# I. 构造过程抽象(2)

过程抽象的进一步问题，重点：

- 分析过程（静态，描述）产生的计算进程（动态，行为）
- 计算进程的类型：线性递归，线性迭代和树形递归
- 计算的代价
- 高阶过程：以过程为参数和/或返回值
- lambda 表达式，let 表达式
- 过程作为解决问题的通用方法
- 语言里的一等公民 (first-class object)



## 过程与其产生的计算

- 要真正理解程序设计，仅学会使用语言功能把程序写出来还不够
  - 完成同一工作有多种不同方式，应如何选择？为什么？
- 要成为程序设计专家，必须能理解所写程序蕴涵的计算，理解一个过程（procedure）产生什么样的计算进程（process）
  - 过程（描述）可看作一个模式，它描述了一个计算的演化进程，说明演化的方式，对一组适当参数确定了一个具体计算进程（一个实例，一系列具体步骤）
  - 完成同一件工作，完全可能写出多个大不相同的过程
  - 完成同一工作的两个不同过程导致的计算进程也可能大不相同
- 下面通过例子讨论一些简单过程产生的计算进程的“形状”，观察其中各种资源的消耗情况（主要是时间和空间）
- 从这里得到的认识可供写其他程序时参考



## 线性递归和迭代

- 考虑阶乘计算。一种看法（递归）：

  n! = n • (n-1) • (n-2) • … • 2 • 1 = n • [(n-1) • … • 2 • 1] = n • (n-1)!

- 相应的过程定义：

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

- 采用代换模型推导

  由 (factorial 6) 得到

```
(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
```



## 线性递归和迭代

- 另一观点：n! 是从 1 开始逐个乘各自然数，乘到 n 就得到了阶乘值

  product ← counter · product
  counter ← counter + 1

- 按这种观点写出程序：

```
(define (factorial n)
  (fact-iter 1 1 n))

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

对应计算进程

```
(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
```

## 线性递归和迭代

- 对比两个计算进程：

```
(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
```

先展开后收缩：展开过程中积累一系列计算，收缩就是完成这些计算

解释器需要维护待执行计算的轨迹，轨迹长度等于后续计算的次数

积累长度是线性的，计算步骤序列的长度也是线性的，称为线性递归进程

```
(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
```

无展开或收缩，计算立即进行

轨迹的信息量为常量，只需维护几个变量的当前值

计算序列的长度为线性的。具有这种性态的计算进程称为线性迭代进程



## 线性递归和迭代：分析

- 迭代进程中，计算的所有信息都在几个变量里。即使计算中断，只要有这些变量的当前值，就可以恢复并继续计算
- 在线性递归进程中，相关变量的信息不足以反映计算进程的情况
  - 解释器需要在其他地方保存一些"隐含"信息
  - 这种信息的量将随着计算进程的长度线性增长
- 注意区分"递归计算进程"和"用递归方式定义的过程"
  - 用递归方式定义过程说的是程序的写法，定义一个过程的代码里调用了这个过程本身
  - 递归计算进程，说的是计算中的情况和执行行为，反映计算中需要维持的信息的情况
- 常规语言用专门循环结构（for, while等）描述迭代计算。Scheme 采用尾递归（或称尾递归优化），可用递归方式描述迭代计算
- 练习1.10介绍的Ackermann函数非常有名，其值增长极快



## 树形递归

- 另一常见计算模式是树形递归，典型例子是Fibonacci数的计算

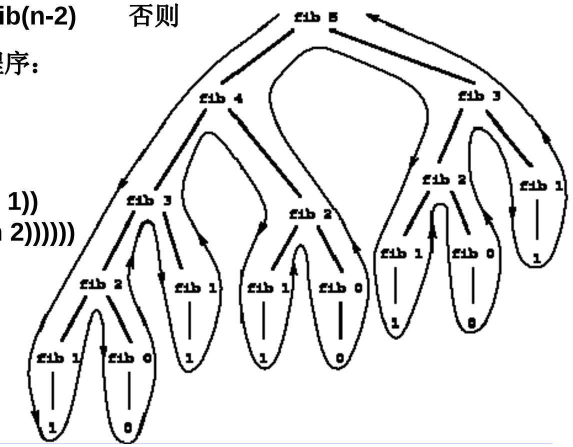
  Fib(n) = 0 若 n = 0
  = 1 若 n = 1
  = Fib(n-1)+Fib(n-2) 否则

- 根据定义直接写出的程序：

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

- (fib 5) 产生的计算

  进程图示：






## 树形递归

- 已知Fibonacci数 Fib(n) 的增长与 n 成指数关系（练习1.13），可知 fib(n) 的计算量增长与 n 的增长成指数关系。很糟糕
- 考虑Fibonacci数的另一算法：取变量 a 和 b，将它们初始化为Fib(0)和Fib(1)的值，而后反复同时执行更新操作：

  a ← a + b
  b ← a

- 写出过程定义：

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

形成的计算进程是一个线性迭代

## 树形递归：换零钱的不同方式

- 现在考虑一个用递归任意描述，不易用循环描述的例子
- 问题：人民币硬币有1元，5角，1角，5分，2分和1分。给了一定的人民币，问有多少种不同方式将它换成硬币？
- 用递归过程描述很自然。先把硬币按某种方式排列，总数为a 的币值换为硬币的不同方式等于：
  - 将 a 换为不用第一种硬币的方式，再加上
  - 用一个第一种硬币（设币值为 b）后将 a-b 换成各种硬币的方式
- 基础情况：
  - a = 0，计1种方式
  - a < 0，计0种方式，因为不合法
  - 货币种类 n = 0，计0种方式，因为已无货币可用
- 很容易把这些考虑直接翻译为递归定义的过程



## 树形递归：换零钱的不同方式

- 过程定义（只算不同换法的数目，并不给出换的方式）

