1。构造过程抽象(3)

要点：

- 高阶过程：以过程为参数和/或返回值的过程
- **lambda** 表达式
- **let** 表达式
- 用过程作为解决问题的通用方法
  - 求函数的 **0** 点
  - 求函数的不动点
- 返回过程值
- 过程是语言里的一等公民 **(first-class object)**

高阶过程

- 过程是抽象，一个过程描述了一种对数据的复合操作

  如求立方过程：

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

- 换个方式，也可以总直接写组合式：

  (* 3 3 3)

  (* x x x)

  (* y y y)

- 不定义过程，总基于系统操作描述，不能提高描述的层次

  虽然也能计算立方，但程序里没建立立方的概念

  将常用公共计算模式定义为过程并命名，就是在程序里建立概念

  过程抽象起着建立新概念的作用。基于定义好的过程编程，就是基于更高级的新概念思考问题，非常重要

高阶过程

- 只能以数值作为参数也限制了建立抽象的能力。例如

  在一些计算模式里，在某个（某些）地方可以使用不同的过程

  要建立这类计算模式的抽象，就需要以过程作为参数或返回值

- 高阶过程：以过程作为参数或返回值，操作过程的过程
  - 高阶过程是非常有用的抽象工具
  - 语言允许定义高阶过程，就能更好地支持描述复杂的程序，因为它们为定义抽象提供了更多的可能性
- 常规语言也提供了一定的定义高阶过程的能力
  - 一些语言里函数提供了过程参数
  - **C/C++** 函数可以指向函数的指针参数，用于传操作性的实参
  - 但这些功能的能力都比较有限
  - **Java**、**C#** 等引进 **lambda** 表达式，是为了在这方面有所前进

高阶过程

- 下面要说明高阶过程抽象的价值（后面章节有很多例子）
  - 高阶过程有什么用，应该怎样定义和使用
  - **lambda** 表达式的构造和使用
  - 也帮助理解 **Java** 等语言引进相关功能的背景
- 需要高阶过程的一类情况
  - 一些计算具有相似的模式，只是其中涉及的几个操作不同
  - 要利用这种公共模式，就需要把这几个操作参数化
  - 具有参数化操作的过程，就是高阶过程（一类情况）
- 以数值为参数的过程（作为比较）
  - 一些计算具有类似的情况，只是其中牵涉的几个数值不同
  - 要利用这种公共性，就需要把几个被操作的数值参数化
  - 这样就定义出了以数值为参数的过程

以过程为参数

- 考虑几个过程：

```
(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))))
```

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

收敛到π/8

虽然细节有许多差异，但它们有共同的模式：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

以过程作为参数

- 把其中的公共模式较严格地写出来（标明变化的部分）：

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

一些过程有公共的模式，说明可能存在一个有用的抽象。如果编程语言的功能足够强，就有可能利用和实现这种抽象

- **Scheme** 允许以过程为参数，上面抽象的实现很直接：

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

参数 **term** 和 **next** 是计算一个项和计算下一 **a** 值的过程

以过程作为参数

- 几个函数都能基于 sum 定义（只需提供具体的 term 和 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) )
```

- 使用示例：

```
(sum-cubes 1 10)
3025

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

收敛非常慢，收敛到 π

- 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，要求它实现线性迭代

以过程作为参数：数值积分

- 如果某个抽象真有用，那么一定能用它形式化很多概念。例如，可以用 sum 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

- 实现和使用：

```
(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

在 C 语言里以“过程”为参数

- C 语言里不允许把函数为参数，但允许函数指针参数

  由于有类型，函数指针参数也要声明指针的类型

- 首先声明

```
typedef double (*MF) (double);
```

- 可定义（没用 inc，加上后定义复杂一点，也更灵活）：

```
double sum (MF f, double a, double b, double step) {
  double x = 0.0;
  for ( ; a <= b; a += step) x += f(a);
  return x;
}
```

- 可以基于 sum 定义其他函数，如：

```
double integral (MF f, double a, double b, double dx) {
  return sum(f, a + dx/2, b, dx) * dx;
}
```


过程和命名

- 前面用 sum 时都为 term 和 next 定义了专门过程。如

```
(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) )
```

- 这些过程只用一次，为它们命名没价值

  最好是能表达“那个返回其输入值加 4 的过程”等概念，而不专门定义 pi-next 等命名过程

- 这实际上是一个数学问题：

  f(x) = sin x + x

  说了两件事：描述了一个函数，并给它取了名字

- 实际上，定义函数和给它命名是两件事

  应该作为两件事，分开做，考虑如何说一个函数但不命名

用 lambda 构造过程

- Scheme 通过 lambda 特殊形式处理这个问题，求值“lambda表达式”得到一个匿名过程

  lambda 表达式把一段计算参数化，抽象为一个（匿名）过程

- 例，下面几个表达式都计算距离，有共同的计算模式

