

# 程序设计的基本元素

2022年12月19日 11:50

## 程序设计的基本元素：

基本表达形式

组合的方法（构造复杂元素）

抽象的方法（为复合对象命名，将它们当作单元去操作）

**环境：**要将值与符号关联，而后又能提取出这些值，那么解释器必须维护某种储存能力，以便保持名字-值对偶的关系。这种储存被称为环境（更准确地说是**全局环境**，因为一个计算过程中完全可能涉及若干不同环境）

## 正则序和应用序：

正则序：完全展开而后归约

将按照下面的序列逐步展开：

```
(sum-of-squares (+ 5 1) (* 5 2))  
  
(+      (square (+ 5 1))      (square (* 5 2))  )  
  
(+      (* (+ 5 1) (+ 5 1))    (* (* 5 2) (* 5 2)))
```

而后是下面归约：

```
(+      (* 6 6)      (* 10 10))  
  
(+      36      100)
```

应用序：先求值参数而后应用

# 内部定义

2022年12月20日 12:33

## 内部定义：

将一些内部调用的函数局限在关键函数内部，这样的话那些函数只会在内部起作用（就像是局部变量，出了括号就不存在），便于大程序方便对名称的使用。

例如：牛顿法求平方根

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

直接将所用的过程定义放到内部，这种嵌套的定义称为**块结构**

其实还有更好的方式：因为x在sqrt的定义中是受约束的，过程good-enough?、improve、sqrt-iter也都定义在sqrt里面，也就是说都在x的定义域里，这样就没必要再将x传来传去。我们将x作为内部定义中的自由变量。这样，在外围的sqrt被调用时，x由实际参数得到自己的值。这种方法叫做**词法作用域**。例如改进上面的嵌套：

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

# 递归

2022年12月20日 12:35

完成一些任务需要反复计算，且次数不固定。

例如，计算n的阶乘：

计算6的阶乘，需要5次乘法： $6! = 6 * 5 * 4 * 3 * 2 * 1$

如果过程(factorial n)计算n的阶乘，则过程需执行n-1次乘法

但是一个过程只能调用其他过程固定次。例如sum-of-squares调用两次square，square调用一次乘法，如此看来，一个这样的过程似乎只能完成固定次数的基本运算

那么，如何完成不固定次数的计算？



定义：如果求值(f n)时，求值的步骤会产生需要额外保存的数字和运算，则这个求值的过程称为**递归**计算过程 (recursive process)

递归过程(f x)

f的定义包含递归等式和递归出口

**递归等式**：等式左边是(f x)，等式右边只能出现所有已经定义的过程，以及过程f。

**递归出口**：当x的值足够大（或足够小）到某个阈值时，(f x)可以直接得到计算结果。

如果参数足够小时f直接得到结果，则递归等式中等式右边f的参数一定要小于x。这样，随着程序执行，待求值的f的参数值不断变小。

如果参数足够大时f直接得到结果，则递归等式中等式右边f的参数一定要大于x。这样，随着程序执行，待求值的f的参数的值不断变大，等式右边每向内层求值一次f，都会导致参数更加接近阈值。

因此，递归过程不会导致循环定义。

使用“规模”表示当前参数和递归出口之间的距离

递归算法只需给出“终点”和通用的“中间步骤”，无需给出“每一步”

递归等式描述了“中间步骤”，递归出口描述了“终点”

