1。构造过程抽象(3)

要点：

- lambda 表达式
- let 表达式
- 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
- 以过程作为返回值
- 过程作为语言里的一等公民 (first-class object)

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

在 C 语言里以“过程”为参数

- C 语言里不允许以函数为参数，但允许以函数指针为参数

  由于有类型，以函数指针为参数的函数，要声明指针的类型

- 假设声明

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

- 可定义（没用 inc，用了程序复杂一点，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;
}
```

- 然后就可以定义各种使用 sum 求和的函数了。例如：

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

用 lambda 构造过程

- 前面用 sum 定义过程时都为 term 和 next 定义过程。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) )
```

这些过程只在一处使用，给予命名没有价值。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 特殊形式可解决这个问题，用 lambda 写出的表达式称为“lambda表达式”，求值这种表达式将得到一个匿名过程
- 利用 lambda 表达式，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))
```


用 lambda 构造过程

- 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))
```

- 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写形式

用 lambda 构造过程

- 对 lambda 表达式求值得到一个过程

  它可以用在任何需要过程的地方。如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 表达式的这种直接使用主要可以

  避免引入过多的过程名（如果只用一次）

  直接用作过程，可能使程序清晰一些

- 这些似乎没表现出 lambda 表达式的本质性的价值

  因为上面实例中的 lambda 表达式的内容都是静态确定的。下面很快会看到动态构造 lambda 表达式的价值

- C 语言没有 lambda 表达式，至今为止的情况都可以用命名函数模拟

lambda 表达式

- 假设要定义一个复杂函数，例如：

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

较好的定义方式是：

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

常需引进辅助变量记录算出的中间值

- 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 和 let

- 该辅助函数可以用一个 lambda 表达式代替。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

- 上面写法不清晰（lambda 表达式较长，与参数的关系不易看清），但这种结构很有用。Scheme 专门引进一种 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

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*$_n$>)
  <*body*>)
 <*exp*$_1$>
 … …
 <*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

let 和局部变量

- 在 let 结构里，给局部变量提供值的表达式是在 let 之外计算的

  这种表达式里只能用本 let 之外有定义的变量

  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，这一规定的意义就会表现出来

- 例如：

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

  let 前部的变量约束部分把 y 约束到由外面的 x 求出的值

  而不是约束到同一个 let 里的那个 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

- 高阶过程将计算中所需的一些过程抽象为参数，威力强大

  定义好的高阶过程可能用于解决一族问题

- 下面讨论两个更有趣更精妙的高阶函数实例，研究两个通用方法：
  - 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，使用区间折半法找方程的根：

  给定区间 [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)，根据其正负将区间缩短一半

  计算所需步数为 O(log(L/T))，其中

  L: 初始区间长，T: 容许误差

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))
```

评价区间足够小的标准可根据需要选定

函数的零点

- 用户提供的区间可能不满足 search 的要求（要求两端点值异号）。可定义一个包装过程，参数合法时才调用 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）

函数的不动点

- 有些函数，从一些初值出发反复应用可以逼近它的一个不动点

  f(x) = x

- 下面过程求不动点，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
```


C 高阶函数（不动点）

- 找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。
- 例如，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;
    }
}
```

- 同样有类型问题，上面这个函数也只能用于以 double 为参数返回 double 结果的“数学函数”

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 ))
```

  但它一般不终止（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

- 控制振荡的一种方法是减少变化的剧烈程度。因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) )
```

  这将使计算收敛。这种减少振荡的方法称为平均阻尼技术

过程作为返回值

- 许多问题可以归结为求不动点。书上有些练习，其中讨论了许多与正文有关的有趣问题。值得读一读、想一想
- 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 的平均阻尼函数，是基于 f 定义另一个过程
- 过程 average-damp 算出 f 对应的平均阻尼过程：

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

  注意这个过程：它以过程 f 为参数，返回基于 f 构造的另一个过程

  实现从过程到过程的变换。这是以前没遇到的新问题，新技术，是 lambda 表达式最重要的作用

过程作为返回值

- 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
```

- 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：

  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

  具体函数是用 **lambda** 表达式直接构造的

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
  - 另请考察 **C#**、**Java** 和 **C++** 里的 **lambda** 库的研究和使用情况
- 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？

牛顿法

- 现在用牛顿法作为返回 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

做数值计算，可考虑 dx 取一个很小的数值，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（另一方式是作为参数）：

```
(define dx 0.00001)
```

- 现在可以对任何函数求数值导函数。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

- 上面用了两种不同方法求平方根，实际上是都是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

  其中都把求平方根看作是求一个函数在某种变换下的不动点
- 这一想法也可以推广，可得到下面通用过程

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

从一个猜测出发，求 g 经某种变换得到的函数的不动点

抽象和一级过程

- 现在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 ↦ y² − x
  (fixed-point-of-transform (lambda (y) (- (square y) x))
                            newton-transform
                            1.0))
```

- 编程中注意有用抽象，识别并根据需要推广，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

- 一种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称为语言中的“一等”元素，是语言里的“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
- Scheme（及其他 Lisp 方言）里过程具有完全的一等地位。这给实现带来困难，也带来非常强大的潜能（后面有讨论和更多例子）

回顾

- 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
- 过程作为过程的返回值
- 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，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）