Exercise 1.31.

a. The `sum` procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-order procedures.[1] Write an analogous procedure called `product` that returns the product of the values of a function at points over a given range. Show how to define `factorial` in terms of product. Also use `product` to compute approximations to π using the formula[2]

$$\frac{\pi}{4} = \frac{2\cdot 4\cdot 4\cdot 6\cdot 6\cdot 8\cdots}{3\cdot 3\cdot 5\cdot 5\cdot 7\cdot 7\cdots}$$

b. If your `product` 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. Much similar to the `sum` procedure we've seen before, the procedure of `product` here follows a abstraction of *product of a series*. Thus, we can obtain the procedure `product` by performing only a minor adjustment on `sum`. Both the recursive and iterative version are okay, but we discuss on the recursive version first.

However, in order to complete our procedure elegantly, we have to tackle two problems: What is the boundary of our test? How does the evaluator deal with it? Well, inspired by the `sum` procedure, we know that the evaluation should terminate whenever a is greater than b. Further more, the evaluator should return 1 in this case.

```
(define (product term a next b)
  (if (> a b)
      1
      (* (term a)
         (product term (next a) next b))))
```

Therefore, we can immediately come up with the implementation for procedure `factorial`:

```
(define (factorial n)
  (product identity 1 inc n))
```

Before computing approximations to π, let's first analyze those patterns submerged in the right part of the equation:

$$\frac{2\cdot 4\cdot 4\cdot 6\cdot 6\cdot 8\cdots}{3\cdot 3\cdot 5\cdot 5\cdot 7\cdot 7\cdots}$$

Note that the numerator here can evolve into:

$$2\cdot(4\cdot 6)^2\cdot 8$$

Similarly, we can view the denominator as:

$$(3\cdot 5\cdot 7\cdot 9\cdots)^2$$

Hence, both the numerator and the denominator can be easily expressed in terms of the procedure `product`. This eventually leads to the definition of the right part of the formula:

```
(define (pi-quotient a b)
  (define (pi-next x) (+ x 2))
  (define numer
    (product square (+ a 2) pi-next (- b 2)))
  (define denom
    (product square (+ a 1) pi-next (- b 1)))
  (/ (* a numer b) denom))
```

Using this procedure, we can compute an approximation to π:

1. The intent of exercises 1.31–1.33 is to demonstrate the expressive power that is attained by using an appropriate abstraction to consolidate many seemingly disparate operations. However, though accumulation and filtering are elegant ideas, our hands are somewhat tied in using them at this point since we do not yet have data structures to provide suitable means of combination for these abstractions. We will return to these ideas in section 2.2.3 when we show how to use *sequences* as interfaces for combining filters and accumulators to build even more powerful abstractions. We will see there how these methods really come into their own as a powerful and elegant approach to designing programs.

2. This formula was discovered by the seventeenth-century English mathematician John Wallis.

```scheme
(* 4 (pi-quotient 2 1000))
```

b. Inspired by exercise 1.30, we can easily come up with an iterative version for procedure `product`:

```scheme
(define (product term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (* (term a) result))))
  (iter a 1))
```