例如：计算n的阶乘

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

计算6! 的线性递归过程：

```
(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
```

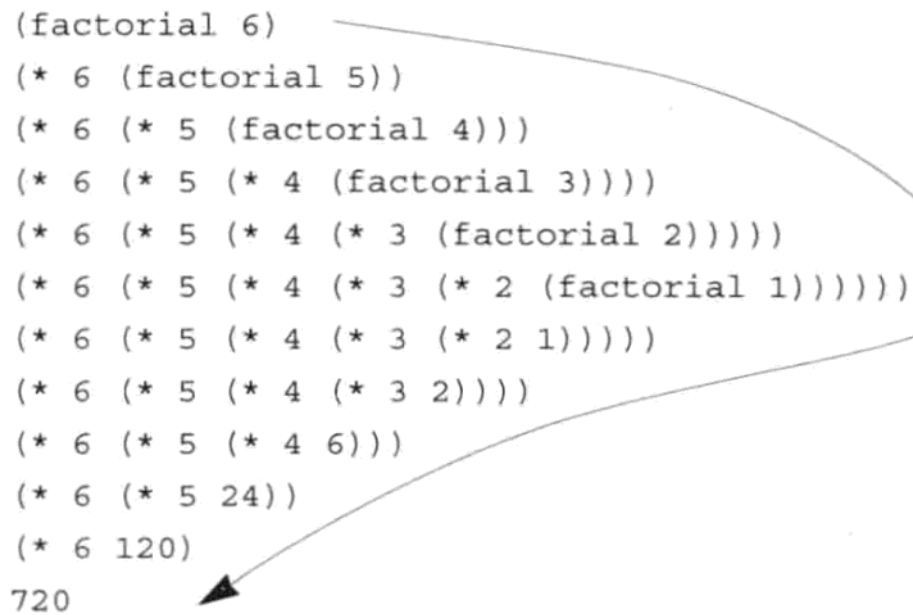


图1-3 计算6!的线性递归过程

### 中间信息

递归等式中不会涉及计算中每一步的那些信息

但是，递归过程在执行时会不断的“深入”，根据代换模型，这些信息会存在表达式中，导致整个表达式越来越长

这些信息就是递归过程执行时需保存的中间信息，由编译器/解释器负责保存。程序员无需关注

例如上图中的\*6 \*5 \*4 \*3 \*2 就是**中间信息**

# 迭代

2022年12月20日 17:31

定义：如果求值(f n)时，求值的每一步都可以用固定数目的状态变量描述，则这个求值的过程称为**迭代**计算过程 (iterative process)

优点：过程不使用递归的信息只有固定数目的参数，而不会产生额外中间信息。

缺点：需要人工设计过程，而不能把递归等式直接翻译为迭代程序。

事实上是不断地在填一张二维表，表的每一行的列数相同。每一列都有自己的含义。表格的每一行的各个列之间都满足一定的关系，过程的执行，就是不断地基于表格的当前行，填写表格下一行，当表格当前行的某些列满足特定条件时，执行终止，此时当前行的某一列就是计算结果，

例如：

;迭代计算阶乘

```
;满足result=counter!,且最终要计算counterBound的阶乘
(define (factorial-iter result counter counterBound)
  (if (= counter counterBound)
      result
      (factorial-iter (* (+ counter 1) result)
                      (+ counter 1)
                      counterBound)))
```

;包装迭代算法

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

图解(factorial 6):

result=counter!	counter	counterBound
1	1	6
$1*(1+1)=2$	2	6
$2*(2+1)=6$	3	6
$6*(3+1)=24$	4	6
$24*(4+1)=120$	5	6
$120*(5+1)=720$	6	6
(factorial 6)的计算结果为720		

# 去掉重复计算

2022年12月29日 20:29

方法：

将计算过的子过程的值保存在某个地方

当需要递归计算时，先去查询之前是否计算过这样的值

如果计算过，就直接返回存储好的值。否则，才会实际运行子过程

# 增长阶

2022年12月29日 11:17

## 增长阶

举例：

递归的斐波那契算法计算(fib n)需要执行至少 $((1 + \sqrt{5})/2)^{n+1} / \sqrt{5}$ 次运算

迭代的斐波那契算法计算(fib n)需要依次计算出(fib 2)、(fib 3)、..., (fib n)。每次计算需要更新一行的三个参数。共执行 $(n-1)*3$ 次运算

当n足够大时,  $((1 + \sqrt{5})/2)^{n+1} / \sqrt{5}$ 远远大于 $(n-1)*3$

程序执行会消耗资源：时间资源，空间资源

定义算法的好坏为：**增长阶（消耗的资源量所处数量级）**的高低

时间增长阶：程序执行消耗的时间资源

空间增长阶：程序执行消耗的空间资源

## 时间增长阶：

用 $R(n)$ 定义在输入的规模为n的情况下，程序执行占用的资源数量。对时间，资源指令的数目，可以简单理解为代换模型执行的行数

输入规模：依问题而定，往往是参数的大小，数据的长度等

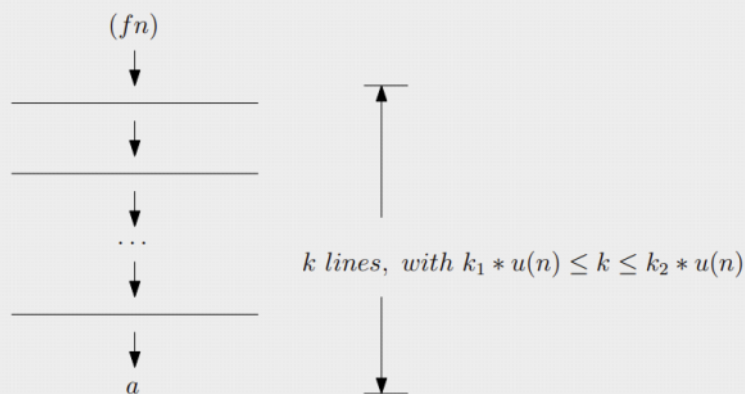
注意，时间增长阶中的时间指的是执行的指令数目(本章中是代换模型执行的次数)，而不是具体的时间

我们称过程f有着 $\Theta(u(n))$ 的时间增长阶，如果存在与n无关的整数 $k_1$ 和 $k_2$ ，使得对每个参数n，(f n)在代换模型下执行的行数在 $k_1*u(n)$ 和 $k_2*u(n)$ 之间