```
(define (count-change amount)  (cc amount 6))

(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
                        (first-denomination kinds-of-coins))
                     kinds-of-coins)))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 2)
        ((= kinds-of-coins 3) 5)
        ((= kinds-of-coins 4) 10)
        ((= kinds-of-coins 5) 50)
        ((= kinds-of-coins 6) 100)))
```

请思考，翻转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

- 描述递归进程的过程的价值：
  - 是某些问题的自然表示，如一些复杂数据结构操作（如树遍历）
  - 编写更简单，容易确认它与原问题的关系
  - 做出对应复杂递归进程的迭代进程的过程，常需要付出很多智力
- 换零钱不同方式，用递归过程描述很自然，它蕴涵一个树形递归进程
  - 写出解决这个问题的迭代不太容易，大家自己做一做
- 通常递归描述可能比较清晰简单，但它有可能实现了一种代价很高的计算过程。而高效的迭代过程可能很难写。人们一直在研究：
  - 能不能自动地从清晰易写的程序生成出高效的程序？
  - 如果不能一般性地解决这个问题，是不是对一些有价值的问题类，或者在特定的描述方式下，可以有解决的办法？
- 这一问题在计算机科学技术中处处可见，永远值得研究



## 增长的阶

- 算法和数据结构课程讨论了计算代价的问题，其中最主要想法
  - 在抽象意义上考虑计算的代价（增长的阶）
  - 考虑计算中的各种资源消耗如何随着问题规模的增长而增长
- 这方面问题不再讨论，书中用 Θ(f(n)) 表示增长的阶是 f(n)，我们下面用 O(f(n)) 表示上界（不要求精确下界），但总考虑尽可能紧的上界
- 应记得：

  O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3) < ... < O(2^n) <

  常量　对数　线性　平方　立方　指数
- 下面看两个例子

## 实例：求幂

- 求 b^n 最直接方式是利用下面递归定义：

  b^n = b • b^(n-1)　　　　b^0 = 1

- 直接写出的程序需要线性时间和线性空间（线性递归计算）：

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

- 不难改为实现线性迭代的过程（仿照前面阶乘程序）

```
(define (expt b n) (expt-iter b n 1))

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

  需要线性时间和常量空间（O(1) 空间）



## 实例：求幂

- 用逐次乘的方式求 b^8 要 7 次乘法，实际上可以只做 3 次

  b^2 = b • b　　　b^4 = b^2 • b^2　　　b^8 = b^4 • b^4

- 对一般整数 n，有

  n为偶数时　　b^n = (b^(n/2))^2

  n为奇数时　　b^n = b • b^(n-1)　　请注意，n-1 是偶数

- 根据这一认识写的过程

```
(define (fast-expt b 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))
```

  用 ? 作为谓词函数名最后字符是 Scheme 的编程习惯

  这一过程求幂只需 O(log n) 次乘法，是重大改进



## 实例：素数检测

- 判断整数 n 是否素数。下面给出两种方法，一个复杂性是 O(sqrt n)，另一个概率算法的复杂性是 O(log n)
- 找因子的直接方法是用顺序的整数去除。下面过程找出最小因子：

```
(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))
```

