



函数式程序设计

郭 炜

<http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>



第二讲

表达式求值

Welcome to [DrRacket](#), version 5.92--2014-01-25(-/f) [3m].

Language: **racket**; memory limit: **128 MB**.

> +

#<procedure:+>

> abs

#<procedure:abs>

> (define p 123)

> p

123

> (define (q x) (+ x 1))

> q

#<procedure:q>

表达式求值

特殊形式，如`define`, `and`, `or` 等，不能求值。特殊形式不能被看作过程。

> `define`

. `define: bad syntax in: define`

> `and`

. `and: bad syntax in: and`

>

define 的用法

1. 定义变量

```
(define p 123)
```

```
(define x (* 3 5))
```

2. 定义过程

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

```
(define (sum x y) (+ x y))
```

```
(define (sum-squares x y) (sum (square x) (square y)))
```

define 的用法

2. 定义过程

> (define p 5) ;定义变量

> p

5

> (define (p2) 5) ;定义无参函数 p2

> p2

#<procedure:p2>

> (p2) ;调用过程需要将过程名放入括号

5

>

define 的用法

2. 定义过程

```
(define (func x)  
  (+ x 1) (display "tmp") (newline) (* x 2))
```

```
(func 6)
```

define 的用法

2. 定义过程

```
(define (func x)  
  (+ x 1) (display "tmp") (newline) (* x 2))
```

```
(func 6)
```

输出:

```
tmp  
12
```


内部定义和块结构

牛顿迭代法求平方根

```
(define (square x)      (* x x))  
(define (average a b)   (/ (+ a b) 2))  
  
(define (improve guess x) (average guess (/  
                                         x guess)))  
  
(define (good-enough? guess x)  
  (< (abs (- (square guess) x)) 0.001))  
  
(define (sqrt-iter guess x) ;guess是猜测值  
  (if (good-enough? guess x)  
      guess  
      (sqrt-iter (improve guess x)  
                  x)))  
  
(define (sqrt x) (sqrt-iter 1.0 x))
```

Improve, good-enough?
sqrt-iter 貌似除了 sqrt, 其他过程都不会调用它们

内部定义和块结构

应该允许一个过程里带一些内部定义，使得它们是局部于这一过程的。这种嵌套定义就是“块结构”

```
(define (square x)      (* x x))
(define (average a b)    (/ (+ a b) 2))

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

`improve`, `good-enough?` , `sqrt-iter` 都局部于 `sqrt`, 在 `sqrt` 外不可见

内部定义和块结构

进一步简化，让x作为内部定义中的自由变量：

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

- 信息的局部化，是好的程序的重要特征。
- Java, C++不支持函数嵌套定义。可以用“类”或Lambda表达式实现信息局部化。
- C++程序支持多文件，将函数在一个文件里声明为static也能起到局部化作用

C++如何近似实现函数嵌套定义

```
double Sqrt(double x) {  
    static auto improve = [](double guess, double x) {  
        return average(guess, x/guess);  
    }; // 无static则后面无法capture improve  
    static auto good_enough = [](double guess, double x) {  
        return abs(square(guess)-x) < 0.001 ;  
    };  
    static double (* sqrt_iter)(double, double) =  
        [](double guess, double x) -> double {  
            if( good_enough(guess, x))  
                return guess;  
            else  
                return sqrt_iter(improve(guess, x), x);  
        }; // 递归, 因此无法用auto 定义 sqrt_iter  
    return sqrt_iter(1.0, x);  
}
```

C++如何近实现局部化

- C++程序支持多文件，将函数在一个文件里声明为**static**也能起到局部化作用

例，为**sqrt** 建立一个独立文件:

```
static double sqrt_iter (double guess, double x){...}  
static double improve (double guess, double x){...}  
static int good_enough (double guess, double x){...}  
static double average (double x, double y){...}  
double sqrt (double x) {...}
```

线性递归和迭代

- 求n!的第一种算法:

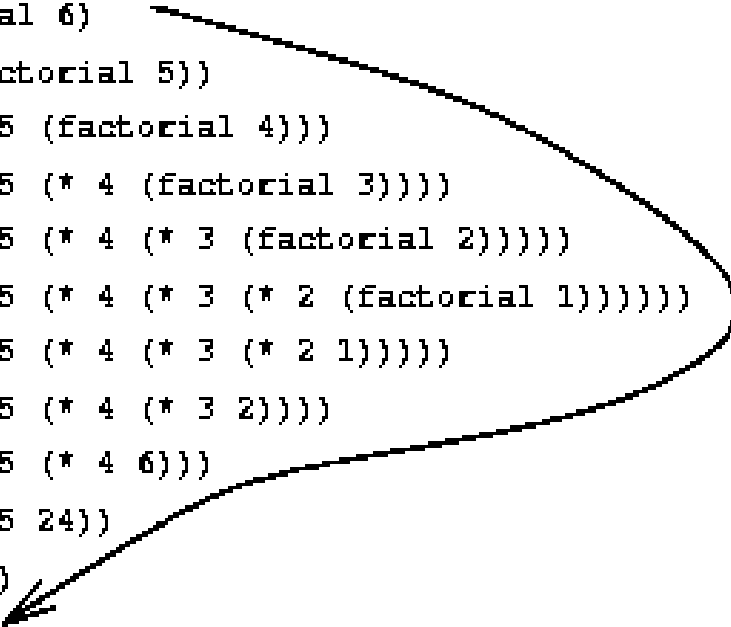
$$n! = n * (n-1)!$$

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

- 线性递归，时间和空间复杂度都为线性

(factorial 6) 计算过程:

