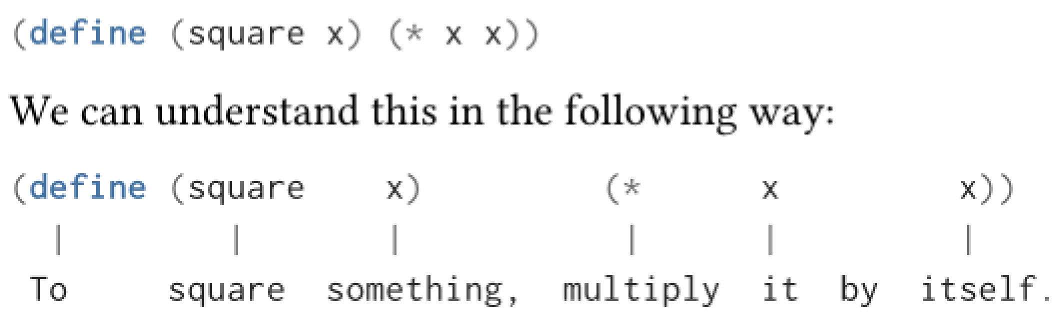
Compound Procedures

计算平方数



```
(define (square x) (* x x))
```

We can understand this in the following way:

```
(define (square    x)          (*      x         x))
  |         |       |           |       |         |
 To      square  something,  multiply  it   by  itself.
```

两种展开计算方式

有以下定义的过程：

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

对于计算 `f(5)` 有两种计算方式:

1. 将参数相应的带入计算 (***applicative-order***)

    ```
    f(5)
    (sum-of-squares (+ 5 1) (* 5 2))
    (+ (square 6) (square 10))
    (+ 36 100)
    136
    ```

2. 带入参数，直到需要计算它的值 (***normal-order***)

    ```
    f(5)
    (sum-of-squares (+ 5 1) (* 5 2))
    (+ (square (+ 5 1)) (square (* 5 2)))
    (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
    (+ (* 6 6) (* 10 10))
    (+ 36 100)
    136
    ```

Lisp use applicative-order evaluation

Conditional Expressions and Predicates

$$|x| = \begin{cases} x & \text{if} \quad x > 0, \\ 0 & \text{if} \quad x = 0, \\ -x & \text{if} \quad x < 0. \end{cases}$$

This construct is called a *case analysis*, and there is a special form in Lisp for notating such a case analysis. It is called cond (which stands for "conditional"), and it is used as follows:

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

This uses the special form `if`, a restricted type of conditional that can be used when there are precisely two cases in the case analysis. The general form of an `if` expression is

```
(if ⟨predicate⟩ ⟨consequent⟩ ⟨alternative⟩)
```

`if` 和 `cond` 不同的是， `if` 是一种特殊的形式，当它的 `predicate` 部分为真时， `then-clause` 分支会被求值，否则的话， `else-clause` 分支被求值，两个 `clause` 只有一个会被求值。

Lisp operators are compound expressions

Exercise 1.4: Observe that our model of evaluation allows for combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

whether the interpreter useing applicative-order or normal-order

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))
```

- 假设使用applicative order

```
(test 0 (p))
(test 0 (p))
(test 0 (p))
......
; 将参数带入 (p) 计算导致一直循环
```

- 假设使用normal order

```
(test 0 (p))
(test 0 (p))
0
; 将参数带入，但是不会立即求其值，到需要计算时计算，所以(p)一直没有被计算
```

Newtod's Method

How does one compute square roots? The most common way is to use Newton's method of successive approximations, which says that whenever we have a guess y for the value of the square root of a number x, we can perform a simple manipulation to get a better guess (one closer to the actual square root) by averaging y with x/y.[21] For example, we can compute the square root of 2 as follows. Suppose our initial guess is 1:

```
Guess       Quotient                Average
1           (2/1) = 2               ((2 + 1)/2) = 1.5
1.5         (2/1.5) = 1.3333        ((1.3333 + 1.5)/2) = 1.4167
1.4167      (2/1.4167) = 1.4118     ((1.4167 + 1.4118)/2) = 1.4142
1.4142      ...                     ...
```