- 素数就是“大于 2 的最小因子就是其本身”的整数：

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

  n 非素数就一定有不大于其平方根的因子。要检查 O(sqrt n) 个整数



## 实例：素数的费马检查

- 基础是费马小定理：若*n*是素数，*a*是任一小于*n*的正整数，那么*a*的*n*次方与*a*模*n*同余。（两个数模*n*同余：它们除以*n*余数相同。数*a*除以*n*的余数称为*a*取模*n*的余数，简称*a*取模*n* ）
- n 非素数时多数 a < n 都不满足上述关系。这样就得到一个“算法”
  - 随机取一个 a < n，求 a^n 取模 n 的余数
  - 如果结果不是 a 则 n 不是素数，否则重复这一过程
- n 通过检查的次数越多，是素数的可能性就越大
- 实现这一算法需要一个计算自然数的幂取模的过程：

```
(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))))
```

## 实例：素数的费马检查

- 执行费马检查需要随机选取 1 到 n-1 之间的数，过程：

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

  random 取得随机数，(random (- n 1)) 得到 0 到 n-2 间的随机数

- “判断”是否为素数需要反复做费马检查。可以把次数作为参数：

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

  在被检查的数通过了 times 次检查后返回真，否则返回假



## 概率算法

- 上述算法只有概率意义上的正确性，随着检查次数增加，通过检查的数是素数的概率越来越大
- 一点说明：费马小定理只说明素数能通过费马检查，并没说通过检查的都是素数。确实存在不是素数的数能通过费马检查
- 人们已找到了别的检查方法，能保证通过检查的都是素数
- 结果只有概率意义的算法称为概率算法，概率算法已发展成了一个重要研究领域，有许多重要应用。实际中常常只需要概率性的保证



## 高阶过程

- 过程是一类抽象，描述对一些数据的某种复合操作。求立方过程：

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

- 完全可以不写过程，总直接用系统操作写组合式：

```
(* 3 3 3)
(* x x x)
(* y y y)
```

- 这就是只在系统操作的层面上工作，不能提高描述层次。虽然能计算立方，但程序里没有立方的概念。应该为程序里的公共计算模式命名，建立概念。过程抽象能起这种作用
- 只能以数作为参数也限制了建立抽象的能力。有些计算模式能用于多种不同过程，为这类计算模式建立抽象就需要以过程作为参数或返回值
- 以过程作为参数或返回值的，操作过程的过程称为高阶过程。下面讨论如何用高阶过程作为抽象的工具，提高语言的表达能力



## 以过程作为参数

- 考虑下面几个过程：

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

$$a + \cdots + b$$

```
(define (sum-cubes a b)
 (if (> a b)
     0
     (+ (cube a) (sum-cubes (+ a 1) b))))
```

$$a^3 + \cdots + b^3$$

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

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

虽然细节不同，但它们都是从参数 a 到参数 b，按一定步长，对依赖于参数 a 的一些项求和

## 以过程作为参数

- 这几个过程的公共模式是：

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

许多过程有公共模式，说明这里存在一个有用的抽象。如果所用语言足够强大，就可以利用和实现这种抽象

- Scheme 允许将过程作为参数，下面的过程实现上述抽象

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

其中的 term 和 next 是计算一个项和下一个 a 值的过程



## 以过程作为参数

- 有了 sum，前面函数都能按统一方式定义（提供适当的 term/next）

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

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

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

- 使用的例子：

```
(sum-cubes 1 10)
3025

(* 8 (pi-sum 1 1000))
3.139592655589783
```

收敛非常慢，到 pi/8

- sum 实现的是线性递归计算进程
- 练习1.30 要求写完成同样功能的高阶过程，实现线性迭代



## 以过程作为参数：数值积分

- 抽象有用，表现在可用于形式化其他概念。如 sum 可用于实现数值积分，公式是

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

- 其中 dx 是很小的步长值。实现：

```
(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)
.24998750000000042

(integral cube 0 1 0.001)
.249999875000001
```

x^3 在 [0, 1] 积分的精确值是 1/4



## 用 lambda 构造过程

- 前面用 sum 定义过程时都为 term 和 next 定义过程。只在一处用，给过程命名没什么价值。最好能表达"那个返回其输入值加 4 的过程"，而不专门定义命名过程 pi-next
- lambda 特殊形式用于解决这个问题。一个 lambda 表达式求值得到一个匿名过程。如 pi-sum 可以重定义为：

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

定义积分函数 integral 也不必再定义局部函数：

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

## 用 lambda 构造过程

- lambda 表达式的形式与 define 类似：

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

  下面两种写法等价：

  (define (plus4 x) (+ x 4))

