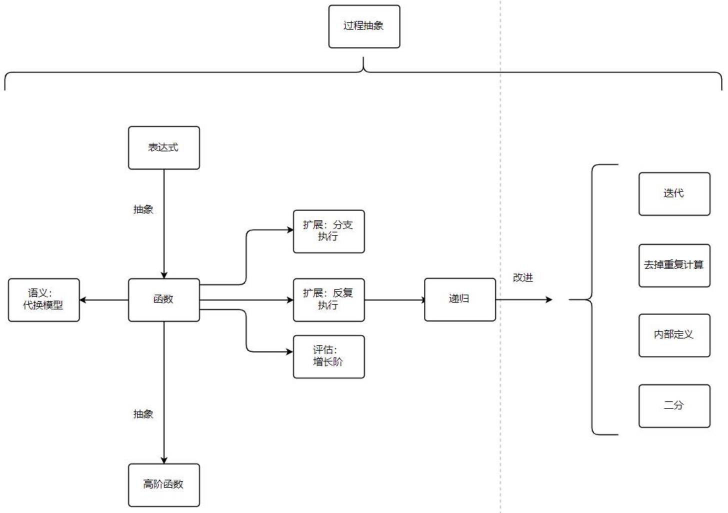
计算机科学导论-SICP
第一章 构造过程抽象
1.3小节 用高阶过程做抽象

王超

Center for Research and Innovation in Software Engineering (RISE), Southwest University

2022 年 9 月 15 日

第一章知识间的关系


过程抽象
表达式
抽象
扩展：分支执行
语义：代换模型
函数
扩展：反复执行
递归
改进
评估：增长阶
抽象
高阶函数
迭代
去掉重复计算
内部定义
二分

1.3节讲了什么故事

- 我们把相似的表达式抽象为过程，那对相似的过程呢
- 遵循类似的思路，我们把相似的过程抽象为某个更高层次的东西，即高阶过程
- 过程的参数和求值结果是数字，而高阶过程的参数和计算结果可以是（高阶）过程或数字
- 高阶过程提供了更强的抽象能力
- 1.3节引入高阶过程的定义，通过例子说明高阶过程可以实现一些更加“通用”的任务

1.3节的PPT涉及哪些内容

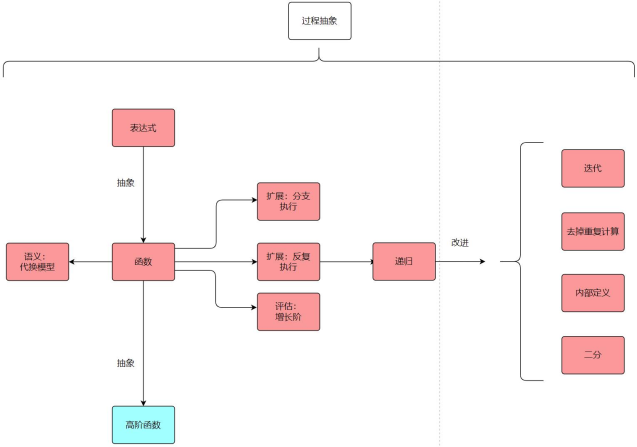
- 以过程为参数的高阶过程
- 以过程为执行结果的高阶过程
- lambda语句和let语句
- 经典例子：求积分，求导，牛顿法，不动点计算

内容调整

- 1.3.3节通过区间折半寻找方程的根不讲

1.3节的内容

红色为已讲解内容，蓝色为本次要讲解的内容



大纲

- 以过程为参数的高阶过程
- 以过程为执行结果的高阶过程
- let语句
- 使用高阶过程
- 结束

引入

一个过程抽象了一种表达式模式，可以理解为是一本操作手册

- 手册里写了每一步该如何操作
- 手册里有些地方是空格子，在使用手册时需要填入参数
- 例如：当前的手册是一本菜谱：x炒鸡蛋，其中x是空格子
- 步骤一：锅加热，倒油，放入鸡蛋，加热并搅拌，之后取出
- 步骤二：锅加热，倒油，先放入葱，用铲子拌一下
- 步骤三：把x和鸡蛋放入锅里，加热并搅拌，之后取出

