Exercise 2.36.

The procedure `accumulate-n` is similar to `accumulate` except that it takes as its third argument a sequence of sequences, which are all assumed to have the same number of elements. It applies the designated accumulation procedure to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if `s` is a sequence containing four sequences, `((1 2 3) (4 5 6) (7 8 9) (10 11 12))`, then the value of `(accumulate-n + 0 s)` should be the sequence `(22 26 30)`. Fill in the missing expressions in the following definition of `accumulate-n`:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init <??>)
            (accumulate-n op init <??>))))
```

Answer.

We have seen in the problem description that `accumulate-n` combines elements in the same order of the sequences to produce a sequence of the results. It is natural to express this strategy in a recursive way:

- If the first subsequence is the empty list, then the result is just `nil`
- Otherwise, `Accumulate` all the first elements of the sequences, and `cons` the result onto the `accumulate-n` of all the subsequent elements of the sequences:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init (car-n seqs))
            (accumulate-n op init (cdr-n seqs)))))
```

The auxiliary procedures `car-n` withdraws all the first elements of the sequences and arranges them in a list in the same order:

```
(define (car-n seqs)
  (if (null? seqs)
      nil
      (cons (car (car seqs))
            (car-n (cdr seqs)))))
```

And after that, `cdr-n` produces the reduced sequences:

```
(define (cdr-n seqs)
  (if (null? seqs)
      nil
      (cons (cdr (car seqs))
            (cdr-n (cdr seqs)))))
```

However, it is neither the only nor the best way to extract this problem. Taking a close look at `car-n` and `cdr-n`, we find that they both attain a sequence from the known one with the same number of elements. More over, all the elements in the new sequence are gained in the same way. This pattern reflects the idea of mapping we saw in section 2.2.1. Therefore, we can use `map` to obtain those two sequences and express `accumulate-n` in a more elegant way:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init (map car seqs))
            (accumulate-n op init (map cdr seqs)))))
```