Exercise 1.32.

a. Show that **sum** and **product** (exercise 1.31) are both special cases of a still more general notion called **accumulate** that combines a collection of terms, using some general accumulation function:

```
(accumulate combiner null-value term a next b)
```

Accumulate takes as arguments the same term and range specifications as **sum** and **product**, together with a **combiner** procedure (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a **null-value** that specifies what base value to use when the terms run out. Write **accumulate** and show how **sum** and **product** can both be defined as simple calls to **accumulate**.

b. If your **accumulate** procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

Answer.

a. Well, here we try to come up with a recursive version of **accumulate** first, the iterative version will be shown in part b. of this answer. Now, let's go!

Let's first recall the general routine we should follow to design a recursive procedure:

i. Wishful Thinking: Assume the desired procedure exists, but only for versions of the problem smaller than the current one

ii. Decompose the Problem: Combine the solution to a smaller version, plus a set of simple operations, to create the solution to the full version of the problem.

iii. Identify Non-decomposable Problem: Find out the smallest sized problem that can be solved directly, without using wishful thinking.

Inspired by the former recursive versions of **sum** and **product**, one might easily find out that the simplest case here lies in the case when $a > b$, where the result of the **accumulate** expression should be **null-value**. Thus, we may easily shape the prototype of **accumulate** in out mind:

```
(define (accumulate combiner null-value term a next b)
  (if (> a b)
      null-value   ;; a direct answer to the simplest case
      (<a decomposed version of the problem>)))
```

By now, the problem has been reduced into how to decompose the problem into a still simpler version, plus a set of simple operations. On the other hand, to accumulate a collection of terms, a general practice would be picking the current term and combining it with the rest of the collection. Thus, the simpler version here appeals to be accumulating the rest of the collection in terms of current term, and that simple operation indicates to address the current term. Hence, this reveals the true features of *alternative* in the if conditional above:

```
(combine (term a)
         (accumulate combiner null-value term (next a) next b))
```

Therefore, our **accumulate** can be express as:

```
(define (accumulate combiner null-value term a next b)
  (if (> a b)
      null-value
      (combiner (term a)
                (accumulate combiner null-value term (next a) next b))))
```

Using this procedure, we can redefined the **sum** and **product** in a pretty easy way:

```
(define (sum term a next b)
  (accumulate + 0 term a next b))

(define (product term a next b)
  (accumulate * 1 term a next b))
```

b. Generally speaking, to accumulate a collection of terms iteratively, we would combine the terms one by one from the begining to end. Although a bounch of variables keep changing thoughout the computation, what we need to record the current status are only 3 variables: the current index, the current accumulation and the upper bound. Therefore, our iterative version of **accumulate** appeals to be:

```scheme
(define (accumulate combiner null-value term a next b)
  (define (accum-iter index accum upper)
    (if (> index upper)
        accum
        (accum-iter (next index)
                    (combiner (term index) accum)
                    upper)))
  (accum-iter a null-value b))
```