过程可以调用过程，这进一步加强了语言的抽象能力

- 例子：另一本手册名为举办宴席
- 步骤：从x=1开始，到x=100为止，分别执行使用x炒鸡蛋

但这个能力有时还是不够的

- 有时候，需要以一个流程(手册)，而不是具体材料(西红柿)，作为参数

例子：建模厨师

- 一个厨师被建模为一个过程，参数是菜谱x和材料y，要求依照菜谱x的要求，以y为材料，做菜

更进一步：建模厨师考核

- 一次厨师考核被建模为一个过程，参数是厨师a、菜谱b和材料c，要求厨师a依照菜谱b，以材料c做菜
- 高阶过程(以过程为参数或返回值的过程)可以成为另一个高阶过程的参数

让我们从生活中的例子回到计算机

- 在上一节我们使用牛顿法计算平方根
- 在草稿纸上，给定一个方程f(x)=0，我们可以给出从x_i计算x_{i+1}的公式：$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
- 但我们还无法编程实现“给定方程f(x)，计算方程的解的通用方法”，因为此时方程成为了参数
- 我们目前只能为$f(x) = x^2 - a$，$f(x) = x^3 - a$，$f(x) = x^4 - a$，. . .分别构造算法

困难之处

- 如何在参数中描绘方程f(x)，方程中可能会有各种数字，乘法，指数，三角函数等运算
- 如何在代码中使用这样的描述

以过程为参数的高阶过程

幸运的是，Scheme可以很简单地完成这个任务

- Scheme允许过程的参数是一个过程
- 当过程f的参数a为一个过程时，f的过程体中可直接以过程的方式应用a，例如执行(a 2)

我们称这样的过程为高阶过程

- 高阶过程也是一种过程
- 注意，Scheme的参数是不写类型的，因此需要由程序员记住哪些参数是过程。可以在注释中写明这一点

例子1：通用的牛顿法

- 上一节的牛顿法不够通用，因为improve是针对求平方根定义的
- 如果希望牛顿法变得通用，就需要改进improve的定义

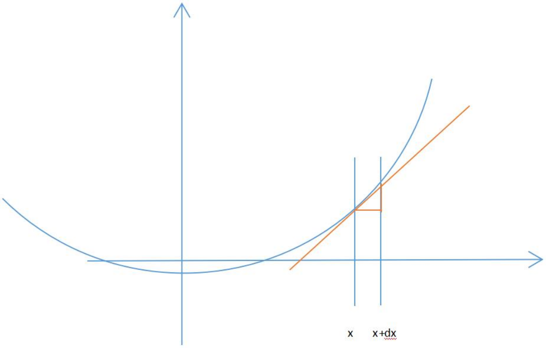
```
#lang sicp

;求平方根
(define (sqrt n)
  (define (sqrt-iter x n)
    (if (good-enough? x n)
        x
        (sqrt-iter (improve x n) n)))
  (define (improve x n)
    (average x (/ n x)))
  (define (average n y)
    (/ (+ n y) 2))
  (define (good-enough? x n)
    (< (abs (- (square x) n)) 0.0001))
  (define (square n)
    (* n n))
  (sqrt-iter 1.0 n))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (sqrt 2)
1.4142156862745097
>
```

- 改进后的improve的任务是根据给定的参数x_i，计算$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
- 为了达到通用的要求，这里的函数f也要作为参数

如何计算$f'(x)$

- $f'(x)$表示在横坐标为x时曲线的斜率
- 计算方法为：$f'(x) = \frac{f(x+dx)-f(x)}{dx}$，这里dx是一个较小的数
- 导数的计算可能是有偏差的，不过，dx选的越小，往往$f'(x)$计算的偏差就越小



- 下面是新的improve方法的代码：计算$x_i - \frac{f(x_i)}{f'(x_i)}$
- 计算$f'(x)$时，选择dx为0.01

```
(define (improve f xi)
  (- xi
     (/ (f xi)
        (/ (- (f (+ xi 0.01)) (f xi))
           0.01))))
```

- (newtons-method g guess)：通用的牛顿法代码。注意，g是由方程左边构成的函数
- 例如，如果求a的平方根，则方程为$x^2 - a = 0$。此时过程g为给定参数x，计算$x^2 - a$
- (good-enough? a)计算的是方程g(a)的值是否足够接近0
- newtons-iter类似之前的迭代函数

```
(define (newtons-method g guess)
  (define (good-enough? a)
    (< (abs (g a))
       0.0001))

  (define (improve f xi)
    (- xi
       (/ (f xi)
          (/ (- (f (+ xi 0.01)) (f xi))
             0.01))))

  (define (newtons-iter h x)
          (if (good-enough? x)
              x
              (newtons-iter h (improve h x))))
  (newtons-iter g guess))
```