  ```
  (sqrt (+ (* 5 5) (* 8 8)))
  (sqrt (+ (* 11 11) (* 17 17)))
  (sqrt (+ (* 128 128) (* 213 213)))
  ```

  可以考虑其共性的抽象，建立相应的过程对象

- 描述相应过程对象的lambda表达式：

  ```
  (lambda (x y) (sqrt (+ (* x x) (* y y))))
  ```

  求值这个表达式，得到“求距离的过程对象”

用 lambda 构造过程

- 利用 lambda 表达式重新定义 pi-sum：

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

- 两个 lambda 表达式

  ```
  (lambda (x) (/ 1.0 (* x (+ x 2))))
  ```

  带有一个参数 x 的 lambda 表达式

  表达式的体是 (/ 1.0 (* x (+ x 2)))

  ```
  (lambda (x) (+ x 4))
  ```

  带有一个参数 x 的 lambda 表达式

  表达式的体是 (+ x 4)

用 lambda 构造过程

- 用同样技术定义积分函数 integral，也不再定义局部函数：

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

- lambda 表达式的思想源自数学家 A. Church 提出的 λ-演算

  Church 当时也在研究计算的抽象表述问题

  他在 1940 年代提出了 λ-演算，其工作的精髓是提出了一套数学记法和规则，表示函数的描述与函数应用

  已证明，λ-演算是与图灵机等价的计算模型

  λ-演算可以看作抽象的编程语言，它是编程语言理论研究的一个重要基础和工具，在计算机科学领域具有重要地位

Lambda 表达式

- Scheme 里 lambda 表达式的一般形式：

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

  形式上与 define 类似，有形式参数表和体（但没有过程名）

  lambda 是特殊形式，其参数不求值

- 实际上，用 define 定义过程只是一种简写形式，下面两种写法等价：

