```
;Exercise 1.1.  Below is a sequence of expressions. What is the result printed
by the interpreter in response to each expression? Assume that the sequence is
to be evaluated in the order in which it is presented.
;10
;(+ 5 3 4)
;(- 9 1)
;(/ 6 2)
;(+ (* 2 4) (- 4 6))
;(define a 3)
;(define b (+ a 1))
;(+ a b (* a b))
;(= a b)
;(if (and (> b a) (< b (* a b)))
;    b
;    a)
;(cond ((= a 4) 6)
;      ((= b 4) (+ 6 7 a))
;      (else 25))
;(+ 2 (if (> b a) b a))
;(* (cond ((> a b) a)
;         ((< a b) b)
;         (else -1))
;   (+ a 1))

;Exercise 1.2.
;(/
; (+ 5 4
;    (- 2
;       (- 3 (+ 6 (/ 4 3)))))
; (* 3
;    (- 6 2)
;    (- 2 7)))

;Exercise 1.3
(define square
  (lambda (x)
    (* x x)))

(define sum-of-two-square-larger
  (lambda (a1 a2 a3)
    (cond ((and (< a1 a2) (< a1 a3)) (+ (square a2) (square a3)))
          ((and (< a2 a1) (< a2 a3)) (+ (square a1) (square a3)))
          (else
           (+ (square a1) (square a2))))))
```

```
;;Exercise 1.4
;a+ |b|

;Exercise 1.5.  Ben Bitdiddle has invented a test to determine whether the
interpreter he is faced with is using applicative-order evaluation or normal-
order evaluation. He defines the following two procedures:
;(define (p) (p))
;(define (test x y)
;  (if (= x 0)
;      0
;      y))
;
;Then he evaluates the expression
;(test 0 (p))

;It is a no end loop as applicative-order



;Exercise 1.6
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))

(define (improve guess x)
  (average guess (/ x guess)))

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

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (my-sqrt x)
  (sqrt-iter 1.0 x))

(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))

;(define (sqrt-iter guess x)
;  (new-if (good-enough? guess x)
```

```
;          guess
;          (sqrt-iter (improve guess x)
;                     x)))
;(my-sqrt 9)
;;This gets a no end loop,because new-if is defined as a procedure,so in
evaluation we use the
;;substitution model, eval all the arguments first,(sqrt-iter ...) as the third
argument will be evaluated
;;before the new-if,so loop gets no end. We use IF primitvie procedure to end
when the guess is ok enough

;;Exericse 1.7
(define sqrt-exercisel.7
  (lambda (x)
    (define tolerence 0.00000000001)
    (define good-enough?
      (lambda (new-guess old-guess)
        (< (abs (- old-guess new-guess)) tolerence)))
    (define (sqrt-iter new-guess guess)
      (if (good-enough? new-guess guess)
          new-guess
          (sqrt-iter (improve new-guess)
                     new-guess)))
    (define (improve guess)
      (average guess (/ x guess)))
    (sqrt-iter (improve 1.0) 1.0)))

;;Exercise 1.8 :I use the same strage as in Exercise 1.7, so I just modify the
imporve method
(define cube-root
  (lambda (x)
    (define tolerence 0.00000000001)
    (define good-enough?
      (lambda (new-guess old-guess)
        (< (abs (- old-guess new-guess)) tolerence)))
    (define (sqrt-iter new-guess guess)
      (if (good-enough? new-guess guess)
          new-guess
          (sqrt-iter (improve new-guess)
                     new-guess)))
    (define (improve guess)
      (/ (+ (/ x (square guess))
            (* 2 guess))
         3))
```

```
    (sqrt-iter (improve 1.0) 1.0)))


;Exercise 1.9.  Each of the following two procedures defines a method for adding
two positive integers in terms of the procedures inc, which increments its
argument by 1, and dec, which decrements its argument by 1.

(define (my+ a b)