```
; newton method
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

```scheme
(define (sqrt-iter guess x)
  (if (good-enougth? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (improve guess x)
  (average guess (+ guess x)))

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

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

对于函数 $f(x) = x^2 - n$ 求平方根也就是求 x , 也就是函数 $f(x) = 0$ 的时候 x 的值， 首先有一个猜测值 $(x_0, f(x_0))$, 可以得到过此点的切线方程如下:

$$f(x) - f(x_0) = f^{'}(x_0)(x - x_0)$$

也就是求这个切线方程在纵坐标为 $f(x) = 0$ 的时候横坐标 x, 也就是:

$$x = x_0 - \frac{f(x_0)}{f^{'}(x_0)}$$
$$f^{'}(x) = \frac{f(x+dx) - f(x)}{dx}$$

然后取这个 x 作为上一个的 x_0 继续这个步骤， 会持续逼近函数 $f(x) = x^2 - n$ ， $f(x) = 0$ 的时候 x 的值， 也就是平方根。

```scheme
(define dx 0.00001)
(define tolerance 0.00001)
; 不动点迭代
(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))
; 求微分
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx))
          (g x))
```

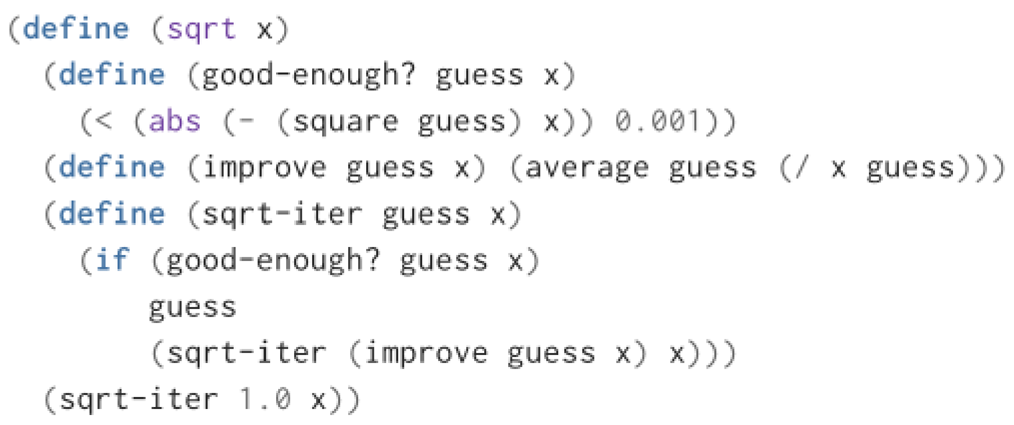
```scheme
       dx)))

(define (newton-transform g)
  (lambda (x)
    (- x (/ (g x)
            ((deriv g) x)))))
; 牛顿法
(define (newton-method g guess)
  (fixed-point (newton-transform g)
               guess))
; 求平方根
(define (sqrt x)
  (newton-method (lambda (y)
                   (- (* y y)
                      x))
                 1.0))

; sqrt x
(sqrt 2)
```

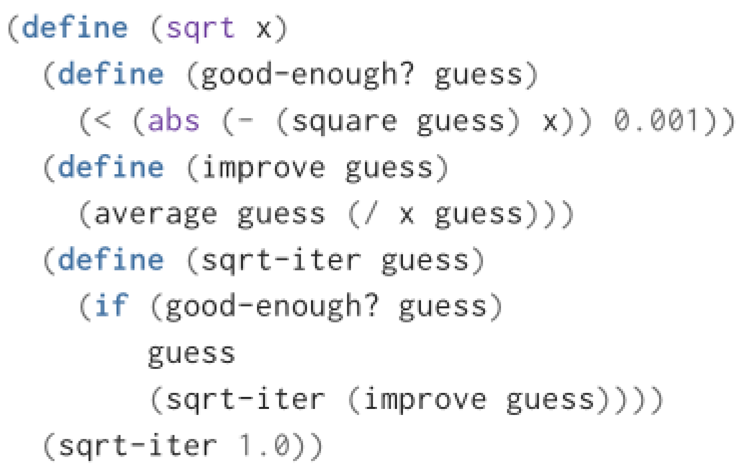
Internal definitions and block structure

块结构可以避免大型项目中，函数依赖混乱的问题



```scheme
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x) (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