```
(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
```



需要在栈中存 6 5 4 3 2 1

线性递归和迭代

- 求n!的第二种算法，反复做：

product = counter · product

counter = counter + 1

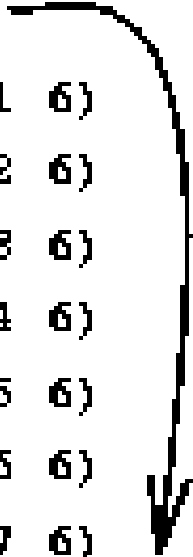
从counter=1 做到counter = n

```
(define (factorial n)
  (define (fact-iter product counter
                    max-count)
    (if (> counter max-count)
        product
        (fact-iter (* counter product)
                    (+ counter 1)
                    max-count)))
  (fact-iter 1 1 n))
```

- 线性迭代，时间复杂度线性，空间复杂度常数

(factorial 6) 计算过程：

```
(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
```



只需要在栈中存三个参数

尾递归

- 命令式语言使用循环描述迭代过程
- Scheme使用“尾递归优化”，可以以递归形式实现迭代的空间复杂度

如果一个函数中**所有**递归调用都出现在函数的末尾，且递归调用是整个函数体中最后执行的语句且它的返回值不属于表达式的一部分时，这个递归调用就是尾递归。

进行尾递归调用时，因为栈中存放的内容已经不再有用，所以可以不必让栈加深，只在原位置进行下一层函数调用即可。

- 一些命令式语言编译器会进行尾递归优化，即用迭代实现尾递归以降低空间复杂度。如何做实验判断编译器有无尾递归优化？

尾递归

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

不是尾递归，因为栈中要存 n

尾递归

- 一些命令式语言编译器会进行尾递归优化，即用迭代实现尾递归以降低空间复杂度。如何做实验判断编译器有无尾递归优化？

尾递归

- 一些命令式语言编译器会进行尾递归优化，即用迭代实现尾递归以降低空间复杂度。如何做实验判断编译器有无尾递归优化？

```
void f(int n)
{
    f(n);
}
int main()
{
    f(20);
    return 0;
}
```

尾递归

- 一些命令式语言编译器会进行尾递归优化，即用迭代实现尾递归以降低空间复杂度。如何做实验判断编译器有无尾递归优化？

