Exercise 2.39.

Complete the following definitions of `reverse` (exercise 2.18) in terms of fold-right and fold-left from exercise 2.38:

```
(define (reverse sequence)
  (fold-right (lambda (x y) <??>) nil sequence))

(define (reverse sequence)
  (fold-left (lambda (x y) <??>) nil sequence))
```

Answer.

Recall that `fold-right` combines the first element of the sequence with the result of combining all the elements to the right. Going this way, we can reverse a sequence by interchanging the `car` of the sequence with the reverse of the `cdr` of the sequence. And this can be done by `append`ing the reversion of the `cdr` of the sequence by a list which contains only the `car` of the sequence:[1]

```
(define (reverse sequence)
  (fold-right (lambda (x y)
                (append y (list x)))
              nil
              sequence))
```

On the other hand, reversion of a sequence can also be done by left-folding which initialize an empty sequence first, then picks up the head element from the original sequence and insert it in the front of the new one. And notice that edge-insertion is in fact a sort of `append`ing, this reveals another implementation of `reverse`:

```
(define (reverse sequence)
  (fold-left (lambda (x y)
               (append (list y) x))
             nil
             sequence))
```

1. Someone might come up with an idea of `cons`ing the reversion of the `cdr` of the sequence onto the `car` of the sequence `(e1 e2 e3 ... en)`. Unfortunately, this strategy fails by producing the reversed sequence in a strange nested way:

```
(cons (cons (cons (... (cons nil en) ...)
                  e3)
            e2)
      e1)
```

Another hasty practice would be by simply `list`ing the reversion of the `cdr` of the sequence with the `car` of it. Similarily, trying this plan also generates a nested sequence though the elements are in a reversed order:

```
(list (list (list (... (list nil en) ...)
                  e3)
            e2)
      e1)
```