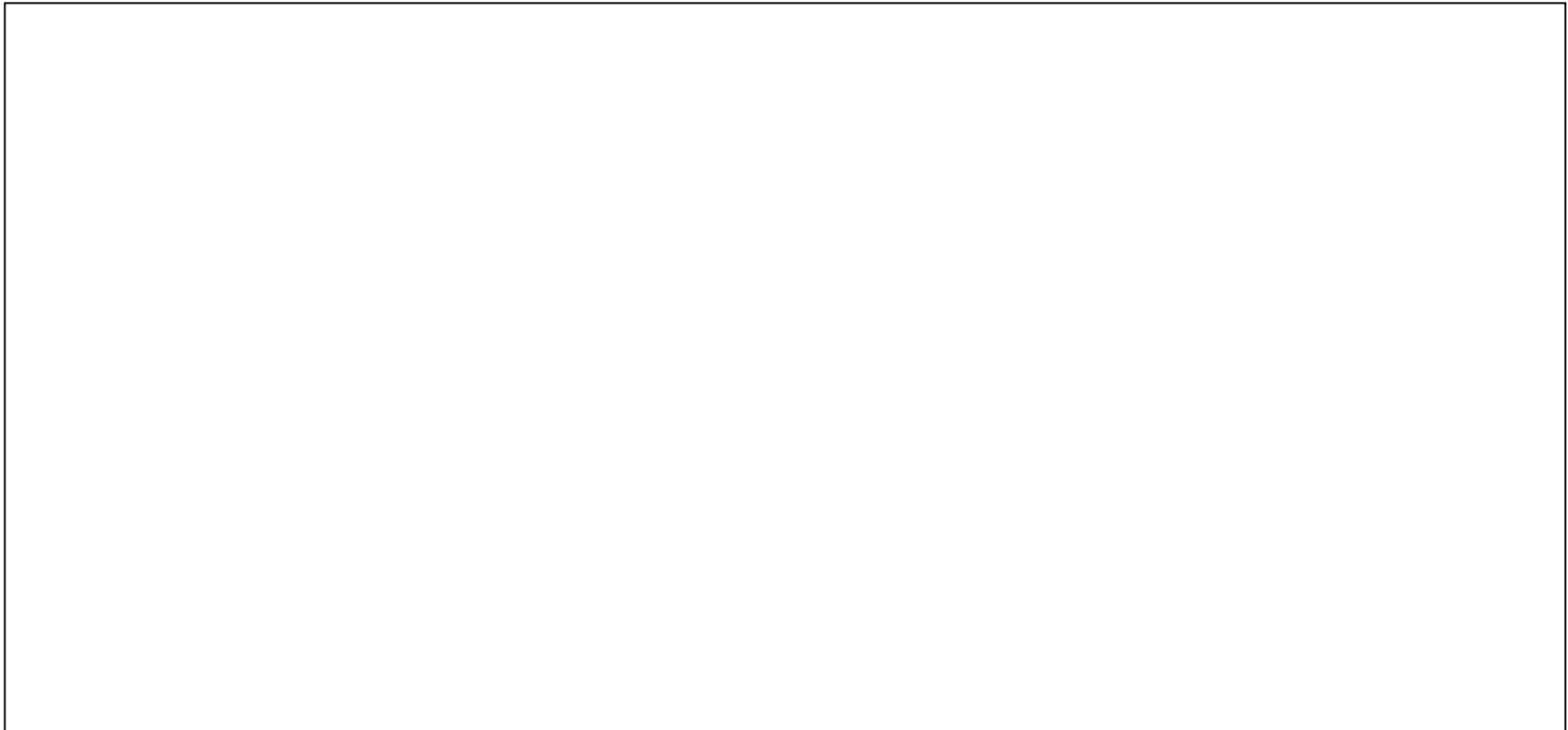
```
(define (if-then-else? exp)
```


IF-THEN-ELSE direct

- b. We add the following clause to the evaluator: `((if-then-else? exp) (eval-if-then-else exp env))`

Write the procedure `eval-if-then-else` by completing the following definition. You may assume that a correct `if-then-else` expression has both labels and clauses for the consequent and alternative.

```
(define (eval-if-then-else exp env)
```



IF-THEN-ELSE: syntactic transformation

- c. We now decide to use syntactic transformation instead of directly adding if-then-else directly to our evaluator. We add the following clause to m-eval:

```
((if-then-else? exp) (m-eval (if-then-else->if exp) env))
```

Write the procedure if-then-else->if. You may assume that a correct if-then-else expression has both labels and clauses for the consequent and alternative.

```
(define (if-then-else->if exp)
```