**时间增长阶更在意数量级**，即是 $\Theta(n)$ 还是 $\Theta(n^2)$  由于 $\Theta(n) = \Theta(2 * n)$ ，所以时间增长阶不在意时间是否变化了固定的倍数



时间增长阶的图示，以时间增长阶为 $\Theta(u(n))$ 为例



### 空间增长阶：

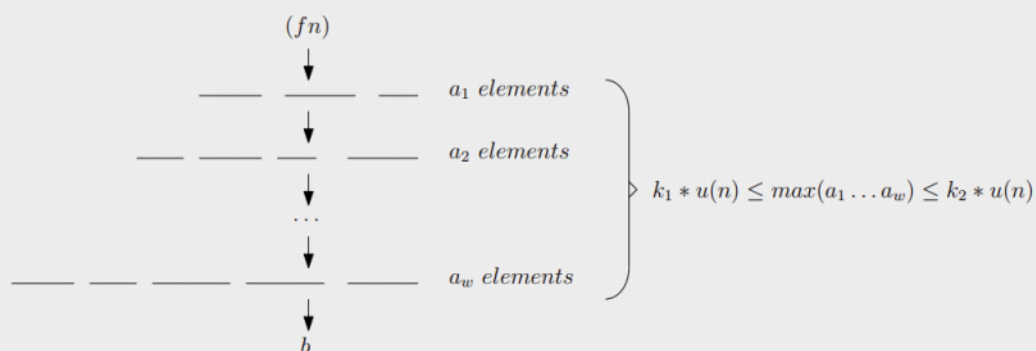
空间增长阶关注程序占用的存储空间

在当前章节可以理解为代换模型执行过程中最长的一行的函数应用+参数的数目

我们称过程 $f$ 有着 $\Theta(u(n))$ 的空间增长阶，如果存在与 $n$ 无关的整数 $k_1$ 和 $k_2$ ，使得对每个参数 $n$ ， $(f\ n)$ 在代换模型下每行的存储空间的最大值在 $k_1 * u(n)$ 和 $k_2 * u(n)$ 之间

类似时间增长阶，**空间增长阶关注的也是数量级**

空间增长阶的图示，以空间增长阶为 $\Theta(u(n))$ 为例



同一个算法对不同输入的“时间”可能是不同的

例如，如果算法的任务需要是在一群人里找人，实现方法是把人排为一列点名。如果要找的人排在第一个，那一步就执行完成；如果要找的人不在队列里，则需要把队列都点名完毕

第一种情况的执行时间是 $\Theta(1)$ ，第二种情况的执行时间是 $\Theta(n)$

(时间/空间)增长阶这个概念往往也有最坏情况下的增长阶和平均情况下的增长阶等细致区分

**一般提到增长阶时，往往指的是最坏情况下的增长阶**

例子：

考察factorial过程的执行，以递归的(factorial 3)为例

- (factorial 3)
- (if (= 3 1) 1 (\* 3 (factorial (- 3 1))))
- (\* 3 (factorial 2))
- (\* 3 (if (= 2 1) 1 (\* 2 (factorial (- 2 1)))))
- (\* 3 (\* 2 (factorial 1)))
- (\* 3 (\* 2 (if (= 1 1) 1 (\* 1 (factorial (- 1 1)))))
- (\* 3 (\* 2 1))
- (\* 3 2)
- 6

时间增长阶：计数(factorial n)的执行行数

- 写下(factorial n)
- 从(factorial k)到(factorial k-1)需2行。这样的减小共n-1次，直到(factorial 1)
- 把(factorial 1)代换为1需要2行
- 之后做n-1次乘法，每次乘法代换1行
- 求值(factorial n)共需 $2*(n-1)+2+(n-1)=3n-1$ 行

因此，时间复杂度是线性的，即 $\Theta(n)$

空间增长阶：

计数(factorial n)执行时元素最多的行的元素数目 参数每减小1，就多一次乘法和—个被乘数，即多2个元素 元素最多的一行是(\* n (\* n-1 ... (\* 2 (if (= 1 1) 1 (\* 1 (factorial (- 1 1) )))))

(这里把if、过程和数字都算一个元素，括号不算)