  (if (= a 0)
      b
      (inc (my+ (dec a) b))))

(define (my+ a b)
  (if (= a 0)
      b
      (my+ (dec a) (inc b))))

(define (inc value)
  (+ 1 value))
(define (dec value)
  (- value 1))
;;Version1 is a recursvie one, version2 is a iterative one
;;ues the test (my+ 1000000 1000000) the version1 will cost out of the
memrory......this is included in the course vedios.

;Exercise 1.10
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))

;(A 1 10) ;2的10次方
;(A 2 4) ;2的16次方
;(A 3 3) ;2的16次方
;(A 2 3)
;(A 2 1)

(define (f n) (A 0 n)) ;;2*n

(define (g n) (A 1 n)) ;;2的n次方

(define (h n) (A 2 n)) ;;2^2^2^2 n个2
```

```
(define (k n) (* 5 n n))

;;Exercise 1.11
(define compute
  (lambda (n)
    (if(< n 3)
       n
       (+ (compute (- n 1))
          (* 2 (compute (- n 2)))
          (* 3 (compute (- n 3)))))))

;;Exercise 1.12
(define pasca1-triangle
  (lambda(n k)
    (if (or (= k 1) (= k n))
        1
        (+ (pasca1-triangle (- n 1) (- k 1))
           (pasca1-triangle (- n 1) k)))))

;;Exercise 1.13
;已知,φ^2 = φ+1, 那么 φ^n = φ^(n-1) + φ^(n-2)
;同理, ψ^2 = ψ+1, 那么 ψ^n = ψ^(n-1) + ψ^(n-2)

;;Exercise 1.14
;;the number of steps required by a tree-recursive process will be proportional
to the number of nodes in the tree,
;;the space required will be proportional to the maximum depth of the tree.
;;space O(n)
;;step O(n^k)  I can't get K =_=!!

;;Exercise 1.15
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
   (if (not (> (abs angle) 0.1))
       angle
       (p (sine (/ angle 3.0)))))
;a  12.15连除5次 3.0 小于0.1  所以p调用5次
;b  space and time all cost O(loga), O(times of p applied)



;;Exercise 1.16
(define square
```

```
  (lambda (n)
    (* n n)))
(define iterative-expt
  (lambda (b n)
    (define iter
      (lambda (base N product)
        (cond ((= N 0) product)
              ((even? N) (iter (square base) (/ N 2) product))
              (else (iter base
                          (- N 1)
                          (* product base))))))
    (iter b n 1)))

(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
;;Exercise 1.17
(define (double x)
  (* x 2))
(define fast-multiply
  (lambda (a b)
    (cond((= b 1) a)
         ((even? b) (fast-multiply (double a) (/ b 2)))
         (else
          (+ a
             (fast-multiply a (- b 1)))))))

;;Exercise 1.18
(define halve
  (lambda (x)
    (/ x 2)))
(define iter-multiply
  (lambda (a b)
    (define iter
      (lambda (a b product-now)
        (cond((= b 0) product-now)
             ((even? b) (iter (double a) (halve b) product-now))
             (else
              (iter a (- b 1) (+ a product-now))))))
    (iter a b 0)))

;;Exercise 1.19
(define (fib n)
```

```
  (fib-iter 1 0 0 1 n))
;;p=p^2+ q^2
;;q=q^2+2pq
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   (+ (* p p) (* q q))
                   (+ (* 2 p q ) (* q q))
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))

;;Exercise 1.20
;applicative-order evaluation: 4 times
;normal-order evaluation: 18 times

;;Exericse 1.21
(define (smallest-divisor n)
  (find-divisor n 2))
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))
(define (divides? a b)
  (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))

;(smallest-divisor 19999)
;(smallest-divisor 1999)
;(smallest-divisor 199)

;;Exercise 1.22
(define (timed-prime-test n)
  (newline)
  (display n)
  (start-prime-test n (runtime)))
```

```
(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (runtime) start-time))))
(define (report-prime elapsed-time)
