Exercise 2.33.

Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

```
(define (map p sequence)
  (accumulate (lambda (x y) <??>) nil sequence))

(define (append seq1 seq2)
  (accumulate cons <??> <??>))

(define (length sequence)
  (accumulate <??> 0 sequence))
```

Answer.

Remember that in the process of accumulation, `accumulate` composes the result we desired element-by-element from a given list. How these elements are processed and assembled is specified by the parameter `op`.

We saw in section 2.2.1 that the procedure `map` takes as its arguments a procedure of one argument and a list, and returns a list of results produced by applying the procedure to each element in the list. This was done by applying the procedure element-by-element using `cons`.

By introducing `accumulate`, the process of element-by-element procedure application can be taken over by `op`. And we can manage to do so specifying `op` to be a procedure which successively `cons`ing up the processed elements:

```
(define (map p sequence)
  (accumulate (lambda (x y)
                (cons (p x) y))
              nil
              sequence))
```

where the parameters `x` and `y` will be substituted by the first elements of a list and the accumulation of the rest elements respectively.

To give the definition of `append` in this way, recall the recursive plan we took in implementing `append` in section 2.2.1:

- If `seq1` is the empty list, then the result is just `seq2`.
- Otherwise, `append` the `cdr` of `seq1` and `seq2`, and `cons` the `car` of `seq1` onto the result.

This indicates that in the body of `accumulate`, `cons` acts as the operator `op`, `seq2` sets the `initial` value of the composition and `seq1` therefore takes the place of `sequence`

```
(define (append seq1 seq2)
  (accumulate cons seq2 seq1))
```

As is introduced in section 2.2.1, we get the `length` of a list with a recursive plan:

- The `length` of the empty list is 0.
- The `length` of any list is 1 plus the `length` of the `cdr` of the list.

As far as `accumulate` concerns, the reduction step can be captured by an operation which adds `1` to the accumulation of the rest of the sequence. This reveals the complete picture of `length` defined by `accumulate`:

```
(define (length sequence)
```


Email address: informlarry@gmail.com

```
(accumulate (lambda (x y) (+ 1 y))
            0
            sequence))
```