CS 61A Lab 6 solutions

LAB ASSIGNMENT:

1. Scheme-1 stuff.

(a) ((lambda (x) (+ x 3)) 5)

Here's how Scheme-1 handles procedure calls (this is a COND clause inside EVAL-1):

```
((pair? exp) (apply-1 (eval-1 (car exp))        ; eval the operator
                      (map eval-1 (cdr exp))))  ; eval the args
```

The expression we're given is a procedure call, in which the procedure (lambda (x) (+ x 3)) is called with the argument 5.

So the COND clause ends up, in effect, doing this:

```
(apply-1 (eval-1 '(lambda (x) (+ x 3))) (map eval-1 '(5)))
```

Both lambda expressions and numbers are self-evaluating in Scheme-1, so after the calls to EVAL-1, we are effectively saying

```
(apply-1 '(lambda (x) (+ x 3)) '(5))
```

APPLY-1 will substitute 5 for X in the body of the lambda, giving the expression (+ 5 3), and calls EVAL-1 with that expression as argument. This, too, is a procedure call. EVAL-1 calls itself recursively to evaluate the symbol + and the numbers 5 and 3. The numbers are self-evaluating; EVAL-1 evaluates symbols by using STk's EVAL, so it gets the primitive addition procedure. Then it calls APPLY-1 with that procedure and the list (5 3) as its arguments. APPLY-1 recognizes that the addition procedure is primitive, so it calls STk's APPLY, which does the actual addition.

(b) As another example, here's FILTER:

```
((lambda (f seq)
  ((lambda (filter) (filter filter pred seq))
       (lambda (filter pred seq)
       (if (null? seq)
```

```
        '()
        (if (pred (car seq))
        (cons (car seq) (filter filter pred (cdr seq)))
        (filter filter pred (cdr seq)))))))
 even?
 '(5 77 86 42 9 15 8))
```

(c) Why doesn't STk's map work in Scheme-1? It works for primitives:

```
        Scheme-1: (map first '(the rain in spain))
        (t r i s)
```

but not for lambda-defined procedures:

```
        Scheme-1: (map (lambda (x) (first x)) '(the rain in spain))
        Error: bad procedure: (lambda (x) (first x))
```

This problem illustrates the complexity of having two Scheme interpreters coexisting, STk and Scheme-1. In Scheme-1, lambda expressions are self-evaluating:

```
        Scheme-1: (lambda (x) (first x))
        (lambda (x) (first x))
```

But in STk, lambda expressions evaluate to procedures, which are a different kind of thing:

```
        STk> (lambda (x) (first x))
        #[closure arglist=(x) 40179938]
```

STk's MAP function requires an *STk* procedure as its argument, not a Scheme-1 procedure! Scheme-1 uses STk's primitives as its primitives, so MAP is happy with them. But a Scheme-1 lambda-defined procedure just isn't the same thing as an STk lambda-defined procedure.

d) Scheme-1 AND form.

Special forms are handled by clauses in the COND inside EVAL-1, so we start by adding one for this new form:

```
(define (eval-1 exp)
  (cond ((constant? exp) exp)
```

```
        ((symbol? exp) (error "Free variable: " exp))
        ((quote-exp? exp) (cadr exp))
        ((if-exp? exp)
        (if (eval-1 (cadr exp))
        (eval-1 (caddr exp))
        (eval-1 (cadddr exp))))
        ((lambda-exp? exp) exp)
        ((AND-EXP? EXP) (EVAL-AND (CDR EXP)))          ;; added
        ((pair? exp) (apply-1 (car exp)
                              (map eval-1 (cdr exp))))
        (else (error "bad expr: " exp))))
```

Note that the new clause has to come before the PAIR? test, because special forms are also pairs, and must be caught before we try to interpret them as ordinary procedure calls.

We also need the helper that checks for a list starting with the word AND:

```
(define and-exp? (exp-checker 'and))
```

That was the easy part. Now we have to do the actual work, in the procedure EVAL-AND. I chose to give it (CDR EXP) as its argument because I'm envisioning a recursive loop through the subexpressions, and we want to leave out the word AND itself, which isn't to be evaluated.

What AND is supposed to do is to go through the subexpressions from left to right, evaluating each in turn until either some expression's value is #F (in which case we return #F) or we run out (in which case we return, to get exactly Scheme's behavior, the value of the last expression, which might be some true value other than #T).

```
(define (eval-and subexps)
  (if (null? subexps)                    ; Trivial case: (AND)
        #T                               ;  returns #T
        (let ((result (eval-1 (car subexps))))   ; else eval first one.
        (cond ((null? (cdr subexps)) result)    ; Last one, return its value.
        ((equal? result #F) #F)          ; False, end early.
        (else (eval-and (cdr subexps)))))))  ; else do the next one.
```

The LET here is used so that there is only one recursive call to EVAL-1, but the program can be written without it, and turns out only to call EVAL-1 once anyway, even though the call appears in two different places in the code, because only one of them will be carried out (per invocation of EVAL-AND, of course).

```
(define (eval-and subexps)
  (cond ((null? subexps) #T)
        ((null? (cdr subexps)) (eval-1 (car subexps)))
        ((equal? (eval-1 (car subexps)) #F) #F)
        (else (eval-and (cdr subexps)))))
```

Note that the first NULL? test is not really a base case; unless the entire expression given to us was exactly (AND), the second NULL? test will always become true before the first one does. It's that second one that's the base case.

(If we wanted AND always to return either #T or #F, rather than return the value of the last expression, then we'd leave out the second NULL? test, and the first one *would* be the base case of the recursion.)

2.62. Union-set in Theta(n) time.

The key is to realize the differences between union-set and intersection-set. The null case for union-set will be different, because if one of the sets is empty, the result must be the other set. In the element comparisons, one element will always be added to the result set. So the expressions with the element will be the same as intersection-set, only with a cons added. Here's the solution:

```
(define (union-set set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else (let ((x1 (car set1)) (x2 (car set2)))
                (cond ((= x1 x2)
                (cons x1 (union-set (cdr set1) (cdr set2))))
                ((< x1 x2)
                (cons x1 (union-set (cdr set1) set2)))
                ((< x2 x1)
                (cons x2 (union-set set1 (cdr set2)))))))))
```

Trees on page 156:

```
(define tree1
  (adjoin-set 1
   (adjoin-set 5
```

```
        (adjoin-set 11
        (adjoin-set 3
        (adjoin-set 9
        (adjoin-set 7 '()))))))))

(define tree2
  (adjoin-set 11
   (adjoin-set 5
        (adjoin-set 9
        (adjoin-set 1
        (adjoin-set 7
        (adjoin-set 3 '()))))))))

(define tree3
  (adjoin-set 1
   (adjoin-set 7
        (adjoin-set 11
        (adjoin-set 3
        (adjoin-set 9
        (adjoin-set 5 '()))))))))
```

Other orders are possible; the constraint is that each node must be added before any node below it. So in each case we first adjoin the root node, then adjoin the children of the root, etc. To make sure this is clear, here's an alternative way to create tree1:

```
(define tree1
  (adjoin-set 11
   (adjoin-set 9
        (adjoin-set 5
        (adjoin-set 1
        (adjoin-set 3
        (adjoin-set 7 '()))))))))
```