```
(define (plus4 x) (+ x 4))                  定义一个过程并用 plus4 命名
(define plus4 (lambda (x) (+ x 4)))         给 plus4 关联一个过程对象
```

- 一般说：

  (define (<*name*> <*formals*>) <*body*>)　　相当于

  (define <*name*> (lambda (<*formals*>) <*body*>))

  Scheme 允许前一写法只是为了易用，没有任何功能扩充

Lambda 表达式

- 求值 **lambda** 表达式建立一个新过程

  建立的过程可以用在任何需要用过程的地方

- 在 **Scheme** 语言里，过程也是对象

  就像整数、字符串等，都是对象

  程序运行中可以建立各种对象

  求值 **lambda** 表达式是建立过程对象的唯一方式

- 由于 **lambda** 表达式的值是过程，因此可以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

  得到的过程应用到其他参数的值（组合式的求值规则）

Lambda 表达式

- 前面几个例子用 **lambda** 表达式作为过程的参数
  - 组合式求值时先求值各参数表达式
  - 对作为参数的 **lambda** 表达式求值得到相应过程，它们被约束到形参，然后在过程里面用
- 这样直接使用 **lambda** 表达式，主要作用是
  - 不引入过程名，简单（如果只用一次）
  - 直接描述过程，有可能使程序更清晰
- 这些还没表现出 **lambda** 表达式的本质价值
  - 前面实例里的 **lambda** 表达式都是静态确定的
  - 下面会看到在更复杂的环境中 **lambda** 表达式的价值
- **C** 等常规语言没有 **lambda** 表达式
  - 但至今为止的情况都可以用命名函数模拟

Lambda 表达式应用：保存中间信息

- 设要定义一个复杂函数，例如：

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

直接定义，有些子表达式需要多次计算

  - 如果子表达式计算非常耗费资源，就会带来很大资源浪费
  - 多次出现同一表达式，造成代码依赖，给程序维护修改带来困难

- 一种技术是引进临时变量保存中间信息：

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

- 程序中常出现这类情况

需要引进辅助变量记录一段程序计算出的中间值

保存中间信息

- Scheme 里的一种解决方法：定义一个内部辅助函数

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

- 这里的技术是：

1. 把各公共表达式抽象为辅助过程的参数，定义辅助函数，它基于这些参数描述所需计算，从公共子表达式的值算出整个表达式的值
2. 在函数调用中以各公共表达式作为对应实参，安排好过程返回值

对更复杂的情况，也可以考虑多层分解

用 lambda 保存和传递中间信息

- 前面辅助函数可以用一个 **lambda** 表达式代替。定义：

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

为 **lambda** 表达式引进两个参数，以公共表达式作为应用的对象

- 技术

1, 用一个 **lambda** 表达式作为运算符，其中公共表达式抽象为参数

2, 把实际的公共表达式作为运算对象

- 这种写法不太清晰（**lambda** 表达式较长，与参数的关系不易看清）

Let 表达式

- 程序中经常需要这种结构，**Scheme** 引进 **let** 结构（是上述 **lambda** 表达式的变形）用于引进辅助变量，传递中间结果
- 过程 **f** 可重新定义为：

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

let 引进了两个局部变量并分别约束值，**let** 体用这些变量完成计算

- 比较：

let 的局部变量约束在前计算在后，更符合阅读习惯，更清晰

用 **lambda** 时计算表达式在前，实参/形实参约束关系不容易看清

let 和局部变量

- let 表达式的一般形式：

(let ((<var_1> <exp_1>)
... ...
(<var_n> <exp_n>))
<*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，表示建立约束关系

- 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>)
<*body*>)
<exp_1>
... ...
<exp_n>)

let 和局部变量

- 在 let 结构里，为局部变量提供值的表达式在 let 之外计算，其中只能引用本 let 之外的变量

简单情况里看不出这种规定的意义

如果出现局部变量与外层变量重名，就会看到这一规定的意义

- 例如：

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

let 头部的变量约束中把 y 约束到由外面的 x 求出的值

不是约束到这个 let 里面的 x 的值

如果外面的 x 约束值是 5，表达式的值将是 21

- 请考虑与上面 let 对应的 lambda 表达式，比较（作为复习题）

作为通用方法的过程

- 高阶过程的功能很强大
  - 表示高级的计算模式，其中把一些操作参数化
  - 解决的是一族问题
    - 用一组适当过程实例化，得到一个具体的过程
    - 把具体计算过程用于适当的数据，得到一个具体计算
  - 可以同时做过程参数的实例化和提供被操作数据
  - 也可以分步提供，为程序的分解和设计提供了自由度
- 下面讨论两个更有趣的高阶函数实例，研究两个通用问题的解决方法：
  - 找函数的 **0** 点
  - 找函数的不动点

  基于这两个方法定义的抽象过程可用于解决许多具体问题

例：找方程的根（函数零点）

- 问题：找区间里方程的根：

  区间 **[a, b]**，若 **f(a) < 0 < f(b)**，**[a, b]** 中必有 **f** 零点（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
- 易见，上面操作做一次，区间长度减半

  假设初始区间的长度为 **L**，容许误差为 **T**

  所需计算步数为 **O(log(L/T))**

  是对数时间算法
- 不难定义一个 **Scheme** 过程实现这个算法

函数零点

- 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))))))
```

- 这里定义的是核心过程
  - 参数应该是被处理函数 f 以及它的一个负值点和一个正值点，只有保证参数正确，才能得到正确结果
  - 过程实现折半计算的基本过程，以尾递归方式定义

函数零点

- 编程原则：
  - 总采用功能分解技术，最高层的过程实现算法框架
  - 把具有独立逻辑意义的子计算抽象为子过程调用
  - 子过程的实现另行考虑
- 需要一个子过程判断区间足够小（谓词）：

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

  评价标准可根据需要确定
- 把判断区间满足要求的方法抽象为过程，另行定义，优点：
  - 可以单独研究判断的技术，选择适当的方法
  - 容易通过替代的方法，独立改进程序中的重要部分
    - 例如采用某种相对度量

函数零点

- 用户提供的区间可能不满足 **search** 的要求（两端点函数值异号）。可以定义一个包装过程，只在参数合法时调用 **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
```


函数零点

- 用折半法求一个函数的根

```
(half-interval-method
  (lambda (x) (- (cube x) (* 2 x) 3))
  1.0
  2.0 )
1.89306640625
```

- 很多问题可以归结到求函数 **0** 点（求函数的根）

  数值计算情况，都可以用 **half-interval-method**

  函数可以事先定义，也可以直接用 **lambda** 描述

例：函数的不动点

- 定义：x 是函数 f 的不动点，如果将 f 作用于 x 得到的还是 x

  f 的不动点就是满足下面等式的那些 x

  $$f(x) = x$$

- 显然，并非每个函数都有不动点。反例很多，如

