Exercise 1.37.

a. An infinite *continued fraction* is an expression of the form

$$f = \frac{N_1}{D_1 + \frac{N_2}{D_2 + \frac{N_3}{D_3 + \cdots}}}$$

As an example, one can show that the infinite continued fraction expansion with the N_i and the D_i all equal to 1 produces $1/\phi$, where ϕ is the golden ratio (described in section 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation—a so-called *k-term finite continued fraction*—has the form

$$\frac{N_1}{D_1 + \frac{N_2}{\ddots + \frac{N_k}{D_k}}}$$

Suppose that `n` and `d` are procedures of one argument (the term index i) that return the N_i and D_i of the terms of the continued fraction. Define a procedure `cont-frac` such that evaluating `(cont-frac n d k)` computes the value of the k-term finite continued fraction. Check your procedure by approximating $1/\phi$ using

```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
           k)
```

for successive values of `k`. How large must you make `k` in order to get an approximation that is accurate to 4 decimal places?

b. If your `cont-frac` 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. Before setting out to design such a procedure, we are supposed to find out the patterns submerged in the expression

$$\frac{N_1}{D_1 + \frac{N_2}{\ddots + \frac{N_k}{D_k}}} \qquad (1)$$

An intuitive strategy to evaluate it would be proceeding the computation by two steps:

i. Evaluate N_i and D_i.

ii. Combine N_i and D_i with the rest part of the expression.

where i varies from 0 to k, and the simplest case lies where $i = 0$.

Thinking hastily, one might express this idea quite straightforward in Lisp

```
(define (cont-frac n d k)
  (if (= k 0)
      0
      (/ (n k)
         (+ (d k)
            (cont-frac n d (- k 1))))))
```

and delighted with the output in testing with

```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
           k)
```

However, this delusion ruins when it comes to evaluate with other kinds of `n` and `d` whose return value are not constant to `i`. `(lambda (i) i)` for example, falsifies this naive practice.

The reason that the procedure above fails to express our intension lies in the practice which the guy took to deal with the index. Instead of increasing, the index value in the `cont-frac` procedure above decreases as the evalutor descends into each layer of denominators. Thus, this acts in direct contravention to our original purpose.

To fix the procedure, we'd better change the way in controlling the index. We hope the index starts from 1 and increases one by one through out the computation to reflect on the pattern emerged in formula (1), and keeps the value of `k` constant in the mean time. So we can keep track of the index by using a local variable i starting from 1, and increase by 1 on each step, as is shown in Figure 1. Finally, we hve to identify the non-decomposable

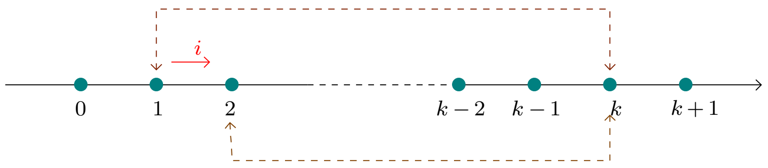

Figure 1. Increasing Process of Index in the Computation

problem. That is, when $i=k$, the fraction $\frac{N_i}{D_i}$ is the result we seek. Therefore, after amelioration, our procedure turns out to be:

```
(define (cont-frac n d k)
  (define (cfr i)
    (if (= i k)
        (/ (n i) (d i))
        (/ (n i)
           (+ (d i) (cfr (+ i 1))))))
  (cfr 1))
```

As the test shows, in order to get an approximation that is accurate to 4 decimal places, the value of `k` should be at least raised up to 11.[1]

b. We can also formulate an iterative process for computing the k-term finite continued fraction. Suppose we are about to compute a 5-term fraction

$$\frac{1}{1+\frac{2}{\ddots+\frac{5}{5}}}$$

As is shown in Figure 2, the value of k starts from 5 and decreases by 1 on each step. Every successive result

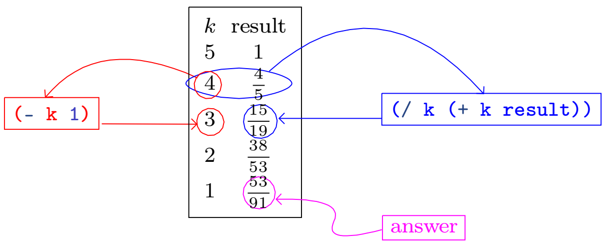

Figure 2. Iterative Process Generated in the Computation

can be obtained through deviding k by the sum of k and the current result

$$\text{Result}_{k-1} = \frac{k}{k+\text{Result}_k}$$

And finally, when k becomes 0, the fraction is therefore the result we seek.

So in order to obtain the value of the $(k-1)$-term fraction, just divide N_{k-1} by the sum of D_{k-1} and $\frac{N_k}{D_k}$

$$\frac{N_{k-1}}{D_{k-1}+\frac{N_k}{D_k}}$$

Note that the value of the denominator accumulates with k counting down on each level of the fraction. Hence, during this process, what the evaluator has to keep track of are the current index k and the result accumulated. Therefore, we can express this idea in Lisp:

1. For the sake of aesthetics, the tedious output generated by the interpreter is not presented here. It is located in file *Test_for_Exercise_1.37.scm*.

```scheme
(define (cont-frac n d k)
  (define (iter k result)
    (if (= k 1)
        result
        (iter (- k 1)
              (/ (n k)
                 (+ (d k) result)))))
  (iter k (/ (n k) (d k))))
```