FOR: syntactic transformation

2. We decide to add a new special form called a `for`, such as `(for i 0 4 (display i) (newline))`. The format of a `for` is as follows: The first expression after the `for` is a variable name; the next two expressions must be integers (note that expressions other than integers are not allowed, e.g. `(+ 2 2)`). The final sequence of expressions within the `for` will be referred to as the body. The above `for` statement is then executed as follows: the body is evaluated with `i` taking values from 0 to 4 inclusive, and the value `'done` is returned. So in this case we would have the behavior:

```
(for i 0 5 (display i) (newline))
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
done
```

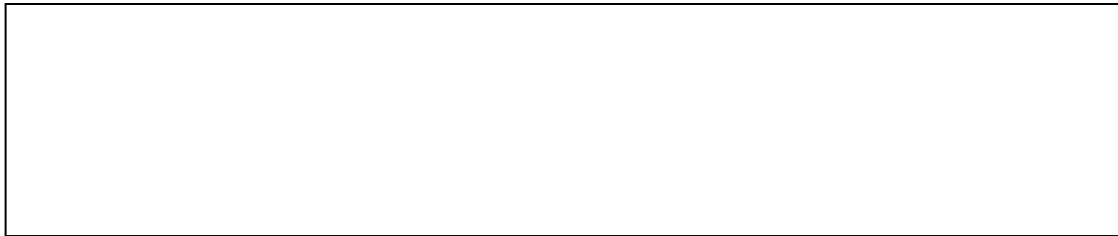
(continued next slide)

FOR: syntactic transformation (cont'd)

2. e.g. (for i 0 5 (display i) (newline))

a. We are going to write for->if, but first we want a procedure that creates a for expression given the parts.

(define (make-for var init end body))



FOR: syntactic transformation (cont'd)

2. e.g. (for i 0 5 (display i) (newline))

Suppose we add this clause to m-eval: ((for? exp) (m-eval (for->if exp) env))

Suppose that we also define:

```
(define (for-tag 'for))
(define (for? exp) (tagged-list? exp for-tag))
(define for-var cadr)
(define for-start-value caddr)
(define for-end-value caddr)
(define for-body cddddr)
```

b. Here is a template for the syntactic transformation. The basic idea is that we are going to create a local frame using a let, in which we bind the loop variable and relative to which we can evaluate the subsequent expressions.

```
(define (for->if exp)
  (list 'let
        ANSWER1
        (list 'if
              ANSWER2
              'done
              ANSWER3)))
```

FOR: syntactic transformation (cont'd)

2. e.g. (for i 0 5 (display i) (newline))

(define for-var cadr)

(define for-start-value caddr)

(define for-end-value caddr)

(define for-body cddddr)

(define (for->if exp)

(list 'let

ANSWER1

(list 'if

ANSWER2

'done

ANSWER3)))

The expression for ANSWER-1 should create an expression that when evaluated will bind the variable to the initial value. Provide the expression.

The expression for ANSWER-2 should create an expression that when evaluated will determine if the for should be exited.

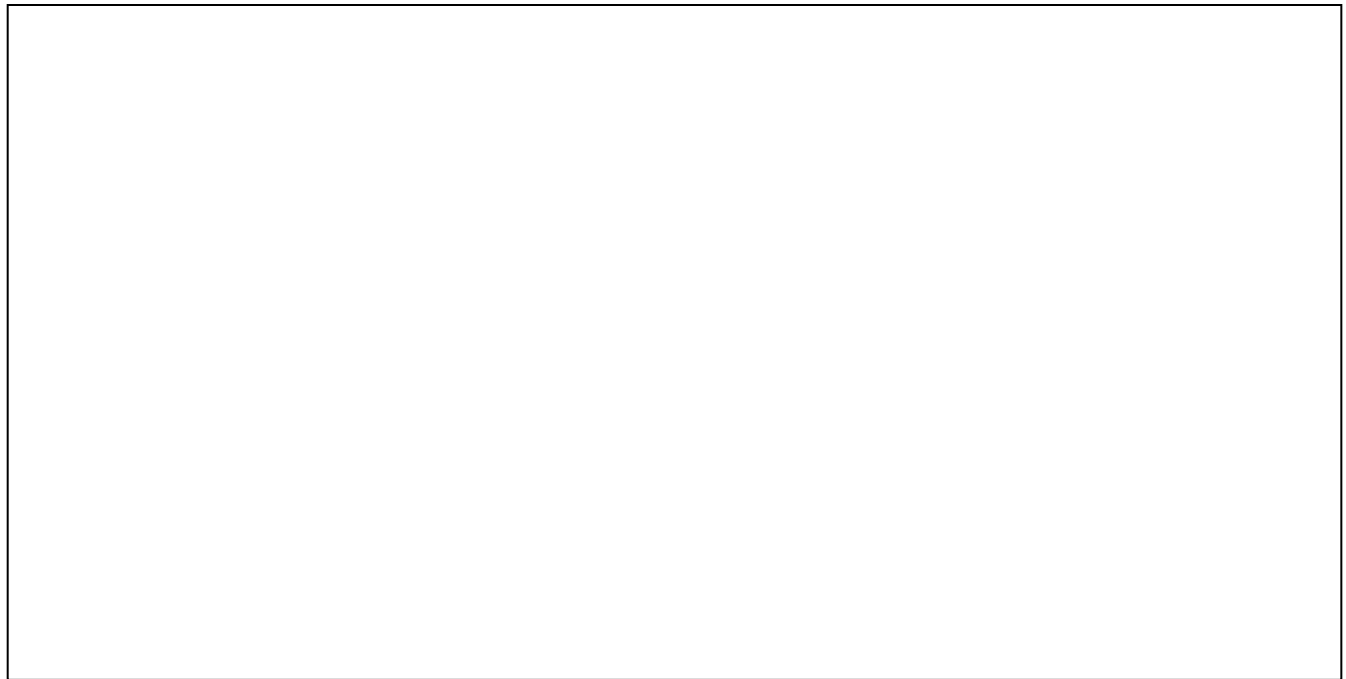
FOR: syntactic transformation (cont'd)