  ```
  (lambda (x) (+ x 1))
  ```

- 存在一些有不动点的函数，从某些初值出发，反复应用这个函数，可以逼近它的一个不动点

  即使有不动点，也未必满足具有这种性质

  可能与初值选择有关。有这样的函数，从一些初值出发可以达到不动点，从另一些初值出发达不到不动点

- 下面只考虑有不动点的函数，而且假设知道合适的初值。在这种情况下考虑不动点的计算问题

函数的不动点

- 按上面想法求函数不动点的函数

```
(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) )
```

  其中反复应用 f，直至连续两个值足够接近：

- 用 fixed-point 过程求 cos 的不动点，以 1.0 作为初始值：

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

  .7390822985224023

C 高阶函数（不动点）

- 求函数零点的二分法和找函数不动点的过程都容易在 C 语言里实现。
- 下面是一个找函数不动点的 C 函数：

```
const double tolerance = 0.00001;
int closeQ (double a, double b) {
    return fabs(a - b) < tolerance;
}

double fixed (MF f, double guess) {
    while (1) {
        double next = f(guess);
        if (closeQ(next, guess)) return guess;
        guess = next;
    }
}
```

- 有类型问题，只能用于 double 参数返回 double 值的“数学函数”

函数的不动点：控制振荡

- 有些函数存在不动点，但简单地反复应用函数却不能达到不动点
- 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 ))
```

  它一般不终止（产生的函数值序列不收敛），因为：

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

- 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两值之间，可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) )
```

  试验说明，现在计算收敛，能达到不动点（得到平方根）

过程作为返回值

- 上面减少振荡的方法称为平均阻尼技术

  (lambda (y) (average y (/ x y))) 是 **(lambda (y) (/ x y))** 的派生函数，可称为 **(lambda (y) (/ x y))** 的平均阻尼函数

- 看来 **f** 的平均阻尼函数在反复应用中的收敛性优于 **f**，而且 **f** 的平均阻尼函数的不动点一定是 **f** 的不动点（请证明这两点）

- 从函数构造其平均阻尼函数的操作很有用，构造方法具有统一模式，能不能定义过程完成这一构造？

  - 不动点计算中的平均阻尼是一种通用技术
  - 做的事情就是求函数值和参数值的平均值
  - 而从给定函数 **f** 求其平均阻尼函数，却是基于 **f** 定义另一过程

- 在 **Scheme** 里函数用过程表示，上面工作要求定义一个高阶过程，它需要在执行中创建并返回一个新过程，这是个新问题

  这应该是有用的程序技术，因为增强了表述计算进程的能力

过程作为返回值

- 在 **Scheme** 里很容易构造新的过程对象并将其返回

  只需用 **lambda** 表达式描述过程的返回值。最简单的模式：

  (define (g f ...) (lambda (...) ...))

- 下面过程计算出与 **f** 对应的平均阻尼过程：

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

  注意：**average-damp**以过程 **f** 为参数，返回基于 **f** 构造的另一过程，实现的是一种从过程到过程的变换

  对任何函数实参，它都能返回这个实参函数的平均阻尼函数

- 在计算中生成新过程（对象）是前面没遇到过的新问题，实际上这是 **lambda** 表达式最重要的作用

  常规语言（**Java/C++/C#**）引入 **lambda** 表达式的目的也在于此

过程作为返回值

- 把 **(average-damp f)** 构造的过程作用于 **x**，得到所需平均阻尼值

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

- 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))
```

- 注意新定义的特点：

  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

  两个通用过程都可以用于任意函数

  具体函数用 **lambda** 表达式直接构造

- 许多问题可以归结为求不动点。书上有些练习，其中讨论了许多与书中正文有关的有趣问题。值得读一读、想一想

返回构造的过程

- 有兴趣的同学可以考虑：
  - 在 **C++** 里可以做出类似抽象（用函数对象），由于**C++** 支持运算符重载，可以做的比较自然
  - 请各位想想怎么做
- 能否在 **C** 语言里做这种抽象
  - 如果能，怎么做？如不能，为什么？
  - 想想下面问题：
    - 如果能做出这种过程，参数应该是什么，返回值类型是什么？
    - 实际的返回值从哪里来？
    - 有办法建立返回值与参数（两者都是函数）之间的关系吗？
  - 总结上述问题，……
- 总可以做。但如何做的比较自然方便？

主流语言中的 lambda 表达式

- 近几年 C#/Java/C++ 都在积极开发 lambda 表达式功能，其中 C# 走的比较靠前，Java 和 C++ 也在积极跟进

  基本目标是在 OO 语言里提供类似函数式语言 lambda 表达式的功能，主要是提供匿名函数，简化程序书写；也想支持一些高级编程技术

- 各语言的写法不同，C#/Java（考虑）的写法与 Scheme 接近

```
x => x + 1
(x) => x + 1
(int x) => x + 1
(int x, int y) => x + y
(x, y) => x + y
(x, y) => { System.out.printf("%d + %d = %d%n", x, y, x+y); }
() => { System.out.println("I am a Runnable"); }
```