如何使用这个通用的牛顿法：以求a的平方根为例

- 需要提供给newtons-method一个过程，这个过程计算$x^2 - a$
- equalForSqrt就是这个过程
- 为了让equalForSqrt能看到sqrt过程的参数a，equalForSqrt被定义为sqrt过程的内部过程

```
(define (sqrt a)
  (define (equalForSqrt x)
    (- (* x x) a))
  (newtons-method equalForSqrt 1.0))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (sqrt 2)
1.4142250094543332
>
```

伏笔：过程的内部过程传递给其他过程时到底发生了什么，可以从第三章的环境模型得到答案

- 第一个例子告诉我们，高阶过程可以处理“总体流程一样，针对每个任务特定的部分不一样”的数学计算
- 我们实现了一个通用的计算工具，解决了一类问题，非常漂亮
- 不过，例子1难免会给人这样的印象：如果不用数值计算方法，高阶过程就意义不大

下一个例子告诉我们，事情并不是这样

- 很多相对常见的运算，包含着一些共通的模式，可以使用高阶过程来抽象

例子2：求和

- 先看一个非常简单的例子
- 实现过程(sum-integers a b)，求a+...+b的和

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

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (sum-integers 1 100)
5050
>
```

当然，也可以使用求和公式直接计算。不过这里的代码主要是为了展示这些问题的共通之处

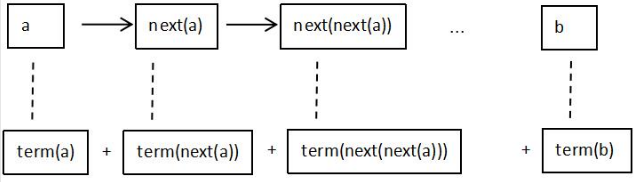
- 再来看一个计算π的方法
- 已知$\frac{\pi}{8} = \frac{1}{1\times 3} + \frac{1}{5\times 7} + \ldots$，或者说，$\pi = \frac{8}{1\times 3} + \frac{8}{5\times 7} + \ldots$
- 实现过程(pi-sum a b)，计算$\frac{8}{a\times(a+2)} + \frac{8}{(a+4)\times(a+6)} + \ldots + \frac{8}{c\times d}$，这里c=$max(\{a+4*i | i \in \mathbb{N} \wedge a+4*i+4 < b\})$
- 例如，如果a=1,b=100，则(pi-sum a b)$= \frac{8}{1\times 3} + \frac{8}{5\times 7} + \ldots + \frac{8}{97\times 99}$
- 换句话说，pi-sum用来计算π，参数a和b调节计算的精度

```
;计算pi
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 8.0 (* a (+ a 2)))
         (pi-sum (+ a 4) b))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (pi-sum 1 1000)
3.139592655589783
>
```

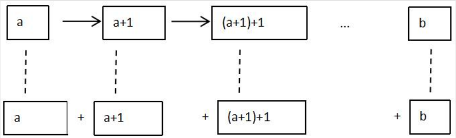
共同的模式

- 都是仅做累加运算
- 累加式的每一个项都有通项公式，可以通过一个数字(称为代表元)计算出来
- 当前项的代表元和下一个项的代表元有着明确的关系
- 假定根据代表元a生成的累加项是term(a)，代表元a的下一个代表元是next(a)
- 图片的上一行是代表元的变化，从a，next(a)，一直变化到b
- 图片的下一行展示了每个代表元对应的项，以及累加操作



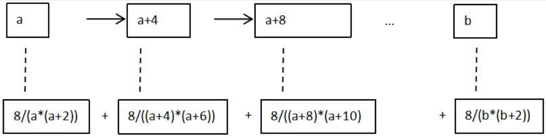
(sum-integers a b)

- 代表元a生成的项就是a：term(a)=a
- 代表元a的下一个代表元是a+1：next(a)=a+1



(pi-sum a b)

- 代表元a生成的项就是$\frac{8}{a\times(a+2)}$：term(a)=$\frac{8}{a\times(a+2)}$
- 代表元a的下一个代表元是a+4：next(a)=a+4



编程实现这个模式

- 实现为一个高阶过程sum
- 参数：term和next两个过程，初始代表元a，和代表元的最大值b

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

类似于普通过程，对高阶过程我们也可以写出递归等式

$$sum(a,b) \;=\; \begin{cases} 0 & if\ a > b \\ term(a) + sum(next(a), b) & otherwise \end{cases}$$

除了更加抽象，高阶过程和上一章的过程差别不大

实例化sum模式

实例化sum，得到sum-integers

- identity过程把x映射为x
- inc过程是Scheme提供的函数，计算+1

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

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

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (sum-integers 1 100)
5050
>
```

