CS 61A Lab 12 Lab Solutions

3. Why doesn't make-procedure call eval?

Because none of the arguments to lambda should be evaluated.
In particular, the expressions that make up the body of the procedure are
not evaluated until the procedure is *invoked*!

4.1, left-to-right

```
(define (list-of-values exps env)        ;; left to right
  (if (no-operands? exps)
          '()
          (let ((left (eval (first-operand exps) env)))
          (cons left (list-of-values (rest-operands exps) env)))))

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

4.2, Louis reordering

(a) The trouble is that APPLICATION? cheats. The book has

```
(define (application? exp) (pair? exp))
```

It really should be something like

```
(define (application? exp)
  (and (pair? exp)
          (not (member (car exp) '(quote set! define if lambda begin cond)))))
```

They get away with the shorter version precisely because EVAL doesn't
call APPLICATION? until after it's checked for all the possible special
forms. Louis (quite reasonably, I think) wants to rely on APPLICATION?
behaving correctly no matter when it's called.

(b) All we are changing is the syntax of an application, so we
change the procedures that define the "application" abstract data type.
These are on page 372 of the text. The new versions are:

```
(define (application? exp)
  (tagged-list? exp 'call))

(define (operator exp) (cadr exp))

(define (operands exp) (cddr exp))
```

4.4 AND and OR special forms

The book suggests two solutions: make them primitive special forms or make them derived expressions. We'll do both.

As primitive special forms:

Change the COND clause in EVAL by adding

```
  (cond ...
        ((and? exp) (eval-and exp env))
        ((or? exp) (eval-or exp env))
        ...)

(define (eval-and exp env)
  (define (iter tests)
        (cond ((null? tests) #t)
        ((null? (cdr tests)) (eval (car tests) env))
        ((true? (eval (car tests) env)) (iter (cdr tests)))
        (else #f)))
  (iter (cdr exp)))

(define (eval-or exp env)
  (define (iter tests)
        (if (null? tests)
        #f
        (let ((result (eval (car tests) env)))
        (if (true? result)
        result
        (iter (cdr tests))))))
  (iter (cdr exp)))
```

Now for the derived expression technique. Modify the COND clause in EVAL this way instead:

```
  (cond ...
        ((and? exp) (eval (and->if (cdr exp)) env))
        ((or? exp) (eval (or->if (cdr exp)) env))
        ...)

(define (and->if exps)
  (cond ((null? exps) #t)
        ((null? (cdr exps)) (car exps))
        (else (make-if (car exps)
                       (and->if (cdr exps))
                       #f))))

(define (or->if exps)
  (if (null? exps)
      #f
      (make-if (car exps)
      (car exps)
      (or->if (cdr exps)))))
```

This version is elegant but has the disadvantage that you end up computing the first true value twice.

4.5 Cond => notation

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false
      (let ((first (car clauses))
      (rest (cdr clauses)))
      (if (cond-else-clause? first)
      (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "..."))
      (IF (COND-ARROW-CLAUSE? FIRST)
                (LIST (MAKE-LAMBDA '(COND-FOO)
                          (MAKE-IF 'COND-FOO
                                   (LIST (COND-ARROW-DOER FIRST)
                                   'COND-FOO)
                                   (EXPAND-CLAUSES REST)))
                (COND-PREDICATE FIRST))
                (make-if (cond-predicate first)
                (sequence->exp (cond-actions first))
                (expand-clauses rest)))))))
```

```
(define (cond-arrow-clause? clause)
  (and (pair? clause)
       (= (length clause) 3)
       (eq? (cadr clause) '=>)))

(define (cond-arrow-doer clause)
  (caddr clause))
```

This may be a little confusing. What it does is to turn a clause like

```
(test => recipient)
```

into

```
((lambda (cond-foo)
(if cond-foo
(recipient cond-foo)
<expanded-rest-of-clauses>))
test)
```

Using the name cond-foo here is a kludge, because what if the user has used the same name for some other purpose within the clause? The right thing would be to generate an otherwise-untypable symbol each time. But this is complicated enough already.

By the way, this is really trying to do

```
(let ((cond-foo test))
(if ...))
```

but we don't yet have LET in the metacircular Scheme.

It might be easier to do this by abandoning the whole idea of cond->if and just implementing cond directly.

5b. In Logo there are no internal definitions; all procedures are global. So we need a situation with two procedures, one of which calls the other:

```
to outer :var
inner
end
```

```
to inner
print :var
end

? outer 23
23
```

To see that this result is different from what would happen with lexical scope, try the same example in Scheme:

```
(define (outer var)
  (inner))

(define (inner)
  (print var))

> (outer 23)
Error -- unbound variable: var
```

(Or you could define a global variable var whose value is something other than 23, and then (outer 23) would print that other value.

5c.

Logo " is like Scheme ' -- it's the quoting symbol. But in Logo it is used only with words, not with lists, and there is no QUOTE special form which the quotation character abbreviates.

Logo [ ] are like '( ) in Scheme -- the brackets both delimit and quote a list. But within a list, brackets are used to delimit sublists, and don't imply an extra level of quotation, so Logo [a [b c] d] means '(a (b c) d), not '(a '(b c) d). So, how do you get the effect of Scheme's ( ) without quotation? In Scheme that means to call a procedure; in Logo you don't need any punctuation to call a procedure! You just give the procedure name and its arguments. But in Logo you *can* use parentheses around a procedure call just as you would in Scheme.

Logo : means that you want the value of the variable whose name follows the colon. In Scheme the name by itself means this -- if you want the value of variable X, you just say X. The reason this doesn't work in Logo is that in Logo procedures aren't just another data type, and a procedure name isn't just the name of a variable whose value happens to be a procedure. (In other words, Logo procedures are not first-class.) In Logo there can be a procedure

and a variable with the same name, so FOO means the procedure and :FOO (pronounced "dots FOO") means the variable.