  (display " *** ")
  (display (/ elapsed-time 1000)))

(define (runtime) (current-milliseconds)) ;;There is no runtime primitive
procedure in Drscheme

(define (search-for-primes start end)
  (define (iter n)
    (timed-prime-test n)
    (if (< n end)
        (iter (+ n 1))))
  (iter start))
;;The results in my computer is all 0 for each prime number
;;my computer is too fast? ;)

;;Exercise 1.23
(define next
  (lambda (input)
    (if (= input 2)
        3
        (+ input 2))))
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (next test-divisor)))))

;;Exercise 1.24
;;Fermat's little theory: if n is prime,and a<n ===> (a^n)mod n=a
;;this is a Probabilistic Method,so we test if (a^n)mod n !=a ===> n is not
prime
;;if (a^n)mode n=a, so the chances are better that n is a prime
;;probabilistic algorithms
;称满足 a^(n-1) mod n = 1 的合数 n 叫做以 a 为底的伪素数(pseudoprime to base a)。
;统计表明:在前 10 亿个自然数中共有 50847534 个素数，而满足 2^(n-1) mod n = 1 的合数 n 有
5597 个。
;这样算下来，算法出错的可能性约为 0.00011。
;通过所有 a 为底数的和书称为 Carmichael 数。最小的为 561，前 10 亿个当中有 600 个。
```

```
(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 (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))

(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))

(define (start-prime-test n start-time)
  (if (fast-prime? n 1) ;;just test one time, so the result may be wrong.....the
character of probabilistic algorithms
      (report-prime (- (runtime) start-time))));;we can test more time for each
number,so we can get a good result

(define (search-for-primes start end)
  (define (iter n)
    (timed-prime-test n)
    (if (< n end)
        (iter (+ n 1))))
  (iter start))

;(search-for-primes 2 100) ;for n=1 the random procedure will get wrong argument

;;Exercise 1.25
;;The fast-expt also cost time O(log n),int the (remainder (fast-expt base exp)
m))
;;(fast-expt base exp) maybe a very big number,it will take very long time
because bignum arithmetic takes much much ;;longer than ordinary arithmetic.

;;Exercise 1.26
;In the case of expmod, instead of logarithmic runtime we get linear runtime
(2^logN = N).
(define (louis-expmod base exp m)
```

```
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (* (louis-expmod base (/ exp 2) m)
                       (louis-expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (louis-expmod base (- exp 1) m))
                    m))))


;;Exercise 1.27
;Numbers that fool the Fermat test are called Carmichael numbers, and little is known about them other than that they are extremely rare. There are 255 Carmichael numbers below 100,000,000. The smallest few are 561, 1105, 1729, 2465, 2821, and 6601
(define test-all-below
  (lambda (N)
    (define (iter test-now)
      (cond ((= test-now N) true)
            (else
             (if (try-it test-now)
                 (iter (+ test-now 1))
                 false))))
    (define (try-it a)
    (= (expmod a N N) a))
    (iter 1)))
;This procedure test all the number below N

;;Exercise 1.28
;p 是素数， 1<a<p，且 a^2 % p = 1
;==> (a^2-1) % p = 0 ==> (a+1)(a-1) % p =0
;那么 a+1 % p = 0 或者 a-1 % p =0，
;又 a<p 且 p 是素数，所以
;a = 1 或者 a = p-1

;演示一下上面的定理如何应用在Fermat素性测试上。341可以通过以2为底的Fermat测试，
;因为 2^340 mod 341=1。如果341真是素数的话，那么2^170 mod 341只可能是1或340；当算得2^170 mod 341确实等于1时，继续查看2^85除以341的结果，2^85 mod 341=32，这一结果说明341不是素数。
;这就是Miller-Rabin素性测试的方法:不断地提取指数n-1中的因子2，把n-1表示成d*2^r（其中d是一个奇数）。
;那么我们需要计算的东西就变成了a的d*2^r次方除以n的余数。于是，a^(d * 2^(r-1))要么等于1，要么等于n-1。
;如果a^(d * 2^(r-1))等于1，定理继续适用于a^(d * 2^(r-2))，这样不断开方开下去，直到对于
```