最后一次判断(if (= 1 1) 1 (\* 1 (factorial (- 1 1) )))共有11个元素。

每次递归生成的\*n\*n-1\*n-2等共有 $2*(n-1)$ 个元素。

共 $2*(n-1)+11=2n+9$ 个元素

因此，这个算法的空间增长阶都是线性的，即 $\Theta(n)$

# 高阶函数

2023年1月17日 18:39

高阶过程：以过程作为参数或者返回值的能操作的过程称为高阶过程

## 一、以过程为参数

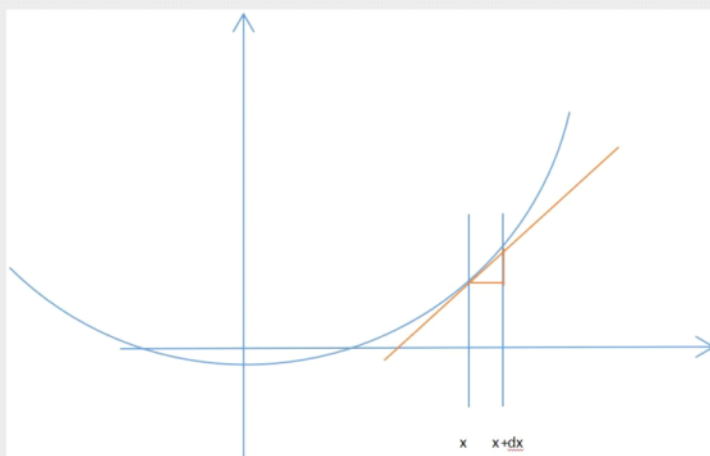
高阶过程可以处理“总体流程一样，针对每个任务特定的部分不一样”的数学计算

例1：将牛顿法通用化，让它不止能求平方根，这就需要改进improve函数（因为之前improve是根据求平方根的等式 $x^2 - 2 = 0$ 来计算下一个预测值的）

- 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



就取dx为0.01为例，improve函数代码如下

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

f是接收的函数式，因为它现在不像求平方根时有现成的式子（ $x^2 - 2$ ）。

最后的代码如下：

；牛顿法，单纯使用高阶过程改造，还不引入不动点

```
(define (newtons-method f guess)
  (define (good-enough? a)
```

```

(< (abs (f a))
  0.0001))

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

(define (newtons-iter f guess)
  (if (good-enough? guess)
      guess
      (newtons-iter f (improve f guess))))
(newtons-iter f guess))

```

这样，如果要求平方根，可以加这样一段代码：

```

(define (sqrt a)
  ;定义函数式
  (define (equalForSqrt x)
    (- (* x x) a))
  (newtons-method equalForSqrt 1.0))

```

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。这里的guess取了1.0。

例2：求和

现在有三个过程：

第一个计算从a到b的各整数之和：

```

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

```

第二个计算a到b之间各整数立方之和

```

(define (cube a) (* a a a))

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

```

第三个是一种计算pi的方式

- 已知 $\frac{\pi}{8} = \frac{1}{1 \times 3} + \frac{1}{5 \times 7} + \dots$ ，或者说， $\pi = \frac{8}{1 \times 3} + \frac{8}{5 \times 7} + \dots$
- 实现过程(pi-sum a b)，计算 $\frac{8}{a \times (a+2)} + \frac{8}{(a+4) \times (a+6)} + \dots + \frac{8}{c \times d}$ ，这里 $c = \max(\{a + 4 * i | i \in \mathbb{N} \wedge a + 4 * i + 4 < b\})$

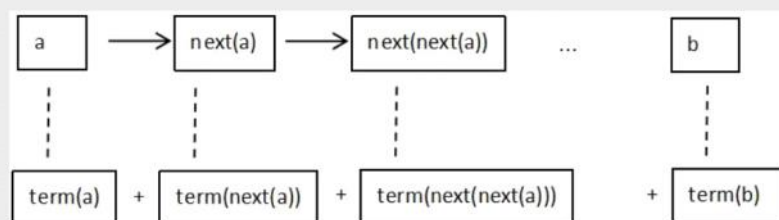
- 已知 $\frac{\pi}{8} = \frac{1}{1 \times 3} + \frac{1}{5 \times 7} + \dots$ ，或者说， $\pi = \frac{8}{1 \times 3} + \frac{8}{5 \times 7} + \dots$
- 实现过程(pi-sum a b)，计算 $\frac{8}{a \times (a+2)} + \frac{8}{(a+4) \times (a+6)} + \dots + \frac{8}{c \times d}$ ，这里 $c = \max(\{a + 4 * i | i \in \mathbb{N} \wedge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

代码如下：

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

这三者有共同的模式：

- 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



这样就可以抽象出一个高阶过程：

参数：term（计算各个项）和next（计算下一个代表元）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) b))))
```

sum高阶过程实例化：

求立方根之和：

```
(define (cube a)(* a a a))

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



```

      (+ (term a)
         (sum term (next a) next b))))

(define (sum-cubes a b)
  (sum cube a inc b))

```

求pi:

```

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

```

用sum高阶过程计算积分:

$\int_a^b f(x) dx = (f(a) + f(a + dx) + f(a + 2 * dx) + \dots + f(b)) * dx$   
 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

代码:

```

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

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

```

# 高阶函数2

2023年2月8日 10:44

## 二、以过程为执行结果的高阶函数：

(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

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

代码：

```
(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))
```

```
> (deriv move)
#<procedure>
> ((deriv move) 10)
20000.30000206607
> ((deriv (deriv move)) 10)
6000.117718940601
>
```

(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

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

所以可以将f-result写成一个匿名过程 (lambda)：

；用lambda

```
(define (deriv2 g)
  (lambda (x)
    (/ (- (g (+ x 0.0001)) (g x)) 0.0001)))
```