```
void f(int n)
{
    f(n);
}
int main()
{
    f(20);
    return 0;
}
```

看这个程序会死循环还是爆栈。

尾递归

- 一些命令式语言编译器会进行尾递归优化，即用迭代实现尾递归以降低空间复杂度。如何做实验判断编译器有无尾递归优化？

观察递归函数参数的地址是否发生变化。

尾递归

- 一些命令式语言编译器会进行尾递归优化，即用迭代实现尾递归以降低空间复杂度。如何做实验判断编译器有无尾递归优化？

观察递归函数参数的地址是否发生变化。

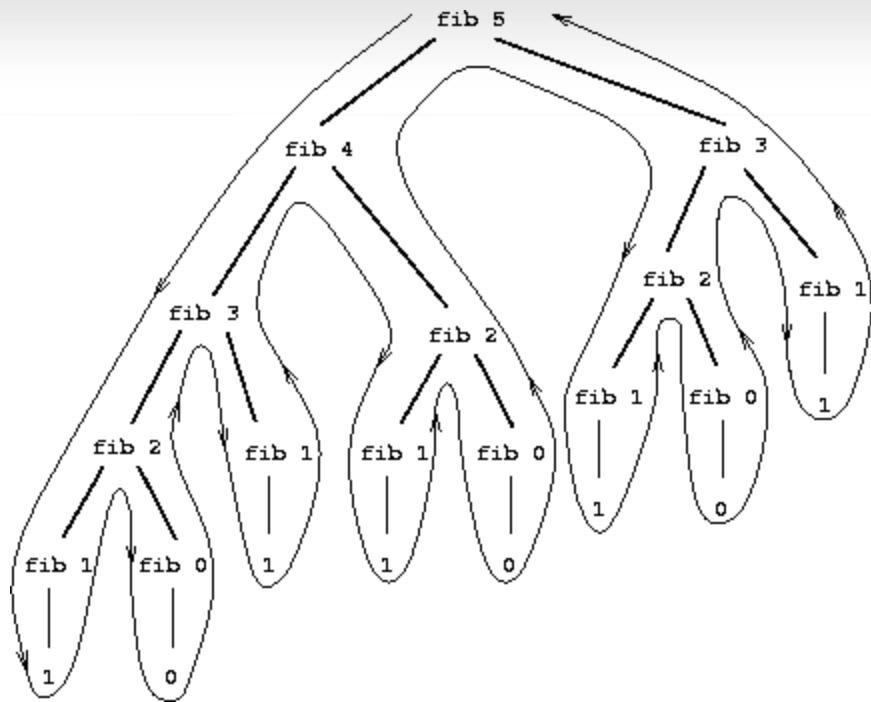
(有的编译器发现要查看参数地址，则就不优化了)

树形递归

●求斐波那契数列第n项

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise} \end{cases}$$

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```



指数级的时间复杂度，因为有重复计算
改进方法：迭代，或动态规划

树形递归

- 迭代法求斐波那契数列第n项，复杂度 $O(n)$

$a \leftarrow a + b$

$b \leftarrow a$

```
(define (fib n)
  (define (fib-iter a b count) ;a > b
    (if (= count 0)
        b
        (fib-iter (+ a b) a (- count 1))))
  (fib-iter 1 0 n))
```


树形递归 – 换零钱方式的统计

- 美元有1分，5分，10分，25分，50分这几种硬币。给定一一个金额，求用硬币凑出该金额的方法数。

树形递归 – 换零钱方式的统计

- 美元有1分，5分，10分，25分，50分这几种硬币。给定一个金额，求用硬币凑出该金额的方法数。

设 $f(n,k)$ 为用前 k 种硬币凑金额 n 的方法数：

$$f(n,k) = f(n,k-1) + f(n-v[k],k) \quad ; v[k] \text{ 是第 } k \text{ 种硬币的币值}$$

边界条件： $n = 0 \Rightarrow 1$

$n < 0 \Rightarrow 0$

$(n \neq 0 \ \&\& \ k == 0) \Rightarrow 0$

树形递归 – 换零钱方式的统计

```
(define (count-change amount) ; (count-change 100) => 292
  (cc amount 5))

(define (cc amount kinds-of-coins) ; 用前kinds-of-coins种币凑amount
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                      (- kinds-of-coins 1))
                  (cc (- amount
                        (first-denomination kinds-of-coins))
                      kinds-of-coins))))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

思考题，倒转各种硬币的排列顺序会怎么样：
程序还正确吗？
效率会改变吗？

树形递归 – 换零钱方式的统计

用数组解决:

```
(define first-denomination (vector 0 1 5 10 25 50))  
(define (count-change amount) ;(count-change 100) => 292  
  (cc amount 5))  
(define (cc amount kinds-of-coins)  
  (cond ((= amount 0) 1)  
        ((or (< amount 0) (= kinds-of-coins 0)) 0)  
        (else (+ (cc amount  
                      (- kinds-of-coins 1))  
                  (cc (- amount  
                        (vector-ref first-denomination  
                                kinds-of-coins))  
                        kinds-of-coins))))))
```

重复计算导致指数复杂度。用动态规划降低时间复杂度到 $O(n*k)$

实例：快速幂

要求 b^n :

- 方法一:

```
(define (expt b n) ;线性时间和空间
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

实例：快速幂

要求 b^n :

- 方法二（迭代）：

```
(define (expt b n) ;尾递归，线性时间，常数空间  
  (expt-iter b n 1))
```

```
(define (expt-iter b counter product)  
  (if (= counter 0)  
      product  
      (expt-iter b  
                  (- counter 1)  
                  (* b product)))))
```

实例：快速幂

实际上，求 b^8 ，只需做3次乘法，而非7次

$$b^2 = b \cdot b$$

$$b^4 = b^2 \cdot b^2$$

$$b^8 = b^4 \cdot b^4$$

对一般整数 n

$$n \text{ 为偶数时 } b^n = (b^{n/2})^2$$

$$n \text{ 为奇数时 } b^n = b \cdot b^{n-1} \quad ; n-1 \text{ 是偶数}$$

实例：快速幂

n 为偶数时 $b^n = (b^{n/2})^2$

n 为奇数时 $b^n = b \cdot b^{n-1}$; $n-1$ 是偶数

思考题，求 b^n ，准确的乘法次数是多少次？

方法三、快速幂算法：