某个i满足a^(d * 2^i) mod n = n-1或者最后指数中的2用完了得到的a^d mod n=1或n-1。

```
(define expmod
  (lambda (base exp m)
    (cond((= exp 0) 1)
         ((even? exp)
          (check? (expmod base (/ exp 2) m) m))
         (else
           (remainder (* base (expmod base (- exp 1) m)) m)))))

(define check?
  (lambda (number mod)
       (let ((temp (remainder (square number) mod)))
         (if(and (> number 1)
                 (< number (- mod 1))
                 (= temp 1))
            0
            temp))))

(define miller-rabin
  (lambda (N)
    (define (iter a)
      (cond ((= a 0) #t)
            ((= (expmod a (- N 1) N) 1) (iter (- a 1)))
            (else #f)))
    (iter (- N 1))))

(define print-out-prime
  (lambda (start end)
    (define (iter index)
      (if (> index end)
          'end
          (begin
            (display index)
            (if (miller-rabin index)
                (display " Prime\n")
                (newline))
            (iter (+ index 1)))))
    (iter start)))

;;1.3  Formulating Abstractions with Higher-Order Procedures
;;Exercise 1.29
(define (sum term a next b)
  (if (> a b)
      0
```

```
      (+ (term a)
         (sum term (next a) next b))))

(define Simpson
  (lambda (func a b N)
    (let((h (/ (- b a) N)))
      (define (new-fun k)
        (let ((temp (+ a (* k h))))
        (cond((or (= k 0) (= k N)) (func temp))
             ((even? k) (* (func temp) 2))
             (else (* (func temp) 4)))))
      (define (next value)
        (+ value 1))
      (* (/ h 3)
         (sum new-fun 0 next N)))))

;;Exercise 1.30
(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (+ a 1) (+ result (term a)))))
  (iter a 0))

;;Exercise 1.31
(define (product term a next b)
  (define (iter start product-now)
    (if(> start b)
       product-now
       (iter (next start) (* product-now (term start)))))
  (iter a 1))

(define factorial
  (lambda (N)
    (product (lambda (x) x)
             1
             (lambda (x) (+ x 1))
             N)))
(define Pi
  (lambda (N)
    (define (func value)
      (/ (- (square value) 1)
         (square value)))
    (define (next value)
```

```
      (+ value 2))
    (product func
            3
            next N)))

;;the recursive version
(define product-recursive
  (lambda (term a next b)
    (if (> a b)
        1
        (* (term a)
           (product-recursive term (next a) next b)))))

;;Exercise 1.32
(define accumulate
  (lambda (combiner null-value term a next b)
    (define computer
      (lambda (start)
        (if (> start b)
            null-value
            (combiner (term start)
                      (computer (next start))))))
    (computer a)))
;;the sum
(define sum
  (lambda (term a next b)
    (accumulate + 0 term a next b)))

;test: (sum (lambda(x) x) 1 (lambda(x) (+ x 1)) 100)
;;the product
(define product
  (lambda (term a next b)
    (accumulate * 1 term a next b)))
;test: (product (lambda(x) x) 1 (lambda(x) (+ x 1)) 4)

(define accumulate-iteractive
  (lambda (combiner null-value term a next b)
    (define (iter start now-value)
      (if(> start b)
         now-value
         (iter (next start)
               (combiner (term start) now-value))))
    (iter a null-value)))
```