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

## 2.1构造数据抽象

2023年2月9日 12:05

数据结构的必要性：

数字计算通过参数传递数据，有时计算需要涉及大量的数据，且编程时无法确切知道数据的规模，我们需要一种机制，把数据堆积起来，并提供访问方法

**数据结构：**数据以特定的方式被组织起来，每一堆被组织起的数据被视作一个抽象的单元，被称为一个数据结构

比如说集合、树、二维表格。

数据结构提供给外界操作接口，操作数据结构时无需知道数据结构的具体实现方式。

用户只能通过数据结构提供的过程来读和修改数据结构，而不被允许绕过数据结构直接读写其存储的内容

举例说明：大脑只会指挥某块肌肉做收缩或拉伸，不会直接指挥具体肌肉细胞，当大脑给每块肌肉下达指令后，由这块肌肉负责指挥自己的肌肉细胞具体完成行动。否则，每次打字都需要指挥到具体细胞。

**数据抽象：**

所谓的**按愿望思维**就是把系统划分为不同的部分，其中一些部分更为基本，作为基础。在设计系统的其他部分时，假定这些基础部分已经被实现，通过基础部分构造其他部分，基础部分往往足够抽象，不暴露实现细节。是数据抽象的一部分。

也就是先把基础部分抽象化，不管他到底怎么运行，先用这些抽象化的东西设计系统

例如实例：有理数的算术运算

可以认为自己在与接口（方法+参数）打交道，而无需关注接口是如何实现的

以有理数为例

make-rat, numer和denom的代码，是有理数的具体定义

make-rat, numer和denom作为接口，被add-rat等过程使用

这两者彼此是独立的

计算机程序是为了实现人的思想。人不擅长思考冗长的步骤，而善于抽象概念，搭建逻辑层次。所以，计算机代码也应该多使用概念，少暴露具体实现细节

我们的算法可以视为对抽象数据的实现。

**抽象数据**包含构造函数和选择函数：



构造函数：生成实例，

选择函数：返回属性。

**合约 (规约 specification)** 定义了构造函数和选择函数之间的关系，用来判断构造函数和选择函数是否被正确地实现

序对的过程性表示：

序对就是构造出的dispatch方法，我们可以将car和cdr用自己的过程解释出来：

```
(define (cons x y)
  (define (dispatch m)
    (cond ( (= m 0) x)
          ( (= m 1) y)
          ( else (error "argument not 0 or 1 -- cons" m))))
  dispatch)

(define (car z) (z 0))

(define (cdr z) (z 1))
```

序对z是(cons x y)，我们把(car z)和(cdr z)实现为对过程dispatch的应用。

举例：

$(\text{car} (\text{cons } 1 \ 2)) \Rightarrow (\text{car dispatch}) \Rightarrow (\text{dispatch } 0) \Rightarrow (\text{cond } (0=0 \rightarrow 1) (0=1 \rightarrow 2) (\text{else } \dots)) \Rightarrow 1$

问题：dispatch只有一个，但序对可以有很多。那么a=(cons 1 2)和b=(cons 3 4)在这种实现下是不同的序对吗？

根据刚才的例子，使用a和b时，会被自动替换为带有不同参数的dispatch，

因此，不同的序对对应参数不同的dispatch，

准确来说，可以认为dispatch不是“只有一个”。**每次定义序对时就定义了一个“不同”的dispatch**

# 序对

2023年2月9日 12:38

一种粘合数据的机制，可以把两个数据粘合在一起，或者说存储一对数据。

序对提供给程序员接口：

(cons a b)：构造由a和b组成的序对，序对的第一个元素为a，第二个元素为b

(car x)：返回序对x的第一个元素

(cdr x)：返回序对x的第二个元素

由于序对的元素可能是序对，因此一个序对中最终存储的数字或函数可能超过两个（嵌套）。

**序对的闭包性质：**cons是一种把小数据组合成大数据的操作，而使用cons操作得到的结果（序对）可以被另一个cons继续使用。

某种组合操作满足闭包性质：通过它组合数据得到的结果还可以通过同样的操作进行组合

# 表

2023年2月9日 15:09

表：以特定的格式，用嵌套的序对存储一个串

例如：

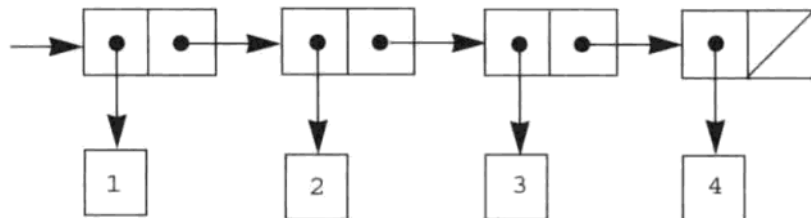


图2-4 将序列1, 2, 3, 4表示为序对的链

内容为1、2、3、4的表它表示为：(cons 1 (cons 2 (cons 3 (cons 4 nil))))

对一些car和cdr组合，提供了缩写形式

```
(cadr items)=(car (cdr items))
(cddr items)=(cdr (cdr items))
(caddr items)=(car (cdr (cdr items)))
(cdddr itmes)=(cdr (cdr (cdr items)))
(cadddr items)=(car (cdr (cdr (cdr items))))
(cddddr items)=(cdr (cdr (cdr (cdr items))))
```

