

Exercise 2.35.

Redefine `count-leaves` from section 2.2.2 as an accumulation:

```
(define (count-leaves t)
  (accumulate <??> <??> (map <??> <??>)))
```

Answer.

The `count-leaves` procedure we saw in section 2.2.2 computed the number of leaves by tree-recursion. This is a natural way because the recursive plan is in accord with the shape of the data structure—tree. However, expressing this strategy directly with accumulation will cause great trouble, for the conventional interface used in the process of accumulation is sequence rather than tree.

To redefine `count-leaves` using `accumulate`, one would consider flatten the tree into a sequence of leaves, and then count the number of elements within that sequence

```
(define (count-leaves t)
  (accumulate (lambda (x y)
                (+ 1 y))
              0
              (enumerate-tree t)))
```


This does not match the problem description above which claimed to use `map`, though it indeed is a nice implementation.

On the other hand, we can perform only a minor adjustment on `count-leaves` to make it meet the needs approximately. Just `map` the sequence to itself

```
(define (count-leaves t)
  (accumulate (lambda (x y)
                (+ 1 y))
              0
              (map (lambda (x) x)
                    (enumerate-tree t))))
```

So far, the implementation seems pretty good, but it still can be optimized. Think about this questions: Isn't it a waste of efforts to count the leaves by `mapping` a sequence to itself and then counting the elements one by one? Well, we can obtain the number of leaves by `mapping` the list into a sequence of 1s and then just get their sum:

```
(define (count-leaves t)
  (accumulate +
              0
              (map (lambda (x) 1)
                    (enumerate-tree t))))
```

*. Creative Commons  2013, Lawrence X. Amlord (颜世敏, aka 颜序).
Email address: informlarry@gmail.com