```
(define (fast-expt b n) ;时间复杂度  $O(\log n)$ 
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

```
(define (even? n)
  (= (remainder n 2) 0)) ;remainder是库函数
```

用? 作为谓词过程名(返回值为true或false) 的最后字符，是Scheme 的编程习惯。

实例：快速幂

n为偶数时 $b^n = (b^{(n/2)})^2$

n为奇数时 $b^n = b \cdot b^{(n-1)}$;n-1 是偶数

思考题，求 b^n ，准确的乘法次数是多少次？

方法三、快速幂算法：

(define (fast-expt b n) ;时间复杂度 $O(\log n)$

(cond ((= n 0) 1)

((even? n) (square (fast-expt b (/ n 2)))))

(else (* b (fast-expt b (- n 1))))))

(define (even? n)

(= (remainder n 2) 0)) ;remainder是库函数

$\lceil \log_2(n) \rceil + (n \text{ 的二进制形式中 } 1 \text{ 的个数})$ ，
取整为去尾取整
 $\leq 2\log_2(n)$

用? 作为谓词过程名(返回值为true或false) 的最后字符，是Scheme 的编程习惯。

实例：最大公约数

欧几里得算法：

$$\text{GCD}(a, b) = \text{GCD}(b, r) \quad r = a \bmod b$$

$$\begin{aligned}\text{GCD}(206, 40) &= \text{GCD}(40, 6) \\ &= \text{GCD}(6, 4) \\ &= \text{GCD}(4, 2) \\ &= \text{GCD}(2, 0) \\ &= 2\end{aligned}$$

```
(define (gcd a b) ;复杂度O(log(n))  
  (if (= b 0)  
      a  
      (gcd b (remainder a b))))
```

实例：素数检测

$O(\sqrt{n})$ 的笨办法：从2一直试到 \sqrt{n}

```
(define (smallest-divisor n) ;找n的最小因子
  (find-divisor n 2)) ;从2开始找n的最小因子
(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) ;a是否是b的因子
  (= (remainder b a) 0))

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

实例：素数检测

费马小定理：若 n 是素数， a 是任小于 n 的正整数，则 a 的 n 次方与 a 模 n 同余。

n 不是素数时多数 $a < n$ 都不满足上述关系

素数检测的概率算法（非确定算法）：

随机取一个 $a < n$ ，求 a^n 除以 n 的余数

如果结果不是 a ，那么 n 不是素数

否则重复上述过程

n 通过检查的次数越多，是素数的可能性就越大

通过费马检查的数，有很大概率是素数，但不一定是素数。

素数的费马检测算法程序

```
(define (fermat-test n) ;判断n是否能够通过一次素数检测
  (define (try-it a) ;a^n mod n 是否等于a
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1))))) ;(random k)产生0至k-1的随机数
                                      ;random是库函数

(define (fast-prime? n times) ;通过times次检测判断n是否是素数
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))
```

素数的费马检测算法程序

```
(define (square x) (* x x))
(define (expmod base exp m) ;求 base^exp mod m
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m))
                     m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                     m)))))
```

依据: $(a * b) \bmod c = ((a \bmod c) * (b \bmod c)) \bmod c$

目的: base^{exp} 可能太大, 导致计算时间增加。对其他语言来说, 可能导致溢出

素数的费马检测算法程序

- 测一次通过，素数的概率大于 $1/2$

- 测二次通过，素数的概率大于 $3/4$

.....

- 但通过费马检测 (不论试多少次) 的数，不一定是素数。能通过费马检测的奇合数，称为Carmichael数。

- 100 000 000以内有255个Carmichael数，如561,1105,1729.....

- `(fast-prime? 561 1000000)` \Rightarrow `#t`

高阶过程

以过程作为参数或返回值的过程，叫高阶过程

C++的STL中可以找到高阶过程的例子：

```
void PrintSquare(int v)  {  
    cout << v*v << " ";  
}  
int main() {  
    vector<int> vt {1,2,3,4,5};  
    for_each(vt.begin(), vt.end(), PrintSquare);  
    return 0;  
} //输出: 1 4 9 16 25
```


高阶过程

C++的lambda表达式和函数指针都可以用于高阶过程：