注意这种缩写的含义，其实是倒着取值。

**list**语句用来简化表的构造

格式：(list a1 . . . ak)

构造元素为a1 . . . ak的表

相当于(cons a1 (cons a2 (cons . . . (cons ak nil) . . . )

上例的表可以由语句(list 1 2 3 4)构造。

(list 1 2 3 4)和(1 2 3 4)的含义是不同的。

执行(list 1 2 3 4)的结果是(1 2 3 4)。

执行(1 2 3 4)，Scheme会认为你想执行一个过程（1是过程名），参数为2、3、4，进而报错

表的操作：以下都是方法或者说高阶函数

表的格式形如(表头 余下部分)，天然适合递归求解

## 1、递归地拆开一个表

需要从表的头元素开始依次读表元素，直到找到特定元素并返回。、

例如：

创造(list-ref items n)：返回表items中，下标为n的元素。

表的首元素下标为0, 下一个元素下表为1, . . .

递归出口: 表items的第0个元素=(car items)

递归等式:  $n > 0$ 时, 表items第n个元素=表(cdr items)第n-1个元素

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
```

## 2. 判断到达表尾

例如:

创造(length items): 返回表items的长度, 即包含的元素数目

递归出口: (length nil)=0

递归等式: items不是nil时,  $(\text{length items}) = 1 + (\text{length (cdr items)})$

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))
```

## 3. 递归地构造表

以递归的方式访问原表的过程中, 可以写下如何使用拆解下的原表中元素来递归地构造新表。

例如:

创造(duplicateElement items): 根据items构造一张新表, 里面每个items的元素出现两次

递归出口: (duplicateElement nil)=nil

递归等式:  $(\text{duplicateElement (cons a s)}) = (\text{cons a (cons a (duplicateElement s))})$

代码:

```
(define (duplicateElement items)
  (if (null? items)
      nil
      (cons (car items) (cons (car items) (duplicateElement (cdr items))))))
```

## 4. 递归地黏贴两个表

构造(append items1 items2): 链接两个表

例如,  $(\text{append (list 1 2) (list 3 4)}) = (1\ 2\ 3\ 4)$

不好的方法:

即 $(\text{append items1 items2}) = (\text{append (removeLast items1) (cons (lastElement items1) items2)})$ 。这里removeLast返回一个去掉了最后元素的表, lastElement求表的最后元素。

不好的原因: 1. 需额外实现removeLast和lastElement过程, 2.items1中的元素会被反复遍历, 导致算法较慢

好的方法:

items1=nil时:  $(\text{append items1 items2}) = \text{items2}$

否则:  $(\text{append items1 items2}) = (\text{cons (car items1) (append (cdr items1) items2)})$

```
(define (append items1 items2)
  (if (null? items1)
      items2
      (cons (car items1) (append (cdr items1) items2))))
```

## 5. 迭代地构造表

例如练习2.18:

构造(reverse items)把表items取反, 要保证构造出来的数据有着表的格式。

例子: (reverse (list 2 6 3))=(3 6 2)

如果items是空表或单元素表, 则(reverse items) = items;

单独处理items首元素, 通过(cons (car items) nil)变为新表的表尾, 然后读items中下一个元素, 成为新表的倒数第二个元素, . . .

代码:

```
(define (reverse items)
  (cond ( (null? items) items)
        ( (null? (cdr items)) items)
        (else (reverse-iter (cdr items) (cons (car items) nil)))))
```

;a是items中剩下未处理的表, b是items中以处理部分的取反结果

```
(define (reverse-iter a b)
  (if (null? a)
      b
      (reverse-iter (cdr a) (cons (car a) b))))
```

注意: 第一次输入的a和b至关重要, 因为要先把b的表尾nil首先构建, 所以第一步与剩下的重复的步骤不同。a输入(cdr items), b输入(cons (car items) nil)。

以(reverse (list 2 6 3))为例:

a (待处理列表)	b (已有结果)
(cons 6 (cons 3 nil))	(cons 2 nil)
(cons 3 nil)	(cons 6 (cons 2 nil))
nil	(cons 3 (cons 6 (cons 2 nil)))

## 6. 处理表中特定下标的元素

一些操作需要依据下标来处理表中元素, 使用递归遍历表时, 可以额外记录当前表的首元素在原表中的下标。

构造过程(filterList items pred): 保留表中有特定下标的元素。

参数pred是一个过程 (even? odd?之类), 其将下标值映射为true或false。

(filterList items pred)的执行结果为一张表, 只保留items中下标满足谓词pred的元素

例子: (filterList (list 2 6 3 5) odd?)=(6 5)会留下表中下标为奇数的元素

注意: nil不属于被过滤的部分, 一定会被留下。

定义一个内部过程(f-rec a index), a为当前表, index为当前表首元素在原表中的下标

```
(define (filterList items pred?)  
  ;a为当前表格, index为当前表格的首元素在items中的下标  
  (define (f-rec a index)  
    (cond ( (null? a) nil)  
          ( (pred? index)  
            (cons (car a) (f-rec (cdr a) (+ 1 index))))  
          ( else (f-rec (cdr a) (+ 1 index)))))  
  (f-rec items 0))
```