实例化sum，得到pi-integers

- pi-term(x)过程计算$\frac{8}{x*(x+2)}$
- pi-next(x)过程计算x+4

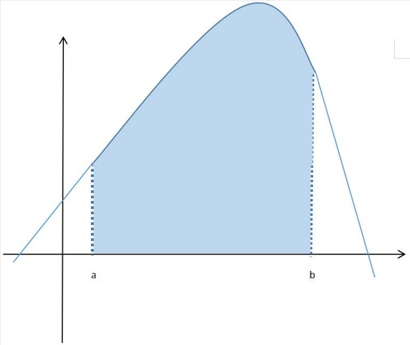
```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))

(define (pi-sum a b)
  (define (pi-term x)
    (/ 8.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (pi-sum 1 1000)
3.139592655589783
>
```

sum与积分

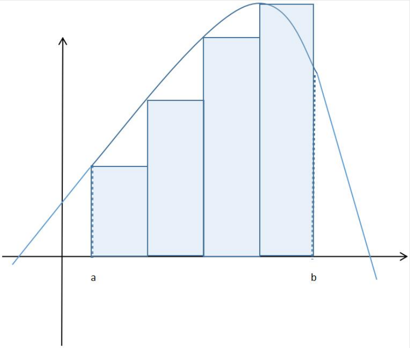
积分

- 积分是高等数学中的基本概念
- 一个积分运算$\int_a^b f(x)\,dx$包括一个函数f(x)，以及两个数字a和b
- 直观上，计算x=a，x=b，x轴和这条曲线围成的图形的面积

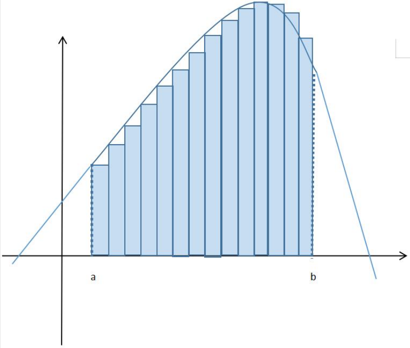


积分的计算方法

- 将[a,b]这个区间平均分为若干份，令每一份长度为d
- 对每个子区间[a_i,a_j]，构造长方形，长方形的一边长度为d，另一边长度为$f(a_i)$
- 这些长方形的面积的和，就是积分的一个近似值
- 以分为4份为例



- 而且，区间[a,b]被分割地越细，这个近似值就越接近积分值
- 以分为14份为例
- 这样，我们就得到了一种计算积分的方法：通过长方形的面积和计算积分，通过缩短长方形的边的长度来改善计算精度



可以应用sum模式计算积分

- $\int_a^b f(x)\,dx = (f(a) + f(a+dx) + f(a+2*dx) + \ldots + f(b)) * dx$
- 积分=累加和*dx，累加和部分使用sum计算
- 需要为sum过程提供term参数和next参数
- term(x)=f(x)，next(x)=x+dx

```
(define (integral f a b dx)
  (define (integral-term x) (f x))
  (define (integral-next x) (+ x dx))
  (* (sum integral-term a integral-next b)
     dx))

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

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (integral cube 0 1 0.01)
0.24502500000000038
>
```

课堂思考题

- 实现过程(f n)，计算1到n的所有偶数的平方和。要求使用高阶过程sum
- 实现迭代版本的高阶过程sum

大纲

- 以过程为参数的高阶过程
- 以过程为执行结果的高阶过程
- let语句
- 使用高阶过程
- 结束

引入

- 高阶过程有时以另一个过程f作为参数
- 有时候我们需要手动生成一个过程f，把它提供给另一个高阶过程作为参数
- 当这样的f较为简单时，可以直接定义f
- 当我们需要构造的多个这样的f有着共同的模式时，我们构造另一个过程g，让g根据不同的参数生成对应的f

