

Exercise 1.29

```
1 (define (sum term a next b)
2   (if (> a b)
3       0
4       (+ (term a)
5           (sum term (next a) next b))))
6 (define (cube x)
7   (* x x x))
8 (define (simpson f a b n)
9   (define h (/ (- b a) n))
10  ;; method 2 use 0-n sequence
11  (define (term x)
12    (+ (* 4 (f x) (/ (- b a) (* 3 n)))
13      (* 2 (f x) (/ (- b a) (* 3 n)))))
14  (define (next x)
15    (+ x (* 2 (/ (- b a) n))))
16  (sum term (+ a (* (f a) (/ (- b a) (* 3 n)))) next (- b (* 2 h)))
17 (simpson cube 0 1 1000)
```

Exercise 1.30

```
1 (define (sum term a next b)
2   (define (iter a result)
3     (if (> a b)
4         result
5         (iter (next a) (+ result (term a)))))
6   (iter a 0))
7 (define (identify x) x)
8 (define (inc x) (+ x 1))
9 (sum identify 1 inc 3)
```

Exercise 1.31

```
1 (define (product factorial a next b)
2   (define (iter a result)
3     (if (> a b)
4         result
5         (iter (next a) (* result (factorial a)))))
6   (iter a 1))
7 ;; test ok
8 ;; (define (identify x) x)
9 ;; (define (inc x) (+ x 1))
10 ;; (product identify 1 inc 4)
```

Exercise1-3

```
11 (define (next x)
12   (+ x 2))
13 (define (square x)
14   (* x x))
15 (define (factorial x)
16   (/ (* x (+ x 2)) (square (+ x 1))))
17 (* 4 (product factorial 2. next 1024))
18 ;; result = 3.1431
```

Exercise 1.32

Recursive

```
1 (define (accumulate combiner null-value term a next b)
2   (if (> a b)
3       null-value
4       (accumulate combiner (combiner (term a) null-value) term (next a)
5                     next b)))
5 ;; (define (combiner x y) (* x y))
6 ;; (define (next x)
7 ;;   (+ x 2))
8 ;; (define (square x)
9 ;;   (* x x))
10 ;; (define (factorial x)
11 ;;   (/ (* x (+ x 2)) (square (+ x 1))))
12 ;; (* 4 (accumulate combiner 1 factorial 2. next 1024))
13 ;; 3.143
```

Iterative

```
1 (define (accumulate combiner null-value term a next b)
2   (define (iter a null-value)
3     (if (> a b)
4         null-value
5         (iter (next a) (combiner null-value (term a)))))
6   (iter a null-value))
7 ;; (define (combiner x y) (* x y))
8 ;; (define (next x)
9 ;;   (+ x 2))
10 ;; (define (square x)
11 ;;   (* x x))
12 ;; (define (factorial x)
13 ;;   (/ (* x (+ x 2)) (square (+ x 1))))
14 ;; (* 4 (accumulate combiner 1 factorial 2. next 1024))
15 ;; 3.143
```

Exercise 1.33

Iterative

```
1 ;; accumulate
2 ;; filter on the term, filter with return value true will be added
3 (define (filtered-accumulate combiner null-value term a next b filter)
4   (define (iter a null-value)
5     (define one-term (term a))
6     (cond ((> a b) null-value)
7           ((filter one-term) (filtered-accumulate combiner (combiner one-
8             term null-value) term (next a) next b filter))
9           (else (filtered-accumulate combiner null-value term (next a) next
10             b filter))))
11   (iter a null-value))
12
13 ;; prime?
14 (define square (lambda (x) (* x x)))
15 (define (expmod base exp m)
16   (cond ((= exp 0) 1)
17         ((even? exp)
18          (remainder
19           (square (expmod base (/ exp 2) m))
20           m))
21         (else
22          (remainder
23           (* base (expmod base (- exp 1) m))
24           m))))
25 (define (prime? n)
26   (if (= n 1) false
27        (prime-test-all-a n 1)))
28 (define (prime-test-all-a n count)
29   (cond ((= n count) true)
30         ((= (expmod count n n) count) (prime-test-all-a n (+ count 1)))
31         (else false)))
32
33 (define (inc x)
34   (+ x 1))
35 (define (identify x)
36   x)
37 (define (sum-square x y)
38   (+ (square x) y))
39 (define (product x y)
40   (* x y))
41
42 ;; note that miller-rabin-test can only be used to odd numbers
43 (define (accumulate-primes-square a b)
44   (filtered-accumulate sum-square 0 identify a inc b prime?))
```

