## Exercise 1.29

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
     (sum term (next a) next b))))
(define (cube x)
  (* x x x))
(define (simpson f a b n)
  (define h (/ (- b a) n))
  ;; method 2 use 0-n sequence
  (define (term x)
    (+ (* 4 (f x) (/ (- b a) (* 3 n)))
       (* 2 (f x) (/ (- b a) (* 3 n)))))
  (define (next x)
    (+ x (* 2 (/ (- b a) n))))
  (sum term (+ a (* (f a) (/ (- b a) (* 3 n)))) next (- b (* 2 h))))
(simpson cube 0 1 1000)
```

## Exercise 1.30

```
(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
    result
    (iter (next a) (+ result (term a)))))
  (iter a 0))
(define (identify x) x)
(define (inc x) (+ x 1))
(sum identify 1 inc 3)
```

## Exercise 1.31

```
(define (product factorial a next b)
  (define (iter a result)
    (if (> a b)
    result
    (iter (next a) (* result (factorial a)))))
  (iter a 1))
;; test ok
;; (define (identify x) x)
;; (define (inc x) (+ x 1))
;; (product identify 1 inc 4)
```

```
(define (next x)
  (+ x 2))
(define (square x)
  (* x x))
(define (factorial x)
  (/ (* x (+ x 2)) (square (+ x 1))))
(* 4 (product factorial 2. next 1024))
;; result = 3.1431
```

## Exercise 1.32

Recursive

```
(define (accumulate combiner null-value term a next b)
  (if (> a b)
      null-value
      (accumulate combiner (combiner (term a) null-value) term (next a)
          next b)))
;; (define (combiner x y) (* x y))
;; (define (next x)
;;   (+ x 2))
;; (define (square x)
;;   (* x x))
;; (define (factorial x)
;;   (/ (* x (+ x 2)) (square (+ x 1))))
;; (* 4 (accumulate combiner 1 factorial 2. next 1024))
;; 3.143
```

Iterative

```
(define (accumulate combiner null-value term a next b)
  (define (iter a null-value)
    (if (> a b)
    null-value
    (iter (next a) (combiner null-value (term a)))))
  (iter a null-value))
;; (define (combiner x y) (* x y))
;; (define (next x)
;;   (+ x 2))
;; (define (square x)
;;   (* x x))
;; (define (factorial x)
;;   (/ (* x (+ x 2)) (square (+ x 1))))
;; (* 4 (accumulate combiner 1 factorial 2. next 1024))
;; 3.143
```

## Exercise 1.33

Iterative

```
;; accumulate
;; filter on the term, filter with return value true will be added
(define (filtered-accumulate combiner null-value term a next b filter)
  (define (iter a null-value)
    (define one-term (term a))
    (cond ((> a b) null-value)
      ((filter one-term) (filtered-accumulate combiner (combiner one-
          term null-value) term (next a) next b filter))
      (else (filtered-accumulate combiner null-value term (next a) next
          b filter))))
  (iter a null-value))

;; prime?
(define square (lambda (x) (* x x)))
(define (expmod base exp m)
  (cond ((= exp 0) 1)
    ((even? exp)
     (remainder
      (square (expmod base (/ exp 2) m))
      m))
    (else
     (remainder
      (* base (expmod base (- exp 1) m))
      m))))
(define (prime? n)
  (if (= n 1) false
      (prime-test-all-a n 1)))
(define (prime-test-all-a n count)
  (cond ((= n count) true)
    ((= (expmod count n n) count) (prime-test-all-a n (+ count 1)))
    (else false)))


(define (inc x)
  (+ x 1))
(define (identify x)
  x)
(define (sum-square x y)
  (+ (square x) y))
(define (product x y)
  (* x y))

;; note that miller-rabin-test can only be used to odd numbers
(define (accumulate-primes-square a b)
  (filtered-accumulate sum-square 0 identify a inc b prime?))

```

```
(accumulate-primes-square 1 5)
;; result: 38

(define (accumulate-relatively-prime n)
  (define (relative-prime a)
    (= (gcd a n) 1))
  (filtered-accumulate product 1 identify 1 inc n relative-prime))

(accumulate-relatively-prime 10)
;; result: 189
```

## Exercise 1.34

```
(define (f g) (g 2))
(f f)
;; error obj 2 is not applicable
```

## Exercise 1.35

$$
\begin{aligned}
x &= 1 + \frac{1}{x} \\
x^2 &= x + 1 \\
x^2 - x - 1 &= 0 \\
x &= \frac{1 \pm \sqrt{5}}{2}
\end{aligned}
$$

```
(define tolerance 0.00001)
(define
  (fixed-point f first-guess)
  (define
    (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define
    (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
      next
      (try next))))
  (try first-guess))
(fixed-point (lambda (x) (+ 1 (/ 1 x))) 1.0)
;; result 1.618
```

## Exercise 1.36

```
(define tolerance 0.00001)
(define
  (fixed-point f first-guess)
  (define
    (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define
    (try guess)
    (newline)
    (display guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
      next
      (try next))))
  (try first-guess))

;; (fixed-point (lambda (x) (/ (log 1000) (log x))) 2)
;; 34 steps

(define (average x y)
  (/ (+ x y) 2))

(fixed-point (lambda (x) (average x (/ (log 1000) (log x)))) 2)
;; 9 steps
```

## Exercise 1.37

iterative