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

  可认为前一形式只是后一表达式的简洁写法
- 对 lambda 表达式求值得到一个过程，因此它可以用在任何需要过程的地方。例如可作为组合式的运算符：

  ((lambda (x y z) (+ x y (square z))) 1 2 3)
  *12*

  第一个子表达式求值得到一个过程，该过程被应用于其他参数的值



## lambda，let 和局部变量

- 假设要定义一个复杂的函数，例如：

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

较好的定义方式是：

$$a = 1 + xy$$
$$b = 1 - y$$
$$f(x,y) = xa^2 + yb + ab$$

我们常需要引进辅助变量记录算出的一些中间值

- 一种解决方法是定义一个内部的辅助函数

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



## lambda，let 和局部变量

- 这个辅助函数可以用一个 lambda 表达式代替。将定义改为：

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

这里用 lambda 表达式引进两个辅助的局部变量

- 上述写法不够清晰（lambda 表达式较长，与参数的关系不易看清）。这种结构很有用，Scheme 专门引进了 let 结构。f 可重定义为：

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

这个 let 结构引进两个局部变量，并给出它们的约束值



## lambda，let 和局部变量

- let 表达式的一般形式：

  (let ((<$var_1$> <$exp_1$>)<br>
  … …<br>
  (<$var_n$> <$exp_n$>))<br>
  <*body*>)

  读作：令 <$var_1$> 具有值 <$exp_1$>，且 <$var_2$> 具有值 <$exp_2$> 且 <$var_n$> 具有值 <$exp_n$> 做 <*body*>

  前面是一组变量/值表达式对，表示要求建立的约束关系
- let 是 lambda 表达式的一种应用形式加上语法外衣，等价于：

  ((lambda (<$var_1$> ...<$var_n$>)<br>
  <*body*>)<br>
  <$exp_1$><br>
  … …<br>
  <$exp_n$>)
- 用 let 写的表达式比较符合人的阅读习惯，更易读易理解

## lambda，let 和局部变量

- 在 let 结构里，为局部变量提供值的表达式是在 let 之外计算的，其中只能使用本 let 的外层的变量

  一般情况下这种规定并不重要，但如果有局部变量与外层变量重名，弄清楚这一点就非常重要。例如：

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

  这里 let 前部的变量约束部分将把 y 约束到由外面的 x 求出的值，而不是同一个 let 里的约束的这个 x

  假设外面的 x 的约束值是 5，这个表达式的值将是 21



## 作为通用方法的过程

- 基本复合过程是某种计算模式的抽象，其中一些值被参数化
- 高阶过程将计算中所需的一些过程抽象为参数，因此威力强大
- 下面讨论两个更精妙的实例，研究找函数的 0 点和不动点的通用方法
- 基于这两个方法定义的过程可用于解决大量具体问题

- 例，区间折半法找方程的根：

  给定区间 [a, b]，若有 f(a) < 0 < f(b)，[a, b] 中必有 f 的零点。

  折半法：取区间中点 x 计算 f(x)，根据其正负将区间缩短一半

  计算所需步数 O(log(L/T))，L: 初始区间长，T: 容许误差



## 找函数的零点

- 实现折半法的过程：

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

- 判断区间足够小的函数：

```
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))
```

  可以根据需要选用其他评价区间足够小的方式



## 找函数的零点

- 直接用 search，用户提供的区间可能不满足要求。为方便，可以定义一个包装过程，在参数合法时调用 search：

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

  error 是发错误信号的内部过程，它逐个打印各参数（任意多个）

- 使用实例：求 pi 的值（sin x 在 2 和 4 之间的零点）：

  (half-interval-method sin 2.0 4.0)
  *3.14111328125*

- 很多问题可能归结到找函数 0 点的问题（求函数的根）

## 找函数的不动点

- 有一些函数，从一些初值出发反复应用它们，可以逼近一个不动点
- 下面是个求不动点过程，它反复应用 f 直至连续两个值足够接近：

```
(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))
```