Exercise1-3

```
45 (accumulate-primes-square 1 5)
46 ;; result: 38
47
48 (define (accumulate-relatively-prime n)
49   (define (relative-prime a)
50     (= (gcd a n) 1))
51   (filtered-accumulate product 1 identify 1 inc n relative-prime))
52
53 (accumulate-relatively-prime 10)
54 ;; result: 189
```

Exercise 1.34

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

Exercise 1.35

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

```
1 (define tolerance 0.00001)
2 (define
3   (fixed-point f first-guess)
4   (define
5     (close-enough? v1 v2)
6     (< (abs (- v1 v2)) tolerance))
7   (define
8     (try guess)
9     (let ((next (f guess)))
10      (if (close-enough? guess next)
11          next
12          (try next))))
13   (try first-guess))
14 (fixed-point (lambda (x) (+ 1 (/ 1 x))) 1.0)
15 ;; result 1.618
```

Exercise 1.36

```
1 (define tolerance 0.00001)
2 (define
3   (fixed-point f first-guess)
4   (define
5     (close-enough? v1 v2)
6     (< (abs (- v1 v2)) tolerance))
7   (define
8     (try guess)
9     (newline)
10    (display guess)
11    (let ((next (f guess)))
12      (if (close-enough? guess next)
13          next
14          (try next))))
15   (try first-guess))
16
17 ;; (fixed-point (lambda (x) (/ (log 1000) (log x))) 2)
18 ;; 34 steps
19
20 (define (average x y)
21   (/ (+ x y) 2))
22
23 (fixed-point (lambda (x) (average x (/ (log 1000) (log x)))) 2)
24 ;; 9 steps
```

Exercise 1.37

iterative

```
1 (define (cont-frac n d k)
2   (define (cont-iter count result)
3     (cond ((= count k)
4           (cont-iter (- count 1) (/ (n count) (d count))))
5           ((= count 0)
6            result)
7           (else (cont-iter (- count 1) (/ (n count) (+ (d count) result))))))
8   (cont-iter k 0))
9
10 (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 11)
```

recursive

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

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

Exercise 1.38

```
1 (define (cont-frac n d k)
2   (define (cont-iter count result)
3     (cond ((= count k)
4           (cont-iter (- count 1) (/ (n count) (d count))))
5           ((= count 0)
6            result)
7           (else (cont-iter (- count 1) (/ (n count) (+ (d count) result))))
8           ))
9   (cont-iter k 0))
10 (define (even? n) (= (remainder n 2) 0))
11 (define (square n) (* n n))
12 (define (exp b n)
13   (fast-exp 1 b n))
14 (define (fast-exp a b n)
15   (cond ((= n 0)
16         a)
17         ((even? n)
18          (fast-exp a (square b) (/ n 2)))
19         (else (fast-exp (* a b) b (- n 1)))))
20 (cont-frac (lambda (i) 1.0) (lambda (i) (cond ((= i 2) 2)
21                                              ((= (remainder (- i 2) 3) 0) (exp 2 (+ (/ (- i
22                                              2) 3) 1)))
23                                              (else 1))) 10)
23 ;; 0.7182879
```

Exercise 1.39

```
1 (define (tan-cf x k)
2   (define (cont-frac n d k)
3     (define (cont-iter count result)
4       (cond ((= count k)
5             (cont-iter (- count 1) (/ (n count) (d count))))
6             ((= count 0)
7              result)
8             (else (cont-iter (- count 1) (/ (n count) (- (d count) result))))
9             ))
10    (cont-iter 1 x k))
```

Exercise1-3

```
9 (cont-iter k 0))
10 (cont-frac (lambda (i) (cond ((= i 1) x)
11                             (else (* x x)))) (lambda (i) (- (* 2 i) 1)) 10))
12 (tan-cf 1.0 10)
13 ;; 1.5574
```

