Review Questions
1. Remember that **and** in Scheme is a kind of expression. Write a *procedure* `myAnd` that takes any number of arguments and returns `#t` if all of those arguments evaluate to `#t`. 
2. Remember apply-proc in our Minisheme interpreter. This took a procedure and zero or more literal arguments (such as numbers; not parse trees) and returned the result of applying the procedure to the arguments. Here is my code for this procedure:

```
(define apply-proc (lambda (p args)
    (cond
      [(prim-proc? p)
        (apply-prim-proc p args)]
      [(closure? p)
        (eval-exp (Body p)
          (extended-env Params(p)
            (map box args)
            (Env p))))]))
```

How would this procedure change if we used dynamic binding rather than static binding?
3. Use foldl or foldr to write alternating-sum, a procedure that takes vector (a b c ... e) and produces
   a-b+c-d+e

Use foldl or foldr to write (rember-all a lat)

Use foldl or foldr to write (count a lat)

Or to write (index a lat)
4. Here is a binary tree definition.  
   (define new-tree (lambda (value leftChild rightChild)  
       (list 'tree value leftChild rightChild))  
   You can make up getters for the three fields.

Write a procedure that returns a list of the values stored in the tree in a pre-order traversal (root, then everything in its left-most subtree, etc.) For example, with this tree:

```
   5
  / \  /
 6   2
 / \ / \  
1 3 4 7
```

you should return (5 6 1 3 7 2 4)
Write procedure (SameElts lat1 lat2) that returns #t if lat1 and lat2 have the same elements in the same multiplicities but not necessarily the same order.
7. Give a CPS version of (rember a lat). Remember that (rember a lat) removes the first instance of a from lat.
8. Give a Scheme expression that creates the stream Power$ that has powers of 2 and powers of 3, in increasing numerical order starting with 1. If you use print$ on your stream you should get the values (1, 2, 3, 4, 8, 9, 16, 27, 32...)
9. Here are some practice problems for Continuation-Passing Style:

A. Give a tail-recursive continuation-passing-style function \((\text{rember-k a lat k})\) that removes the first occurrence (only the first) of atom \(a\) from \(\text{lat}\) and then applies \(k\) to the result. So

\[
(\text{rember-k 'b '(a b a b a b b) (lambda (x) x)} )
\]

returns '\(a a b a b b)'.

B. Give a tail-recursive continuation-passing style function \((\text{index-k a lat k})\) that returns the 0-based index of the first occurrence of atom \(a\) in \(\text{lat}\). So

\[
(\text{index-k 'b '(a b a b b)top})
\]

returns 1.

C. Give a tail-recursive continuation-passing-style function \((\text{max-k L k})\) that returns the largest element of the not-necessarily-flat list \(L\) of numbers. For example,

\[
(\text{max-k '(5 3 (4 7 2 (5) 1)) top})
\]

returns 7.

D. Give a tail-recursive continuation-passing style function \((\text{replace-k old new L k})\) that replaces each instance of atom \(\text{old}\) with atom \(\text{new}\) in the general list \(L\). For example,

\[
(\text{replace-k 'a 'x '(a b c (b c (a))) (lambda (x) x)} )
\]

produces \((x b c (b c (x)))\).