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. Natural is this way because the recursive plan is in accord with the shape of the data structure—tree.

Expressing this strategy directly with accumulation will cause great trouble, for what conventional interface used in the process of accumulation is sequence rather than tree.

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

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

This does not fix in with the format given by the problem description which claimed to use `map`, though it indeed is a good implementation.

But we can perform only a minor adjustment on `count-leaves` to make it meet the needs. Just `map` the sequence to itself

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

However, this is not the end. Doesn't it a waste of efforts to count the leaves by `map`ing a sequence to itself and then counting the elements one by one? Why not obtain the number of leaves by `map`ing 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))))
```

Obviously, the procedure look more manifest after being improved.