```
int main() {  
    vector<int> vt {1,2,3,4,5};  
    for_each(vt.begin(), vt.end(),  
             [](int x) { cout << x * x << " "; });  
    return 0;  
} //输出: 1 4 9 16 25
```

过程作为参数

考虑三个过程:

```
(define (sum-integers a b) ;a+ ...+b
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b)))))
```

共同模式: 从某参数**a**到参数**b**, 按一定步长, 对依赖于**a**的一些项求和

```
(define (sum-cubes a b) ;a3 + ...+b3
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b)))))
```

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b)))))
```

(pi-sum 1 21) =>
收敛到 $\pi/8$

$$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \dots$$

过程作为参数

共同模式:

$$\sum_{n=a}^b f(n) = f(a) + \cdots + f(b)$$

描述:

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

过程作为参数

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

```
(define (inc n) (+ n 1))
(define (sum-cubes a b)
  (define (cube x) (* x x x))
  (sum cube a inc b))
```

```
(define (sum-integers a b)
  (define (identity x) x)
  (sum identity a inc b))
```

```
(define (pi-sum a b)
  (define (pi-next x) (+ x 4))
  (define (pi-term x) (/ 1.0
                          (* x (+ x 2))))
  (sum pi-term a pi-next b) )
```

sum的C++实现方式:

```
template <class T1,class T2>
double sum(T1 term,T2 next,int a,int b) {
    if( a > b)
        return 0;
    return term(a) + sum(term,next,next(a),b);
}

int cube(int x) {return x * x * x;}
int inc(int x) {return x + 1;}
int sum_cubes(int a,int b) {
    return sum( cube,inc,a,b);
}

double pi_sum(int a,int b){
    return sum( [](int x) {return 1.0/(x*(x+2));},
               [](int x) {return x + 4; },
               a,b);
}
```

以过程作为参数：数值积分

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \cdots \right] dx \quad dx \text{很小}$$

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

```
(integral cube 0 1 0.01)
=> 0.24998750000000042
(integral cube 0 1 0.001)
=> 0.2499998750000001
```

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

x^3 在 $[0, 1]$ 积分的精确值是 $1/4$

用lambda构造匿名过程

```
> (lambda (x y) (sqrt (+ (* x x) (* y y))))  
#<procedure>
```

```
>
```

上面创建了一个匿名过程，可以求 (x, y) 到原点的距离

用lambda重写pi-sum:

```
(define (pi-sum a b)  
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))  
    a  
    (lambda (x) (+ x 4))  
    b))
```

用lambda构造匿名过程

用lambda重写数值积分integral:

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

原来的:

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


lambda表达式

lambda 表达式的一般形式:

(lambda (<formal-parameters>) <body>)

lambda 是特殊形式，其参数不求值

(define (<name> <formals>) <body>) 等价于:
(define <name> (lambda (<formals>) <body>))

(define (plus4 x) (+ x 4)) 等价于
(define plus4 (lambda (x) (+ x 4)))

前一写法只是为了方便

lambda表达式

```
(define p (lambda (x) (+ x 1)))  
p
```

lambda表达式

```
(define p (lambda (x) (+ x 1)))  
p      ;=> #<procedure:p>  
(p 2)
```

lambda表达式

```
(define p (lambda (x) (+ x 1)))
```

```
p      ;=> #<procedure:p>
```

```
(p 2)  ;=> 3
```

```
(define (p2) (lambda (x) (+ x 3)))
```

```
p2
```

lambda表达式

```
(define p (lambda (x) (+ x 1)))
```

```
p  ;=> #<procedure:p>
```

```
(p 2)  ;=> 3
```

```
(define (p2) (lambda (x) (+ x 3)))
```

```
p2 ; => <procedure:p2>
```

```
(p2)
```

lambda表达式

```
(define p (lambda (x) (+ x 1)))  
p      ;=> #<procedure:p>  
(p 2)  ;=> 3
```

```
(define (p2) (lambda (x) (+ x 3)))  
p2 ; => <procedure:p2>  
(p2) ;=> <procedure>  
((p2) 2)
```

lambda表达式

```
(define p (lambda (x) (+ x 1)))  
p      ;=> #<procedure:p>  
(p 2)  ;=> 3
```

```
(define (p2) (lambda (x) (+ x 3)))  
p2 ; => <procedure:p2>  
(p2) ;=> <procedure>  
((p2) 2) ;=> 5
```

p2是一个函数，调用p2的结果，是返回一个函数

lambda表达式

```
(define ((p3)) (lambda (x) (+ x 3)))
```

```
p3  
(p3)  
((p3))  
(((p3)) 4)
```


lambda表达式

```
(define ((p3)) (lambda (x) (+ x 3)))
```

```
p3  
(p3)  
((p3))  
(( (p3)) 4)
```

p3是一个函数，调用p3的结果，是一个函数 **k**，调用 **k**的结果，是返回一个函数 **y**，
y就是 (lambda (x) (+ x 3))