## 7、对表的映射

以同一种方式改变表的每个元素, 除了nil

构建高阶过程(map proc items)

求值结果是这样一张表: 它的元素顺序和数目和表items一样, 且每个元素是通过对表items的相应元素做proc计算得到的

例子: (map square (list 2 6 3))=(4 36 9)

方法: 递归地拆开表+使用proc处理每一个元素+通过递归调用时生成的中间信息来拼接表。

代码:

```
(define (map proc items)  
  (if (null? items)  
      nil  
      (cons (proc (car items))  
            (map proc (cdr items)))))
```

# 树

2023年2月9日 21:12

不同于表，树有着多层结构。

开始提到的表仅其叶子节点是数字，其他节点无数字。

使用嵌套的表，即元素是表的表，来表示树。

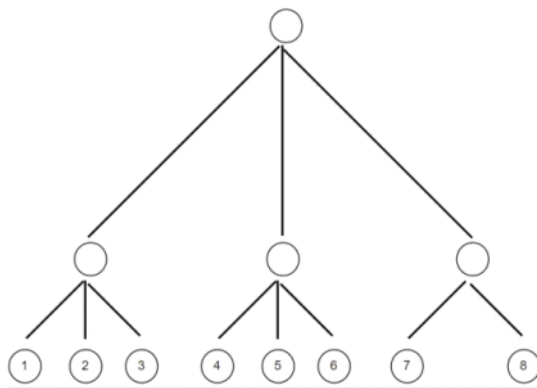
树编码为表的方式：

一棵树是一个表(list a1, . . . , ak), ai是树的第i棵子树，

树的第i棵子树是叶节点时：ai是一个数字；

树的第i棵子树不是叶节点时：ai是编码这棵子树得到的表。

例如：



这棵树可以编码为表s1=(list (list 1 2 3) (list 4 5 6)(list 7 8))

## 拆开一棵树

递归访问树的节点：

递归等式：全部节点= 递归的访问树的第一棵子树的节点 + 访问树其余部分的节点。

为什么要这样？

对表a做递归时，(car a)是数据，而(cdr a)是需要递归处理的表；

对树b做递归时，(car b)和(cdr b)都可能是子树，需要递归处理。

相比表，对树做递归处理时，需要对各棵子树分别做递归处理

构造(count-leaves items)：返回树items的叶节点个数

items=nil时：0；

items是一个数字时：1。

代码实现：

```
(define (count-leaves x)
  (cond ( (null? x) 0)
        ( (not (pair? x)) 1)
        ( else (+ (count-leaves (car x))
                    (count-leaves (cdr x))))))
```

## 对树的映射

以同一种方式改变树的每个叶节点（不包含nil）

构造高阶过程(map-tree proc items)

求值结果是这样一棵树：它的元素顺序和数目和树items一样，且每个叶节点是通过对树items的相应叶节点做操作proc计算得到的

例子：(map-tree square (list (list 1 2) (list 3 4)))=((1 4 ) (9 16))

方法：递归地拆开树+使用proc处理每一个子树+通过递归调用时生成的中间信息把树拼接回去

代码：

```
(define (map-tree proc items)
  (cond ( (null? items) nil)
        ( (not (pair? items)) (proc items))
        (else (cons (map-tree proc (car items))
                      (map-tree proc (cdr items))))))
```

## 另一种方法：

树是一个元素为子树的表，**对树的映射 = 遍历并映射每棵子树**

可以用对表映射中构造的(map proc items)实现这个功能，用参数proc（过程）对每棵子树进行处理，由map自身将处理后的结果拼接为表。

构造：(map-tree2 proc items)。

由于使用map，无需考虑子树为nil的情况。

map的proc：

子树subTree是一棵树时：(f subTree)递归执行(map-tree2 subTree)

子树subTree是一个数字时：(f subTree)的求值结果为(proc subTree)

下面的代码将它实现为一个匿名过程：

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))

(define (map-tree2 proc items)
  (map (lambda (subTree)
        (if (pair? subTree)
            (map-tree2 proc subTree)
            (proc subTree)))
      items))
```

间接递归：map-tree2调用map，而map（在匿名过程内）调用map-tree2

**实际上就是一个对表映射的改造，在map的proc上加上了判断是否为序对这个步骤。**我觉得不如第一种。



例题:

计算树中值为奇数的叶节点的平方和:

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

(define (sum-odd-squares tree)
  (cond ( (null? tree) 0)
        ( (not (pair? tree))
          (if (odd? tree) (square tree) 0))
        ( else (+ (sum-odd-squares (car tree))
                   (sum-odd-squares (cdr tree))))))
```