例子：二阶导数

- 假定我们正在沿着x轴向右移动，我们在时刻t的位置(x坐标)是过程f(t)的值
- 我们的移动未必是匀速的，例如，可以是f(t)=5 $* x^4$
- 速度度量了移动的程度
- t时刻的速度可以通过$\frac{f(t+dx)-f(t)}{dx}$来计算，dx越小，计算越准确
- t时刻的速度可以理解为是过程f(x)在x=t时的导数
- 什么度量了速度变化的程度呢，是加速度
- t时刻的加速度可以通过$\frac{v(t+dx)-v(t)}{dx}$来计算，这里v是一个计算速度的过程
- 加速度是f过程的导数的导数，即f过程的二阶导数

- 如果我们需要定义一阶导数，二阶导数，三阶导数，...
- 一种做法是每次都从头构造
- 另一种思路：构造一个高阶过程(deriv g)，其求值结果是一个计算过程g的导数的过程
- 一阶导数：某个函数的导数，什么函数呢，原函数g
- 二阶导数：某个函数的导数，什么函数呢，一阶导数
- 三阶导数：某个函数的导数，什么函数呢，二阶导数
- 这样，可以以一种相对易于实现的方式构造任意阶的导数

- (deriv g)过程的求值结果是一个求g的导数值的过程，((deriv g) a)计算g(x)在x=a时的导数
- 我们构造一个过程f-result完成导数计算，并让(deriv g)的求值结果为f-result

```
(define (deriv g)
  (define (f-result x)
    (/ (- (g (+ x 0.0001)) (g x))
       0.0001))
  f-result)


(define (move x)(* 5 x x x x))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (deriv move)
#<procedure>
> ((deriv move) 10)
20000.30000206607
> ((deriv (deriv move)) 10)
6000.117718940601
>
```

- 通过先定义过程f-result，再让(deriv g)的求值结果就是这个过程的方式，完成了返回过程的高阶过程
- (deriv move)是一个过程，计算move的导数
- ((deriv move) 10)是一个数字，即move在x=10时候的一阶导数
- ((deriv (deriv move)) 10)计算了(deriv move)的导数，即move的二阶导数
- $(5 * x^4)' = 5 * 4 * x^{4-1}$，$(20 * x^3)' = 20 * 3 * x^{3-1}$，$(60 * x^2)' = 60 * 2 * x^{2-1}$

可以看出

- 通过这样的方式，我们可以继续计算三阶导数、四阶导数，...
- 高阶过程返回过程的能力是有必要的

匿名过程

- 计算二阶导数时的f-result
- 使用sum实例化sum-integers时的identity
- 使用sum实例化sum-pi时的pi-term和pi-next

这些过程的共同特点

- 仅仅是用来提供给另一个高阶过程作为参数或者执行结果
- 仅被那个高阶过程使用一次

有一种叫做匿名过程的方式，可以简化对这样过程的定义

- 在一个需要过程的地方定义一个无名过程

Lambda语句

- 定义匿名过程
- 格式(lambda (参数列表) 过程体)

例子

- (lambda (x) (* x x))
- 定义了一个匿名的过程，其作用是计算平方
- 匿名过程也有对应的过程应用

匿名过程的应用

- 格式：(匿名过程定义 参数)
- 即((lambda (参数列表) (过程体)) 参数）
- 应用这个匿名过程和给定参数求值
- 例如，((lambda (x) (* x x)) 10)的求值结果为100

在代码中使用匿名过程

- 当过程的执行结果需要是另一个过程时，直接构造一个匿名过程
- 当需要以过程为参数时，直接在需要写参数的地方写下一个匿名过程

使用匿名过程实现二阶导数

- (deriv g)的求值结果是一个匿名过程，过程体正是f-result的过程体

```
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x 0.0001)) (g x))
       0.0001)))

(define (move x)(* 5 x x x x))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (deriv move)
((deriv move) 10)
((deriv (deriv move)) 10)
#<procedure>
20000.30000206607
6000.117718940601
>
```