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

;test: (sum (lambda(x) x) 1 (lambda(x) (+ x 1)) 100)

;;Exercise 1.33
(define filtered-accumulate
  (lambda (combiner null-value predicate? term a next b)
    (define (iter start now-value)
      (if(> start b)
         now-value
         (if(predicate? start)
            (iter (next start) (combiner (term start) now-value))
            (iter (next start) now-value))))
    (iter a null-value)))
(define sum-of-square-of-prime
  (lambda (a b)
    (filtered-accumulate
     + 0 Prime?
     (lambda(x) (* x x))
     a
     (lambda(x) (+ x 1))
     b)))
(define Prime? miller-rabin)
;;(sum-of-sqaure-of-prime 1 10)

;;Exercise 1.34
(define (f g)
  (g 2))
;result:precedure application: expected procedure, given: 2; arguments were: 2

;;Exercise 1.35
(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 golden-rate
  (fixed-point (lambda (x) (+ 1 (/ 1 x)))
               1.0))

;;Exercise 1.36
(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
          (begin
            (newline)
            (display guess)
            (try next)))))
  (try first-guess))

;(define solution
;  (fixed-point
;   (lambda (x) (/ (log 1000) (log x)))
;   1.1))
;
;(define solution-average-damping
;  (fixed-point
;   (lambda (x) (/ (+ x (/ (log 1000) (log x)))
;                  2))
;   1.1))

;;cost 11 times,less than prevese version

;;Exercise 1.37
(define  (cont-frac n d k)
  (define loop
    (lambda(i)
      (if (= i k)
          (/ (n i) (d i))
          (/ (n i) (+ (d i) (loop (+ i 1)))))))
  (loop 1))

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

```
            k)))
;(/ 1 (test 12))
;k=12  we can get accurate to 4 decimal places

(define (cont-frac-iter n d k)
  (define iter
    (lambda (index now-value)
      (if (= index 0)
          now-value
          (iter (- index 1) (/ (n index)
                               (+ (d index) now-value))))))
  (iter k 0))

(define test
  (lambda (k)
    (cont-frac-iter (lambda(i) 1.0)
                    (lambda(i) 1.0)
                    k)))
;;Exericse 1.38
(define euler-d
  (lambda(i)
    (let ((k (remainder (- i 2) 3)))
      (if(= k 0)
         (* (+  (/ (- i 2) 3) 1) 2.0)
         1.0))))

(define E-2
  (lambda (k)
    (cont-frac-iter (lambda(i) 1.0)
                    euler-d
                    k)))
(define natural
  (lambda (k)
    (+ 2 (E-2 k))))

;;Exercise 1.39
(define tangent
  (lambda(x k)
    (define func-N
      (lambda(i)
        (if(= i 1)
           x
           (- (* x x)))))
    (define func-D
```

```
      (lambda(i)
        (- (* 2.0 i) 1.0)))
    (cont-frac-iter func-N func-D k)))

;;Exercise 1.40
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))
(define dx 0.00001)
(define (newton-transform g)
  (lambda (x)
    (- x (/ (g x) ((deriv g) x)))))

(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)))

(define zero-point-cubic
  (lambda(a b c)
    (newtons-method (cubic a b c) 1)))

;;Exercise 1.42
(define double
  (lambda (func)
    (lambda (input)
      (func (func input)))))
;answer 21

;;Exercise 1.43
(define compose-func
  (lambda (func-f func-g)
    (lambda (input)
      (func-f (func-g input)))))
;((compose square inc) 6)

;;Exercise 1.44
(define repeat
  (lambda (func times)
    (lambda (input)
      (define (loop index now-value)
```

```
       (if(= index times)
          (func now-value)
          (loop (+ index 1) (func now-value))))
     (loop 1 input))))
;;use the compose-func
(define repeat
  (lambda (func times)
    (if (= times 1)
       func
       (compose func
                (repeat func (- times 1))))))
;;Exercise 1.45
(define smoothed
  (lambda (func)
    (lambda (input)
      (/
       (+ (func (+ input dx))
          (func input)
          (func (- input dx)))
       3))))

(define n-fold-smoothed
  (lambda(func times)
     (repeat (smoothed func) times)))

;;Exercise 1.46
(define (nth-root-test x n k)
  (fixed-point ((repeat average-damp k) (lambda (y) (/ x (fast-expt y (- n
1)))))
             1.0))

(define average-damp
  (lambda (func)
    (lambda (input)
      (average (func input) input))))

;k = log2n
(define (nth-root x n)
  (let ((k (round (/ (log n) (log 2)))))
    (fixed-point ((repeat average-damp k) (lambda (y) (/ x (fast-expt y (- n
1)))))
               1.0)))
;;Exercise 1.47
(define iteractive-improve
```

```
(lambda (good? improve)
  (lambda (guess)
    (define (loop a-guess)
      (if(good? a-guess)
         a-guess
         (loop (improve a-guess))))
    (loop guess))))
```