输出：

```
#<procedure:p3>  
#<procedure>  
#<procedure>  
7
```

lambda表达式

lambda表达式可以被求值，被求值会生成一个过程

对lambda表达式求值是生成过程的唯一方式

lambda表达式可以作为组合式的运算符

((lambda (x y z) (+ x y (square z))) 1 2 3)

>12

lambda过程作为返回值

```
(define (f a b)
  (lambda (x y)
    (+ (* a x)
       (* b y)))))
```

```
((f 1 2) 3 4) ;=>?
```

lambda过程作为返回值

```
(define (f a b) ;生成一个过程，这个过程能求  $ax+by$   
  (lambda (x y)  
    (+ (* a x)  
      (* b y))))
```

```
((f 1 2) 3 4) ;=> 11  
; (f 1 2) 生成一个过程，这个过程能求  $x+2y$ 
```

lambda过程作为返回值

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

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

```
(define (odd-average f)  
  (lambda (x) (average x (f x))))
```

```
(define p (odd-average square))
```

```
(p 4)    ;=> ?
```

lambda过程作为返回值

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

```
(define (odd-average f); 生成一个过程k, k接受参数x, 能求x和f(x) 平均值  
  (lambda (x) (average x (f x))))
```

```
(define p (odd-average square))
```

```
(p 4)    ;=> 10
```

lambda过程作为返回值

高阶函数:

```
(define (square x)  (* x x))  
(define (inc x)    (+ x 1))
```

```
(define (combine f g)  
  (lambda (x) (f (+ (f x) (g x))))))
```

(combine f g) 返回函数 k , $k(x) = f(f(x)+g(x))$

```
((combine square inc) 3)      输出169
```

let表达式

考虑函数：

$$f(x, y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

$(1+xy)$ 和 $(1-y)$ 多次出现。若重复计算，则浪费时间。

多次出现也导致程序不易修改，比如想把 $(1-y)$ 全改成 $(1+y)$ ？

希望引进临时变量保存重复使用的中间结果：

$$\begin{aligned} a &= (1+xy) & b &= (1-y) \\ f(x, y) &= xa^2 + yb + ab \end{aligned}$$

let表达式

实现方法一：

引入辅助函数 `f-helper`

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y)) ;a= 1+xy
            (- 1 y))) ;b = 1-y
```

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

let表达式

实现方法二:

用`labmda`替代辅助函数 `f-helper`

```
(define (f x y)
  ((lambda (a b)
    (+ (* x (square a))
      (* y b)
      (* a b)))
    (+ 1 (* x y)) ;a= 1+xy
    (- 1 y))) ;b = 1-y
```

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

let表达式

实现方法三:

用let 定义局部变量 a b

```
(define (f x y)
  (let ((a (+ 1 (* x y))) ;a = (+ 1 (* x y))
        (b (- 1 y))) ;b = (- 1 y)
    (+ (* x (square a))
      (* y b)
      (* a b))))
```

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

let表达式

let表达式的一般形式:

```
(let ( (<var1> <exp1>)  
      (<var2> <exp2>)
```

; 前面是一组变量/值表达式对, 表示建立约束关系

```
      (<varn> <expn>)) ; var_i的值就是 exp_i  
  <body>)
```

let $\langle var_1 \rangle$ have the value $\langle exp_1 \rangle$ and $\langle var_2 \rangle$ have the value $\langle exp_2 \rangle$ and $\langle var_n \rangle$ have the value $\langle exp_n \rangle$ in $\langle body \rangle$

$var_1, var_2 \dots var_n$ 作用域仅限于 let 表达式内部

$\langle body \rangle$ 可以为多个表达式并列, 最后一个表达式的值就是 let 表达式的值

let表达式

let表达式是lambda的一种应用的语法糖，等价于：

```
( (lambda (<var1> ...<varn>)
  <body>)
  <exp1>
  ...
  <expn>)
```

即以exp1,exp2... expn为实参数，调用定义的匿名函数。函数形参：var1,var2...

let表达式

`var1, var2 ... varn`作用域仅限于 `let` 表达式内部，这样使人能在尽可能接近使用处的地方建立局部约束变量。

```
(define x 5)
(+ (let ((x 3))
    (+ x (* x 10))))
x)
```

`let`表达式外部的`x`, 其值为5, 内部的`x`, 其值为3

let表达式

`let`中用来约束变量的 `exp_i` ，其值是在`let`之外计算的，即`exp_i`中不可包含`let`中定义的变量。