Lambda与过程定义

- 定义过程的格式是(define (f arg) body)，而定义表达式的格式是(define name body)
- 事实上，过程也可以以类似的方式定义

通过lambda语句

- 过程的定义=使用lambda定义匿名过程+给予名字
- 格式：(define 过程名 匿名过程定义)
- 例子：(define square (lambda (x) (* x x)))

大纲

- 以过程为参数的高阶过程
- 以过程为执行结果的高阶过程
- let语句
- 使用高阶过程
- 结束

- 定义过程时，有些时候一些小的计算过程会反复出现
- 根据代换模型，这些反复出现的小的计算过程每出现一次我们就得计算一次
- 例子：f(x,y)=$1*(2+y)^4+2*(2+y)^3*(1+x^2)+3*(2+y)^2*(1+x^2)^2+4*(2+y)*(1+x^2)^3+5*(1+x^2)^4$
- 我们要求值4次$(2+y)$和4次$(1+x^2)$

```
(define (square x) (* x x))
(define (cube x) (* x x x))
(define (quad x) (* x x x x))
(define (f x y)
  (+ (* 1 (quad (+ 2 y)))
     (* 2 (cube (+ 2 y)) (+ 1 (square x)))
     (* 3 (square (+ 2 y)) (square (+ 1 (square x))))
     (* 4 (+ 2 y) (cube (+ 1 (square x))))
     (* 5 (quad (+ 1 (square x))))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (f 2 4)
12281
>
```

- 一种方法是定义另一个过程，这个过程参数为a和b，求值结果为$1 * a^4 + 2 * a^3 * b + 3 * a^2 * b^2 + 4 * a * b^3 + 5 * b^4$
- 当这个过程的参数$a = (2 + y)$且$b = (1 + x^2)$，则其求值结果正是f(x,y)的值
- 这个过程只会被使用一次，适合使用匿名过程实现

```
(define (square x) (* x x))
(define (cube x) (* x x x))
(define (quad x) (* x x x x))
(define (f x y)
  ((lambda (a b)
     (+ (* 1 (quad a))
        (* 2 (cube a) b)
        (* 3 (square a) (square b))
        (* 4 a (cube b))
        (* 5 (quad b))))
   (+ 2 y)
   (+ 1 (square x))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (f 2 4)
12281
>
```

let表达式

- 匿名过程的应用可使用let语句来重写，执行效果一样，仅格式不同
- 将这些小表达式的位置从最后（即参数）提前到了过程体之前
- 可以让程序员更早的知道这些小表达式有哪些

使用let表达式

- 格式：(*let* ((v_1 e_1) ... (v_k e_k)) 过程体)
- 这里有一个匿名过程，参数为$v_1, \ldots, v_k$
- 求值这个式子时，先求值e_1，...，e_k，再将过程体中的$v_1, \ldots, v_k$分别替换为$e_1, \ldots, e_k$的值，之后求值过程体
- 格式更加清晰：参数和值的列表+过程体
- 注意格式，尤其是括号的数目

```
(define (square x) (* x x))
(define (cube x) (* x x x))
(define (quad x) (* x x x x))

(define (f x y)
  (let ((a (+ 2 y))
        (b (+ 1 (square x))))
    (+ (* 1 (quad a))
       (* 2 (cube a) b)
       (* 3 (square a) (square b))
       (* 4 a (cube b))
       (* 5 (quad b)))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (f 2 4)
12281
>
```

大纲

- 以过程为参数的高阶过程
- 以过程为执行结果的高阶过程
- let语句
- 使用高阶过程
- 结束

- 本小节介绍了一种名为不动点的概念
- 牛顿法的计算可以转化为不动点计算
- 展示了高阶过程的抽象能力

不动点

- 一个函数f(x)的不动点是一个值a，使得f(a)=a

例子

- $2x-1$的不动点是1：$2x-1=x$，进而$x=1$
- x^2-1的不动点是$\frac{1+\sqrt{5}}{2}$和$\frac{1-\sqrt{5}}{2}$：$x^2-1=x$，进而$x^2-x-1=0$
- $\frac{a}{x}$的不动点是$\sqrt{a}$和$-\sqrt{a}$：$\frac{a}{x}=x$，进而$x^2=a$