Exercise 1.40

```
1 (define tolerance 0.00001)
2 (define
3   (fixed-point f first-guess)
4   (define
5     (close-enough? v1 v2)
6     (< (abs (- v1 v2)) tolerance))
7   (define
8     (try guess)
9     (let ((next (f guess)))
10      (if (close-enough? guess next)
11          next
12          (try next))))
13   (try first-guess))
14
15 (define
16   (newton-transform g)
17   (lambda (x)
18     (- x (/ (g x) ((deriv g) x)))))
19
20 (define
21   (deriv g)
22   (lambda (x)
23     (/ (- (g (+ x dx)) (g x)) dx)))
24
25 (define dx 0.00001)
26
27 (define (newtons-method g guess) (fixed-point (newton-transform g)
28                                                guess))
29
30 (define (cubic a b c)
31   (lambda (x) (+ (* x x x) (* a x x) (* b x) c)))
32
33 (newtons-method (cubic 1 1 1) 1)
```

Exercise 1.41

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

Exercise1-3

```
2 (lambda (x) (f (f x)))
3
4 (define (inc x)
5   (+ x 1))
6
7 (((double (double double)) inc) 5)
8 ;; result: 21
```

Exercise 1.42

```
1 (define (compose f g)
2   (lambda (x) (f (g x))))
3
4 (define (inc x)
5   (+ x 1))
6
7 ((compose square inc) 6)
8 ;; 49
```

Exercise 1.43

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

Exercise 1.44

```
1 (define (smoothed f)
2   (lambda (x)
3     (/ (+ (f (- x dx)) (f x) (f (+ x dx))) 3)))
4 (define (compose f g)
5   (lambda (x) (f (g x))))
6 (define (repeated f n)
7   (lambda (x) (cond ((= n 1) (f x))
8                     (else ((compose f (repeated f (- n 1))) x)))))
9 (define dx 0.00001)
10
11 ;; n fold smoothed
```



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

Exercise 1.45

```
1 (define (average x y)
2   (/ (+ x y) 2))
3
4 (define (average-damp f)
5   (lambda (x) (average x (f x))))
6
7 (define (fixed-point-of-transform g transform guess) (fixed-point (
8   transform g) guess))
9
10 (define tolerance 0.00001)
11 (define
12   (fixed-point f first-guess)
13   (define
14     (close-enough? v1 v2)
15     (< (abs (- v1 v2)) tolerance))
16   (define
17     (try guess)
18     (let ((next (f guess)))
19       (if (close-enough? guess next)
20         next
21         (try next))))
22   (try first-guess))
23
24 (define (compose f g)
25   (lambda (x) (f (g x))))
26
27 (define (repeated f n)
28   (lambda (x) (cond ((= n 1) (f x))
29                     (else ((compose f (repeated f (- n 1))) x)))))
30
31 ;; (define (f-for-n-root n x)
32 ;;   (* x (repeated (lambda (y) (/ 1 y)) n)))
33
34 ;; x: base, n: times
35 (define (root x n)
36   (fixed-point-of-transform (lambda (y) (/ x (expt y (- n 1)))) (
37     repeated average-damp (- n 1)) 1.0))
38
39 (root 32 5)
40 ;; result: 2.0
```

Exercise 1.46

```
1 (define (iterative-improve good-enough? improve)
2   (lambda (x) (cond ((good-enough? x) x)
3                     (else ((iterative-improve good-enough? improve) (improve x)
4                               )))))
5 (define (average x y)
6   (/ (+ x y) 2))
7
8 (define (sqrt x)
9   (define (improve guess) (average guess (/ x guess)))
10  (define (good-enough? guess) (< (abs (- (square guess) x)) 0.001))
11  ((iterative-improve good-enough? improve) 1.0))
12
13 (define tolerance 0.00001)
14 (define (fixed-point f first-guess)
15   (define (good-enough? x)
16     (< (abs (- x (f x))) tolerance))
17   (define (improve x)
18     (f x))
19   ((iterative-improve good-enough? improve) 1.0))
20
21 (sqrt 4)
22 ;; result: 2.0
23 (fixed-point cos 1.0)
24 ;; result: 0.739
```