2. e.g. (for i 0 5 (display i) (newline))

```
(define for-var cadr)
(define for-start-value caddr)
(define for-end-value caddr)
(define for-body cddddr)
```

```
(define (for->if exp)
  (list 'let
        ANSWER-1
        (list 'if
              ANSWER-2
              'done
              ANSWER-3)))
```

The expression for ANSWER-3 should create an expression that when evaluated will evaluate the body of the for expression then evaluate a new for expression in an iterative fashion. (Use make-for when appropriate.)



LOOP

3. LOOP

We are going to add a new special form called **loop** to our evaluator. For example:

```
(loop (i 1 inc) (= i 4)
      (newline)
      (display (list i (fact i))))
```

```
(1 1)
(2 2)
(3 6)
; Value: done
```

```
(define start-list '(1 3 5))
```

```
(loop (lst start-list cdr) (null? lst)
      (newline)
      (display (fact (car lst))))
```

```
1
6
120
; Value: done
```

The **syntax** of loop is as follows. The first clause includes a **loop variable** (i in the first example), an expression whose value is the **initial value** of the variable (1 in the first example), and **an increment procedure** to apply to the loop variable on each iteration to create a new value for the loop variable (the value associated with inc in the first example). The next clause is an **end test**, an expression that will evaluate to true or false. The remaining expressions are the **body** of the loop.

The **semantics** of loop is as follows. The loop variable is initially set to the value of its initialization expression. The end test is then evaluated. If the value is true, the loop exits, and the symbol done is returned. If not, the expressions in the body of the loop are evaluated. The increment procedure is then applied to the loop variable, and that variable is bound to the returned value. The process then repeats.

LOOP

3. LOOP (cont'd)

Each of the following procedures extracts elements of a loop. Complete the definitions (assume that each would be applied to a full loop expression).

Question 6:

(define (loop-variable exp) YOUR-ANSWER)

Question 7:

(define (loop-initial-value exp) YOUR-ANSWER)

Question 8:

(define (loop-increment exp) YOUR-ANSWER)

Question 9:

(define (loop-end-test exp) YOUR-ANSWER)

Question 10:

(define (loop-body exp) YOUR-ANSWER)

LOOP

3. LOOP (cont'd)

To implement the special form, we add a dispatch to `m-eval`, and create a new evaluation procedure:

```
(define (m-eval exp env)
  (cond ...
    ((loop? exp) (eval-loop exp env))
    ...
    (application? exp) ...)
  (else ....)))

(define (eval-loop exp env)
  (eval-loop-doit (loop-variable exp)
                 (loop-initial-value exp)
                 (loop-increment exp)
                 (loop-end-test exp)
                 (loop-body exp)
                 env))

(define (eval-loop-doit var init next end bod env)
  (let ((new-env (extend-environment
                 ANSWER-11
                 ANSWER-12
                 env)))
    (if ANSWER-13 ; test to see if done
        ANSWER-14 ; value to return
        (begin ANSWER-15 ; evaluate body
                ANSWER-16)))) ; go to next iteration
```

Question 11: Provide an expression for ANSWER-11. (Together with Question 12, this should create a new environment with the loop variable bound to a new value.)

Question 12: Provide an expression for ANSWER-12.

Question 13: Provide an expression for ANSWER-13 to determine if the loop has satisfied the end condition.

Question 14: Provide an expression for ANSWER-14 to return the correct value from the loop.

Question 15: Provide an expression for ANSWER-15 to evaluate the body of the loop.

Question 16: Provide an expression for ANSWER-16 to handle the next loop iteration.