```
(define k 5)
(+ (let ((x 3)
        (y (+ x 1))) ;err x: unbound identifier in module in: x
    (+ x (* x 10)))
  k)
```

let表达式

let中用来约束变量的 **exp_i** ，其值是在**let**之外计算的，即**exp_i**中不可包含**let**中定义的变量。

```
(define x 2)
(let ((x 3)
      (y (+ x 2)))
  (* x y))
```

=>?

请考虑与上面**let** 对应的**lambda** 表达式

let表达式

`let`中用来约束变量的 `exp_i` ，其值是在`let`之外计算的，即`exp_i`中不可包含`let`中定义的变量。

```
(define x 2)
(let ((x 3)
      (y (+ x 2))) ; 此处x为外部的x, 值为2
  (* x y))
```

=> 12

等价于:

```
((lambda (x y)
  (* x y))
  3 (+ x 2))
```

A+B problem

描述

输入两个数，输出它们的和

输入

多组数据。每组一行，两个数

输出

每行输出一个和

样例输入

```
1 2 2.4 3.5
```

样例输出

```
3 5.9
```

A+B problem

```
#lang racket
(define (myloop)
  (let ((a (read)) ;read是库函数每次读入一个完整的scheme表达式
        (b (read)))
    (if (eq? a eof)
        (void)
        (begin (display (+ a b)) (newline) (myloop)))))

(myloop)
```

`;displayln` 可以用来输出并换行

实例：二分法求方程的根(函数零点)

问题：在某区间里找方程的根：

区间 $[a, b]$ ，若 f 是连续函数且 $f(a) < 0 < f(b)$ ，则 $[a, b]$ 中必有 f 的根（中值定理）

折半法：取区间中点 x 计算 $f(x)$

如果 $f(x)$ 是根（在一定误差的意义下），计算结束

否则根据 $f(x)$ 的正负将区间缩短一半

在缩短的区间里继续使用折半法

此操作做一次，查找区间长度减半

假设初始区间的长度为 L ，容许误差为 T (求出的根和真实根的差)

所需计算步数为 $O(\log(L/T))$ 是对数时间算法

实例：二分法求方程的根(函数零点)

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negative? test-value)
                 (search f midpoint pos-point))
                (else midpoint))))))
```

实例：二分法求方程的根(函数零点)

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "Values are not of opposite sign" a b))))))

(half-interval-method
 (lambda (x) (- (cube x) (* 2 x) 3)) ;  $x^3 - 2x - 3 = 0$ 
 1.0
 2.0) ; => 1.89306640625
```

实例：函数的不动点

定义：对于函数 $f(x)$,若 x' 满足 $f(x') = x'$,则称 x' 为 $f(x)$ 的不动点。

有的函数有不动点，有的没有。

求 $f(x)$ 的不动点，就是解方程 $f(x) = x$

求不动点思路：从某个猜测的初始值 x_0 出发，生成以下值的序列，直到序列里相邻的两项差别很小。

$f(x_0), f(f(x_0)), f(f(f(x_0))) \dots$

直到：

$\text{abs}(f' - f(f')) < \text{eps}$, 则 f' 为不动点

此算法是否收敛？

实例：函数的不动点

定义：对于函数 $f(x)$ ，若 x' 满足 $f(x') = x'$ ，则称 x' 为 $f(x)$ 的不动点。

有的函数有不动点，有的没有。

求 $f(x)$ 的不动点，就是解方程 $f(x) = x$

求不动点思路：从某个猜测的初始值 x_0 出发，生成以下值的序列，直到序列里相邻的两项差别很小。

$f(x_0), f(f(x_0)), f(f(f(x_0))) \dots$

直到：

$\text{abs}(f' - f(f')) < \text{eps}$ ，则 f' 为不动点

此算法是否收敛？ 不一定

实例：函数的不动点

```
(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 cos 1.0)` ;求 $\cos(x)$ 不动点,1.0做初值

`=> 0.7390822985224023`

`(fixed-point (lambda (y) (+ (sin y) (cos y))) 1.0)` ; $\sin(x) + \cos(x)$ 不动点

`=> 1.2587315962971173`

实例：函数的不动点

有些函数存在不动点，但用上述算法无法求得不动点（不收敛）

x 的平方根可看作 $f(y) = x/y$ （ y 是自变量）的不动点

求平方根过程：

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y)) 1.0 ))
```

它一般不终止，因为：

$y_3 = x/y_2 = x/(x/y_1) = y_1$ 函数值在两个值之间振荡

实例：函数的不动点

控制振荡的一种方法是消减变化的剧烈程度。因为问题的答案必定在 y_i 和 x/y_i 两值之间，可考虑用 y_i 和 x/y_i 的平均值作为下一猜测值(平均阻尼技术)