Internal definitions 可以进一步有优化块结构，`x` 是可以子结构公用的



```scheme
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

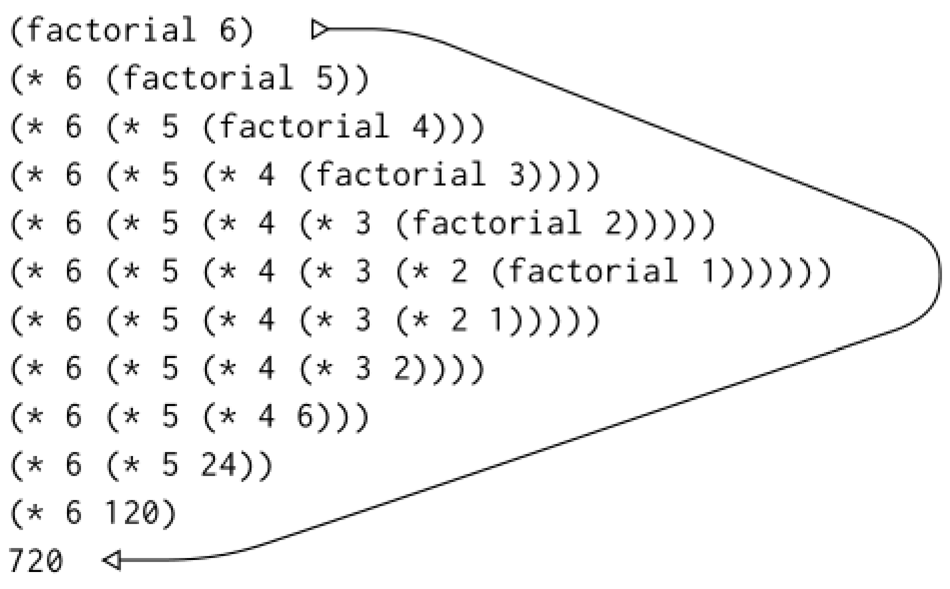
Linear Recursion and Iteration

考虑计算阶乘 `n!`:

$$n! = n \times (n-1) \times (n-2) \ldots 3 \times 2 \times 1$$

- ***A linear recursive process***

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))
   )
)
```

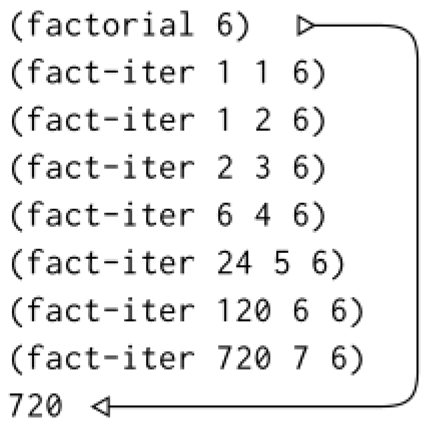


```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Figure 1.3: A linear recursive process for computing 6!.

- ***A linear iterative process***

```
(define (factorial n)
  (define (fact-iter result count)
    (if (> count n)
        result
        (fact-iter (* result count) (+ count 1))
     )
   )
 )
```



```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

Figure 1.4: A linear iterative process for computing 6!.

Coin Change

This problem has a simple solution as a recursive procedure. Suppose we think of the types of coins available as arranged in some order. Then the following relation holds:

The number of ways to change amount a using n kinds of coins equals

- the number of ways to change amount a using all but the first kind of coin, plus
- the number of ways to change amount $a - d$ using all n kinds of coins, where d is the denomination of the first kind of coin.

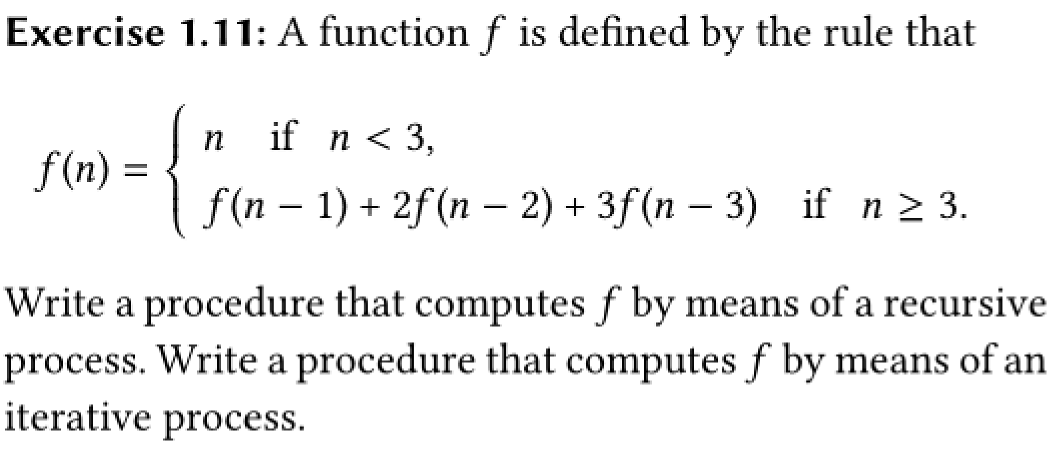
如何理解上述的解法 $f(a,n) = f(a,n-1) + f(a-d,n)$ ，可以从 `DFS` 解决问题角度来理解。