- 用这个函数求 cos 的不动点，以 1.0 作为初始值：

```
(fixed-point cos 1.0)
.7390822985224023
```



## 找函数的不动点

- x 的平方根可看作 f(y) = x/y 的不动点。考虑用下面求平方根过程：

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

可惜它一般不终止（产生的函数值序列不收敛），因为：

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

- 控制这种振荡的方法是减少变化的剧烈程度。因为答案必定在两个值之间，可以考虑用它们的平均值作为下一猜测值：

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

这将使计算收敛。这种减少振荡的方法称为平均阻尼技术

- 许多问题可以归结为求不动点。书上有许多练习，其中讨论了许多与正文有关的有趣问题。无论做或不做，都值得去读一读、想一想



## 过程作为返回值

- 在一个过程里创建新过程并将其返回，也是一种很有用的程序技术，能增强我们表述计算进程的能力
- 实例：不动点计算中的平均阻尼是一种通用技术：求函数值和参数值的平均。求给定函数 f 的平均阻尼值，是基于 f 定义的另一个过程
- 下面过程对 f 算出相应的平均阻尼过程：

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

- 将 (average-damp f) 返回的过程作用于 x，可得到所要平均阻尼值

```
((average-damp square) 10)
55
```



## 过程作为返回值

- 前面平方根函数可以重新定义为：

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

- 注意这一新定义的特点：
  - 它基于两个通用过程，它们分别求不动点和生成平均阻尼函数。这两个通用过程都可以用于任意函数
  - 这里的具体函数是用 lambda 表达式直接构造的
- 有兴趣的同学可以考虑：
  - 能不能在 C 语言里做出上面这样的抽象。如果能，怎么做？如果不能，为什么？
  - 在 C++ 和 Java 里可以做出类似的抽象，C++ 支持运算符重载，事情可能做的更自然。请各位想想怎么做
  - 考察 C#、Java 和 C++ 里的 lambda 库的研究和使用的情况

## 牛顿法

- 下面用牛顿法作为返回 lambda 表达式的另一应用
- 一般牛顿法求根牵涉到求导，g(x) = 0 的解是下面函数的不动点

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

求导就是从一个函数计算出另一个函数

- 现在考虑一种“数值导函数”，g(x) 的数值导函数是

D g(x) = (g(x + dx) – g(x)) / dx

这里的 dx 取一个很小的数值，例如 0.00001

- 生成“导函数”的过程可定义为：

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



## 牛顿法

- 将 dx 定义为一个全局变量（也可以定义为参数）：

```
(define dx 0.00001)
```

- 现在就可以对任何函数求数值导函数了。例如：

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

*75.00014999664018*

- 牛顿法可以表述为一个求不动点的函数：

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

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

- 这样定义的牛顿法求根函数可以用于任何函数
  - 由于采用数值的 dx，不同函数会有不同误差
  - Scheme 的优势是符号计算，很容易实现符号求导（下一章）



## 抽象和一级过程

- 上面用两种不同方法都把平方根表示为更一般的计算方法的实例：
  1. 作为一种不动点搜索过程
  2. 采用牛顿法，而牛顿法本身也是一种不动点计算

  在两种方法中，平方根都是求一个函数在某种变换下的不动点
- 把这一想法推广，可得到下面通用过程（从猜测出发，求 g 经某种变换得到的函数的不动点）

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



## 抽象和一级过程

- 由它可得平方根函数的另外两个定义：

$y \mapsto x/y$

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

$y \mapsto y^2 - x$

```
(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (- (square y) x))
                            newton-transform
                            1.0))
```

- 编程中要注意发现有用的抽象，识别它们，根据需要加以推广，使之能用于更大范围和更多情况
  - 要注意在一般性和使用方便性之间的权衡
  - 利用所用程序设计语言的能力。不同语言构造抽象的能力不同
  - 库是这方面的范例。函数式和面向对象语言提供了更大的思考空间

## 一级过程

- 程序设计语言对各种计算元素的使用可能有限制。例如：
  - C 语言不允许函数返回函数或数组
  - C/Java/C++ 等都不允许在函数里定义函数
- 具有最少使用限制的元素被称为语言中的“一等”元素，它们是语言里的“一等公民”，具有最高的特权。常见的特权包括：
  - 可以用变量命名（在常规语言里，可以存入变量，取出使用）
  - 可以作为参数提供给过程
  - 可以由过程作为结果返回
  - 可以放入各种数据结构
- Scheme（像其他 Lisp 方言一样）为过程提供了完全的一等地位。这种做法给实现带来一些困难（后面讨论），但也获得了极其惊人的描述能力（前面已经看到一些例子，下面将看到更多例子）



## 关注

- 递归的形式和代价
  - 线性迭代和尾递归
  - 线性递归
  - 树形递归
- lambda 表达式，它生成的过程
- let 表达式和局部变量
- 过程作为过程的参数和返回值
- 不动点的概念和计算
- 发掘并利用有用的编程模式，通过抽象构造通用的过程（或其他有用的程序抽象结构）
- 程序设计语言里的一等公民