```
(define (cont-frac n d k)
  (define (cont-iter count result)
    (cond ((= count k)
       (cont-iter (- count 1) (/ (n count) (d count))))
      ((= count 0)
       result)
      (else (cont-iter (- count 1) (/ (n count) (+ (d count) result))))
         ))
  (cont-iter k 0))

(cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 11)
```

recursive

```
(define (cont-frac n d k)
  (define (cont-iter count)
```

```
    (cond ((= count k)
       (/ (n k) (d k)))
      (else (/ (n count) (+ (d count) (cont-iter (+ count 1)))))))
  (cont-iter 1))
(cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 11)
```

## Exercise 1.38

```
(define (cont-frac n d k)
  (define (cont-iter count result)
    (cond ((= count k)
       (cont-iter (- count 1) (/ (n count) (d count))))
      ((= count 0)
       result)
      (else (cont-iter (- count 1) (/ (n count) (+ (d count) result))))
         ))
  (cont-iter k 0))
(define (even? n) (= (remainder n 2) 0))
(define (square n) (* n n))
(define (exp b n)
    (fast-exp 1 b n))
(define (fast-exp a b n)
    (cond ((= n 0)
           a)
          ((even? n)
           (fast-exp a (square b) (/ n 2)))
          (else (fast-exp (* a b) b (- n 1)))))

(cont-frac (lambda (i) 1.0) (lambda (i) (cond ((= i 2) 2)
                        ((= (remainder (- i 2) 3) 0) (exp 2 (+ (/ (- i
                           2) 3) 1)))
                        (else 1)))  10)
;; 0.7182879
```

## Exercise 1.39

```
(define (tan-cf x k)
  (define (cont-frac n d k)
  (define (cont-iter count result)
    (cond ((= count k)
       (cont-iter (- count 1) (/ (n count) (d count))))
      ((= count 0)
       result)
      (else (cont-iter (- count 1) (/ (n count) (- (d count) result))))
         ))
```

```
  (cont-iter k 0))
  (cont-frac (lambda (i) (cond ((= i 1) x)
                   (else (* x x)))) (lambda (i) (- (* 2 i) 1)) 10))
(tan-cf 1.0 10)
;; 1.5574
```

## Exercise 1.40

```
(define tolerance 0.00001)
(define
  (fixed-point f first-guess)
  (define
    (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define
    (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
      next
      (try next))))
  (try first-guess))

(define
  (newton-transform g)
  (lambda (x)
    (- x (/ (g x) ((deriv g) x)))))

(define
  (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x)) dx)))

(define dx 0.00001)

(define (newtons-method g guess) (fixed-point (newton-transform g)
   guess))

(define (cubic a b c)
  (lambda (x) (+ (* x x x) (* a x x) (* b x) c)))

(newtons-method (cubic 1 1 1) 1)
```

## Exercise 1.41

```
(define (double f)
```

```
  (lambda (x) (f (f x))))

(define (inc x)
  (+ x 1))

(((double (double double)) inc) 5)
;; result: 21
```

## Exercise 1.42

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define (inc x)
  (+ x 1))

((compose square inc) 6)
;; 49
```

## Exercise 1.43

```
(define (compose f g)
  (lambda (x) (f (g x))))
(define (repeated f n)
  (lambda (x) (cond ((= n 1) (f x))
            (else ((compose f (repeated f (- n 1))) x)))))
((repeated square 2) 5)
;; result: 625
```

## Exercise 1.44

```
(define (smoothed f)
  (lambda (x)
    (/ (+ (f (- x dx)) (f x) (f (+ x dx))) 3)))
(define (compose f g)
  (lambda (x) (f (g x))))
(define (repeated f n)
  (lambda (x) (cond ((= n 1) (f x))
            (else ((compose f (repeated f (- n 1))) x)))))
(define dx 0.00001)

;; n fold smoothed
```

```
(((repeated smoothed 10) square) 5)
```

## Exercise 1.45

```
(define (average x y)
  (/ (+ x y) 2))

(define (average-damp f)
  (lambda (x) (average x (f x))))

(define (fixed-point-of-transform g transform guess) (fixed-point (
   transform g) guess))

(define tolerance 0.00001)
(define
  (fixed-point f first-guess)
  (define
    (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define
    (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
      next
      (try next))))
  (try first-guess))

(define (compose f g)
  (lambda (x) (f (g x))))

(define (repeated f n)
  (lambda (x) (cond ((= n 1) (f x))
            (else ((compose f (repeated f (- n 1))) x)))))

;; (define (f-for-n-root n x)
;;   (* x (repeated (lambda (y) (/ 1 y)) n)))

;; x: base, n: times
(define (root x n)
  (fixed-point-of-transform (lambda (y) (/ x (expt y (- n 1)))) (
     repeated average-damp (- n 1)) 1.0))

(root 32 5)
;; result: 2.0
```

## Exercise 1.46

```
(define (iterative-improve good-enough? improve)
  (lambda (x) (cond ((good-enough? x) x)
            (else ((iterative-improve good-enough? improve) (improve x)
               )))))

(define (average x y)
  (/ (+ x y) 2))

(define (sqrt x)
  (define (improve guess) (average guess (/ x guess)))
  (define (good-enough? guess) (< (abs (- (square guess) x)) 0.001))
  ((iterative-improve good-enough? improve) 1.0))

(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (good-enough? x)
    (< (abs (- x (f x))) tolerance))
  (define (improve x)
    (f x))
  ((iterative-improve good-enough? improve) 1.0))

(sqrt 4)
;; result: 2.0
(fixed-point cos 1.0)
;; result: 0.739
```