假设有零钱数量 `a=15`，硬币种类数为 `n=5(1， 2， 5， 10 ， 20)`，如果使用 `DFS` 来解决这个问题，也即是，从硬币的种类数来遍历搜索，

```
# 先从1开始遍历搜索
(*1*， 2， 5， 10， 20)
|| # 继续向下搜索，直到和为a时停止搜索，计数加一
(*1*， 2， 5， 10， 20) -> (1， *2*， 5， 10， 20) -> (1， 2， *5*， 10， 20) -> (1， 2，
5， *10*， 20) -> (1， 2， 5， 10， *20*)
||
......
# 当1搜索完成时，开始从下一个2
(1， *2*， 5， 10， 20)
||
(1， *2*， 5， 10， 20) -> (1， 2， *5*， 10， 20) -> (1， 2， 5， *10*， 20) -> (1， 2，
5， 10， *20*)
||
......
# 以此类推
```

从遍历搜索的角度，$f(a, n-1)$ 相当于 $f(15, 4(2, 5, 10, 25))$ 不使用1的所有组合数，$f(a-d, n)$ 相当于 $f(14, 5(1, 2, 5, 10, 20))$ 使用1的所有组合数，所以两种组合为全部的组合数。

recursive ane iterative process



Exercise 1.11: A function f is defined by the rule that

$$f(n) = \begin{cases} n & \text{if } n < 3, \\ f(n-1) + 2f(n-2) + 3f(n-3) & \text{if } n \geq 3. \end{cases}$$

Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

递归方法易得。

迭代的方法，推导如下：

```
f(3) = f(2) + 2f(1) + 3f(0)
f(4) = f(3) + 2f(2) + 3f(1)
f(5) = f(4) + 2f(3) + 3f(2)
....
```

所以可以看出，`f(4)` 与 `f(3)` 相关，有如下的推导

```
a, b, c => a+2b+3c, a, b => ..
```

代码如下：

```
(define (func-iter n)
  (func-calc 2 1 0 0 n)
)

(define (func-calc a b c i n)
  (if (= i n)
      c
      (func-calc (+ a (* 2 b) (* 3 c))
                 a
                 b
                 (+ i 1)
                 n)
  )
)
```

Exercise 1.15

Exercise 1.15: The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin x \approx x$ if x is sufficiently small, and the trigonometric identity

$$\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$$

to reduce the size of the argument of sin. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:

代码为：

```
(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)))))
```

问题为：

a. How many times is the procedure `p` applied when `(sine 12.15)` is evaluated?

b. What is the order of growth in space and number of steps (as a function of *a*) used by the process generated by the `sine` procedure when `(sine a)` is evaluated?

a问题易得，可以通过如下计算求解:

$\frac{12.15}{3} = 4.05, \frac{4.05}{3} = 1.35, \frac{1.35}{3} = 0.45, \frac{0.45}{3} = 0.15, \frac{0.15}{3} = 0.05$

也就是 `5` 次。

b问题，不易得，过程如下：

可以看出 `a` 值会不断被3除，直至其值小于 `0.1` ，由此可以得到 $\frac{a}{3^n} < 0.1$ ，则有:

$$\frac{a}{0.1} < 3^n$$

$$log(\frac{a}{0.1}) < log(3^n)$$

$$log(a) - log(0.1) < nlog(3)$$

$$\frac{log(a) - log(1)}{log(3)} < n$$

从公式可以看出，最后的时间复杂度也就是 $log(a)$。

Testing for Primality

> **Fermat's Little Theorem:** If n is a prime number and a is any positive integer less than n, then a raised to the nth power is congruent to a modulo n.