```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
    1.0) )
```

即求 $f(y) = (1/2)(y+x/y)$ 的不动点

$f(y) = x/y$ 和 $f(y) = (1/2)(y+x/y)$ 有相同的不动点。后者称为前者的“平均阻尼函数”。

$$y = x/y \Leftrightarrow y = (1/2)(y+x/y)$$

试验说明计算收敛，能达到不动点（得到平方根）

过程作为返回值

$f(x)$ 的平均阻尼函数是: $g(x) = (x+f(x))/2$

$f(x)$ 和 $g(x)$ 具有相同的不动点

写出求 f 的平均阻尼函数的过程(参数为 f , 返回值为 f 的平均阻尼函数):

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

改造sqrt:

```
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

; sqrt x就是 $f(y)=x/y$ 的不动点

average-damp的C++实现

```
template <class T>
struct AverageDampObj {
    T f;
    double operator() (double x)
    { return (f(x)+x)/2; }
    AverageDampObj(T f_):f(f_) { }
};

template <class T>
auto AverageDamp( T f) {
    return AverageDampObj<T>(f);
};

double f(double x) { return 12/x; }
```

```
int main()
{
    auto k = AverageDamp(f);
    cout << sqrt(12) << endl;
    cout << f(sqrt(12)) << endl;
    cout << k(sqrt(12)) << endl;
}
```

输出:
3.4641
3.4641
3.4641

;函数 $f(x)=12/x$ 的不动点是 $\text{sqrt}(12)$

牛顿法求根

牛顿法： $g(x) = 0$ 的解是下面函数的不动点：

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

要用这个公式，需要能求出给定函数 g 的导函数
 $Dg(x)$

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}$$

dx 为很小量，比如 0.00001

牛顿法求根

$$Dg(r) = \frac{g(r + dr) - g(r)}{dr}$$

生成“导函数”的函数**deriv**:

```
(define dx 0.00001)
```

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

生成的函数是 **g** 的导函数（数值上）

```
(define (cube x) (* x x x))
((deriv cube) 5)
```

; \Rightarrow 75.00014999664018

牛顿法求根

牛顿法求 $g(x)$ 的根，就是求 $f(x)$ 的不动点：

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

```
(define (newton-transform g) ; g(x) -> f(x)
  (lambda (x)
    (- x (/ (g x) ((deriv g) x)))))
(define (newtons-method g guess) ;求 g(x) = 0的根
  (fixed-point (newton-transform g) guess))
```

```
(define (sqrt x)
  (newtons-method (lambda (y) (- (square y) x))
    1.0))
```

; x 的平方根是 $f(y) = 0$ 的解，其中 $f(y) = y^2 - x$

抽象和一等过程

求 $\text{sqrt}(x)$ 可以通过求某个函数 $t(y)$ 的不动点来完成。假设 $t(y)$ 又是某个函数 $g(y)$ 的变形，则有两种做法：

1. 某 $g(y)$ 的不动点就是 $\text{sqrt}(x)$ ，直接求 $g(y)$ 的不动点不收敛，所以求 $g(y)$ 的平均阻尼函数 $t(y)$ 的不动点

```
(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (/ x y)) ;  $g(y) = x/y$ 
    average-damp ; 变形方式
    1.0))
```

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess)) ; 求g的变形的不动点
```

抽象和一等过程

求 $\text{sqrt}(x)$ 可以通过求某个函数 $t(y)$ 的不动点来完成。假设 $t(y)$ 又是某个函数 $g(y)$ 的变形，则有两种做法：

2. 某 $g(y)=0$ 的根就是 $\text{sqrt}(x)$ ，用牛顿法求 $g(y)=0$ 的根，就是求 $t(y)$ 的不动点。 $t(y)$ 是 $g(y)$ 的牛顿法变形。

```
(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (- (square y) x)) ;  $g(y)=y^2-x$ 
    newton-transform ; 变形方式
    1.0))
```

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess)) ; 求g的变形的不动点
```

抽象和一等过程

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

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

抽象和一等过程

使用限制最少的元素称为语言中的“一等”元素，是语言的“一等公民”，具有最高特权（最普遍的可用性）。常见的包括：

- 可以用变量命名（在常规语言里，可存入变量，取出使用）
- 可以作为参数传给过程
- 可以由过程作为结果返回
- 可以放入各种数据结构
- 可以在运行中动态地构造

在Scheme（及其他Lisp 方言）里，**过程(函数)**具有完全一等地位。