不动点也可以用来计算开平方，是否可以进而用于解方程呢

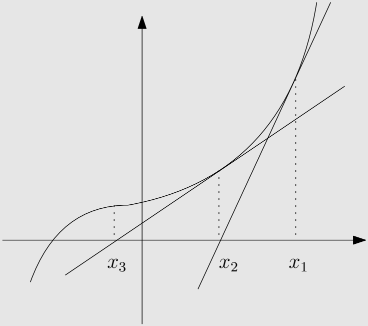
- 可以，因为牛顿法的计算可以转化为不动点的计算

回顾牛顿法

- 给定过程f，解方程f(x)=0
- 猜测值x_1，从x_1计算x_2，从x_2算x_3,...直到算到足够接近方程解的值
- x_2：$f(x_1)$的切线和x轴的交点，可以看到x_1处竖线，x轴和$f(x_1)$的切线构成直角三角形
- 这里稍微修改了计算方法，当算出的某个x_i和x_{i+1}足够接近时，认为x_{i+1}就是方程的根

如何完成从第i轮到第i+1轮的计算

- $f(x_i) + f'(x_i)(x_{i+1} - x_i) = 0$
- $f'(x)$为$f(x)$的导数
- $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$



- $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
- 当算出的某个x_i和x_{i+1}足够接近时，认为x_{i+1}就是方程的根。此时$\frac{f(x_i)}{f'(x_i)}$足够小
- $x - \frac{f(x)}{f'(x)}$的不动点就是方程f(x)=0的解
- 因为此时$x = x - \frac{f(x)}{f'(x)}$

注意

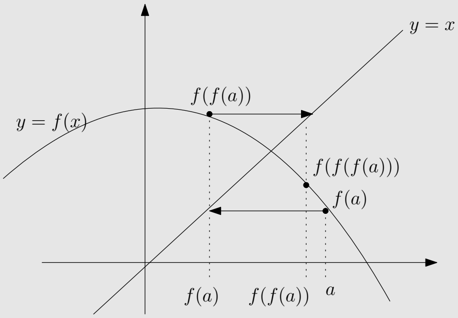
- 在计算浮点数时，我们很少判断两个浮点数相等，而是判断两个数足够接近
- 因为如我们上一节所说，对浮点数的编码是不准确的
- 数学上应该相等的两个浮点数未必相等，例如0.5-0.2和0.1+0.2

那么，如何计算不动点呢

不动点的计算

函数f的不动点的计算方法

- 计算a、f(a)、f(f(a))、. . .，直到收敛
- 最终收敛到y=f(x)和y=x的交点b，此时b=f(b)



代码实现

- let语句使得(f guess)只需求值一次，而不是两次

```
(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))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (fixed-point cos 1.0)
0.7390822985224024
>
```

使用不动点计算牛顿法

把解方程f(x)=0转化为计算$x - \frac{f(x)}{f'(x)}$的不动点

- 参数为待解方程f
- 构造一个过程newton-transform(x)，求值结果是一个计算$x - \frac{f(x)}{f'(x)}$的过程
- 计算newton-transform过程的不动点即可

(deriv g)：求值结果是一个计算过程g的导数的过程

```
(define (deriv g)
  (define dx 0.0001)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))
```

(newton-transform g)

- 使用deriv过程

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

(newtons-method g guess)

- 计算方程g(x)=0的解，猜测的初始解为guess
- 由(newton-transform g)负责构造计算$x - \frac{g(x)}{g'(x)}$的过程，由fixed-point负责计算$x - \frac{g(x)}{g'(x)}$的不动点

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

使用不动点计算牛顿法，并计算平方根。$\sqrt{x}$是方程$y^2 - x = 0$的解

```
(define (fixed-point f first-guess)
  (define tolerance 0.00001)
  (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))

(define (deriv g)
  (define dx 0.0001)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))

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

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

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

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

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (sqrt 2)
1.414213562453059б
>
```

关于fixed-point的讨论

fixed-point算法并不总生效

- 如果f(f(x))=x，则数列x,f(x),f(f(x)),...是数列x,f(x),x,f(x),...，无法收敛

这样的情况可能发生

- 例如，计算a的平方根
- a的平方根是$g(x) = \frac{a}{x}$的不动点
- g(g(x))=$\frac{a}{g(x)} = \frac{a}{\frac{a}{x}} = x$

