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)
            (evl-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 \((a \ b \ c \ \ldots \ \ e)\) and produces \(a-b+c-d+e\)

Use foldl or foldr to write \(\text{(rember-all a lat)}\)

Use foldl or foldr to write \(\text{(count a lat)}\)

Or to write \(\text{(index a la)}\)
4. Here is a 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:

![Tree Diagram]

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 APS and CPS versions of (rember a lat). Call your procedures rember-acc and rember-k. Remember that (rember a lat) removes the first instance of a from lat.
8. Give a Scheme expression that creates the stream \text{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. What does this function return if you evaluate

```
(define f (lambda (vec)
    (call/cc (lambda (k)
        (cond
            [(null? vec) 0]
            [(eq? -1 (car vec)) (k 1)]
            [else (* (car vec) (f (cdr vec)))]))))
```

```(f '(3 2 1 -1 4 3 2 1 0))```