也就是在 $0 < a < n$ 的前提下，如果 `n` 为素数，则有 $a^n \bmod n = a \bmod n = a$ 。

```scheme
; The Fermat test
(define (exptmod base exp m)
    (cond ((= exp 0) 1)

          ((isEven exp)
           (remainder
            (squre (exptmod base (/ exp 2) m))
            m))
          (else
           (remainder
            (* base (exptmod base (- exp 1) m))
            m))
    )
)

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

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

Exercise 1.26

> **Exercise 1.26:** Louis Reasoner is having great difficulty doing Exercise 1.24. His `fast-prime?` test seems to run more slowly than his `prime?` test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the `expmod` procedure to use an explicit multiplication, rather than calling `square`:
>
> ```scheme
> (define (expmod base exp m)
>   (cond ((= exp 0) 1)
>         ((even? exp)
>          (remainder (* (expmod base (/ exp 2) m)
>                        (expmod base (/ exp 2) m))
>                     m))
>         (else
>          (remainder (* base
>                        (expmod base (- exp 1) m))
>                     m))))
> ```
>
> "I don't see what difference that could make," says Louis. "I do." says Eva. "By writing the procedure like that, you have transformed the $\Theta(\log n)$ process into a $\Theta(n)$ process." Explain.

当 `exp` 为偶数时，上述程序会计算两次 `(expmod base (/ exp 2) m)`，而 `(squre (expmod base (/ exp 2) m))` 只计算一次。

所以复杂度会上升如上写运行。

Miller–Rabin primality test

首先根据 `fermant's little theorem` 可以得到

$$a^n \bmod n = a \bmod n = a$$
$$a^n \equiv a \pmod{n}$$

如果 `a` 不是 `n` 的倍数，这个定理也可以写成更加常用的一种形式

$$a^{n-1} \equiv 1 \pmod{n}$$

假设 `p` 是奇素数，则 $x^2 \equiv 1(\bmod p)$ 的解为

$$x^2 - 1 \equiv 0 \pmod{p}$$

也就是

$$(x+1)(x-1) \bmod p = 0$$

可以看出，要么 $(x+1)(x-1) = 0$ 或者 $(x+1)(x-1)$ 为 `p` 的倍数，又结合费马小定理可以知道 $0<x<p$，故解为 $x = 1, x = p - 1$。

结合上述的到的 $a^{n-1} \equiv 1 \pmod{n}$, 则 $n-1$ 为偶数，如果 $n-1$ 为奇数， `n` 为偶数，则可以直接判断其能否被2整除，所以 $n-1$ 为偶数， 故

$$(a^{\frac{n-1}{2}})^2 - 1 \equiv 0 \pmod{n}$$

也就是, 判断 $a^{\frac{n-1}{2}}$ 的解是否为 `1 or n-1`，不为则不是素数，如果解正确，则要模仿之前的操作，再进行一轮检验，变成判断 $a^{\frac{n-1}{4}}$ ，直到最后变成奇数。

Calculate the definite integral

定积分计算为:

$$\int_a^b f = [f(a + \frac{dx}{2}) + f(a + dx + \frac{dx}{2}) + f(a + 2dx + \frac{dx}{2}) + \ldots]dx$$

可以定义一些高级抽象，共享 一些基础模式。

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b)))
)
```

如求解立方的定积分：

```
(define (integral f a b dx)
  (define (add-dx x)
    (+ x dx)
  )
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx)
)

(define (cube x)
  (* x x x)
)

(integral cube 0 1 0.01)
(integral cube 0 1 0.001)
```

Simpson’s Rule Calculate the definite Integration

辛普森规则来计算定积分更加精确，辛普森规则如下:

$$\int_a^b f = \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n)$$
$$h = \frac{b-a}{n}$$
$$y_k = f(a + kh)$$

`n` 为偶数，增大`n`值能够提高定积分近似值。

compute approximations to PI

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

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.

递归和迭代两种方法:

```
; iterative
(define (product term a next b)
  (define (iter a result)
    result
    (iter (next a) (* result (term a))))
  (iter a 1))
; recursive
(define (product term a next b)
  (if (> a b)
      1
      (* (term a) (product term (next a) next b))))

; compute pi
; gen numberator
(define (number-item i)
  (cond
    [(= i 1) 2]
    [(even? i) (+ i 2)]
    [else (+ i 1)]))

; gen denominator
(define (deno-item i)
  (if (odd? i)
      (+ i 2)
      (+ i 1)))

(define (pi n)
  (* 4 (exact->inexact
        (/ (product number-item
                    1
                    (lambda (i) (+ i 1))
                    n)
           (product deno-item
                    1
                    (lambda (i) (+ i 1))
                    n)))))
```