解决方法

- 每一轮不是从x计算f(x)，而是计算一个即足够接近f(x)又有不同的数字
- 例如，计算$\frac{x+f(x)}{2}$

计算平方根$\sqrt{x}$

- (average-dump f)：求值结果是一个计算$\frac{x+f(x)}{2}$的过程
- average-dump称为平均阻尼过程
- 每一轮使用average-dump处理$\frac{x}{y}$

```
(define (fixed-point f first-guess)
  (define tolerance 0.00001)
  (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))

(define (average a b) (/ (+ a b) 2))

(define (average-dump f)
  (lambda (x) (average x (f x))))

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

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (sqrt 2)
1.4142135623746899
>
```

不动点+average-dump计算平方根$\sqrt{x}$

- 从$\frac{y}{x}$开始
- 做不动点的过程是(average-dump (lambda (y) (/ x y)))
- 求值结果是一个过程，将y映射到$\frac{y+\frac{x}{y}}{2}$

不动点+牛顿法计算平方根$\sqrt{x}$

- 从$y^2 - x$开始
- 做不动点的过程是(newton-transform (lambda(y)(- (square y) x)))
- 求值结果是一个过程，将y映射到$y - \frac{f(y)}{f'(y)} = y - \frac{y^2-x}{2*y} = \frac{y+\frac{x}{y}}{2}$

很巧合地，在计算平方根时，这两个计算方法完全一样

- 后者需给出方程，前者需给出对应的不动点的函数

统一这两种做法

- 方法一：不动点计算，使用average-dump处理原函数
- 方法二：不动点计算，使用newton-transform处理原方程

这两者可以再抽象为一个模式：使用不动点计算一个经过处理的过程

- (fixed-point-of-transform g transform guess)
- 不动点+使用transform变换输入函数g+初始猜测为guess

```
;不动点+变换+原过程
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))

(define (sqrt1 x)
  (fixed-point-of-transform (lambda (y) (/ x y))
                            average-dump
                            1.0))

(define (sqrt2 x)
  (fixed-point-of-transform (lambda (y) (- (square y) x))
                            newton-transform
                            1.0))
```

欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.

```
> (sqrt1 2)
1.4142135623746899
> (sqrt2 2)
1.4142135624530596
>
```

编程语言的第一级元素

- 可以用变量命名
- 可以提供给过程作为参数
- 可以由过程作为结果返回
- 可以包含在数据结构中

过程是Scheme语言的第一级元素

- 赋予了Scheme非常强大的灵活性
- 在你们的其他本科课程中，很难再见到这样的编程语言了

大纲

- 以过程为参数的高阶过程
- 以过程为执行结果的高阶过程
- let语句
- 使用高阶过程
- 结束

课堂总结

类似把相似的表达式抽象为过程，Scheme允许把相似的过程抽象为高阶过程

- 高阶过程允许其参数和求值结果是数字或过程
- 从过程到高阶过程的抽象是有必要的，也可以应用在不少场合

Scheme语法

- 高阶过程
- lambda语句
- let语句

本节内容与其他知识的联系

1.2节涉及和导引的其他课程

- 函数式语言

函数式语言

- 不严格的说，常见的编程语言分为命令式和函数式
- 命令式语言以机器指令为核心，程序员写下的每一条语句的作用是以某种方式访问机器的特定部分，程序是解决问题的步骤
- 函数式语言以函数作为核心，抽象程度更高。代码类似数学公式，程序是计算问题的函数

函数式语言的优点

- 代码简洁，不易出错
- 更容易写出性质良好的代码

典型的函数式语言

- LISP，Haskell，Ocaml

总结与预告

- 我们已经学习了编写程序计算数字问题
- 然而，仅做数字运算有时是不够的
- 一些任务需要我们操作一些实体。这些实体包含多种数据，本身可能也有着复杂的结构
- 例如二维表：我们可以向某行某列写入内容，或读某行某列的内容
- 目前我们无法实现这样的程序，但这样的程序显然是存在的
- 曾经有一种说法：程序=算法+数据结构
- 在第2章，我们将接触数据
- 这将大大扩增我们的编程世界

1.3节练习题

练习级

- 教材练习1.29
- 教材练习1.30
- 教材练习1.31
- 教材练习1.36
- 教材练习1.41

挑战级

- 教材练习1.32

- 系统的学习了高阶过程
- 高阶过程是一个强大的工具
- 有问题及时提问，以及和老师交流

下课