- 有兴趣的同学可以选一个语言，考察其中 lambda 表达式的形式，基本设计考虑，目前人们正在考虑的应用技术（网上有很多讨论）

牛顿法

- 下面用牛顿法求根作为返回 lambda 表达式的另一应用
- 一般牛顿法求根牵涉到求导，设要求 g 的根

  牛顿法说 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

  求导是从一个函数（原函数）算出另一函数（导函数）。在 Scheme 里实现，就是构造新过程

- 可以在 Scheme 做符号求导（下一章讨论）

  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

  做数值计算，可以用一个很小的数值作为 dx，如 0.00001

牛顿法

- 把 dx 定义为全局变量（也可以作为参数）：

```
(define dx 0.00001)
```

- 生成“导函数”的过程定义为：

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

生成的过程是 g 的数值导函数

- 用 deriv 可以对任何函数求数值导函数

例：

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

((deriv cube) 5)
75.00014999664018
```


牛顿法

- 可以把牛顿法定义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))
```

newton-transform 从函数 g 构造出另一个Scheme 过程，它能计算 g 的导函数的近似值。这也是构造新过程

- 这个牛顿法求根函数可以用于任何函数
  - 求导函数的操作是数值计算，得到原函数的数值导函数。对同样 dx 不同函数的误差不同。不同 dx 也可能导致不同的计算误差
  - Scheme 的优势是符号计算，操作各种符号形式的表达式
  - 可以用符号表达式表示代数表达式。在 Scheme 里很容易实现符号求导（下一章的内容）

用牛顿法计算平方根

- 基于 **newton-method** 也可以计算平方根

  x 的平方根可以看作函数 **(lambda (y) (- (* y y) x))** 的 **0** 点

  基于这一观点定义的求平方根过程

```
(define (sqrt x)
  (newton-method (lambda (y) (- (* y y) x))
                 1.0 ) )
```

  从初始值 **1.0** 出发求函数的不动点

- 类似的，**x** 的立方根是函数 **(lambda (y) (- (* y y y) x))** 的 **0** 点

  同样可以基于这一观点求 **x** 的立方根

```
(define (cbrt x)
  (newton-method (lambda (y) (- (* y y y) x))
                 1.0 ) )
```


抽象和一级过程

- 上面用两种不同方法求平方根，都是把平方根表示为另一种更一般的计算方法的实例：
  1. 作为一种不动点搜索过程（搜索）
  2. 采用牛顿法，而牛顿法本身也是一种不动点计算

  都把求平方根看作是求一个函数在某种变换下的不动点

- 这两种方法有共同模式，可以进一步推广为一个通用过程

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

  - 基于一个实现某种变换的过程 **transform**
  - 求某个函数 **g** 经某种变换得到的函数的从一个猜测出发的不动点

  只要一个计算可以表达为这种模式，都可以基于这个高阶过程描述

  只需提供适当的实现变换的 **transform**

抽象和一级过程

- 基于前面高阶过程，可以写出平方根函数的另外两个定义：

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

(define (sqrt x)                                    y ↦ y^2 − x
  (fixed-point-of-transform (lambda (y) (- (square y) x))
                            newton-transform
                            1.0))
```

- 在编程中，要注意发现和总结遇到的有用抽象，正确识别并根据需要加以推广，以便用于更大范围和更多情况
  - 注意在一般性和使用方便性
  - 利用所用语言的能力（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 OO 语言提供了更大的思考空间

一等的过程

- 语言对各种计算元素的使用可能有限制。如：
  - C 语言不允许函数返回函数或数组
  - C/Java/C++ 等都不允许数组和函数的赋值
- 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，过程具有完全一等地位。这给语言实现带来一些困难，也为编程提供了极大潜力。后面有更多的讨论

回顾（第一章）

- 基本语言元素：基本表达式，组合式，过程抽象
- 理解计算进程：计算模式和资源消耗，算法的复杂性
- 高阶过程以过程作为参数和/或返回值
  - 过程作为过程的参数
  - 过程作为过程的返回值（通过计算构造新过程）
- **lambda** 表达式

  求值的结果是一个过程，可以存入变量，可以应用于参数

  let 表达式和局部变量
- 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，其他有用的程序抽象结构
- 程序设计语言里的一等公民（**first-class object**）