# 序列作为约定的界面

2023年2月10日 0:14

## 引入

**例1：**上一页的例题：计算树中值为奇数的叶节点的平方和：

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

(define (sum-odd-squares tree)
  (cond ( (null? tree) 0)
        ( (not (pair? tree))
          (if (odd? tree) (square tree) 0))
        ( else (+ (sum-odd-squares (car tree))
                    (sum-odd-squares (cdr tree))))))
```

这是一般写法，寻找数据+计算数据+组合为结果，这三步混杂在递归的过程中。

寻找数据：递归执行到叶节点

计算结果：叶节点为奇数时计算平方，偶数时忽视（被过滤掉）

组合结果：递归调用的中间信息

这样写的缺点：

- 1、不便于提炼算法模式，计算逻辑总和存储逻辑混在一起，
- 2、扩展算法或者改变算法时较为困难。

可以用更自然的解法：

读出所有叶节点（以表的形式保存）为(1 2 3 4)（把树拍扁）

过滤掉偶数值，得到(1 3)

求平方运算，得到(1 9)

组合结果：1+9=10

**例2：**构造出由前n个斐波那契数中为偶数的数组成的表

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ( (f (fib k)) )
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

(fib n) 需要表示，这里省略没写

这是一般写法：

寻找数据：每次执行(next k)时的f

计算结果：f为偶数时保留，f为奇数时忽视（被过滤掉）

组合结果：递归中的cons和nil

一种更加自然的解法：

构造由下标0到下标3构成的表(0 1 2 3)

把每个值a变为(fib a)，得到(0 1 1 2)

过滤掉奇数，得到(0 2)

前面两个例题分别给出了两种算法，不同之处：

1、以树为参数 vs 以数字为参数

2、扫描树 + 处理叶节点 vs 计算斐波那契+求和

### 以表为核心的算法设计思想：

共性1：计算开始于一张表。往往需要人工构造

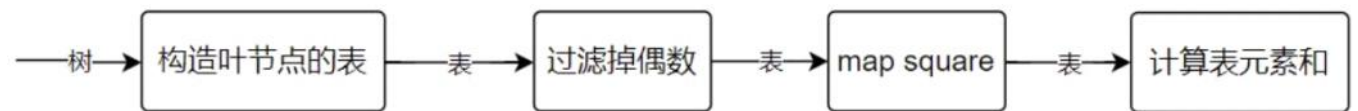
共性2：构造表的过滤操作，过滤掉不参与计算的元素

共性3：使用map（映射表），以统一的方式改变表的每个元素

共性4：从存储着部分结果的表构造最终计算结果

举例：

1、sum-odd-squares的设计方法



2、even-fibs的新解法



重构代码，明确写出各个部分：

生成表；

过滤表：使用过程filter；

改变表的每个元素：使用过程map；

从表计算出最后的结果：使用过程accumulate。

### 生成表：

将树转化成表：

tree为nil时：nil。

**tree为一个数字，即叶节点时：(list tree)，这里需要结果为表。**

剩下的情况，tree的叶节点表=append(tree的第一棵子树的叶节点表，tree余下部分的叶节点表)。

```
(define (enumerate-tree tree)
  (cond ( (null? tree) nil)
        ( (not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                        (enumerate-tree (cdr tree))))))
```

凭空生成一张从小到大排列的表:

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high))))
```

### 过滤表中的元素:

**(filter pred items):** 过滤表items, 只保留满足pred谓词的元素, 最后nil保留。

注意: filter是基于表元素的值过滤的, 而不是基于表元素的下标。

```
(define (filter pred items)
  (cond ( (null? items) nil)
        ( (pred (car items))
          (cons (car items) (filter pred (cdr items))))
        (else (filter pred (cdr items)))))
```

### 改变表的每个元素:

**(map proc items):**

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))
```

### 组合计算结果:

比如求和之类的

**(accumulate op initial items):** 组合表items中已有的计算结果

```
define (accumulate op initial items)
  (if (null? items)
      initial
      (op (car items)
          (accumulate op initial (cdr items)))))
```

例如, 如果求元素和, 则initial=0, op=+

应用以上的计算方法

例1: 重写sum-odd-squares (奇数平方和)

接收到的是树，所以要先用(enumerate-tree tree)生成树tree的叶节点的表。

计算顺序是enumerate-tree, filter, map, accumulate

主要代码：

```
(define (sum-odd-squares tree)
  (accumulate +
    0
    (map square
      (filter odd?
        (enumerate-tree tree))))))
```

(省略各过程具体代码，反正是高阶，用的时候罗列上就行)

例2：重写even-fibs (fib数列偶数表)

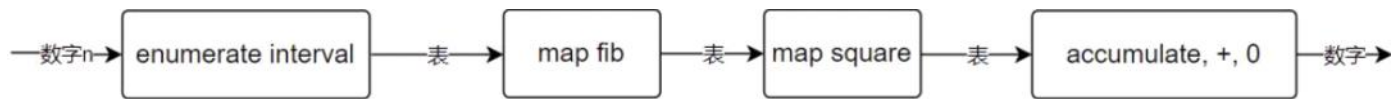
要先生成一张从1到n的表：用(enumerate-interval low high)构造表(low . . . high);  
然后用map对每个元素进行 (fib n) 处理。

计算顺序是enumerate-interval, map, filter, accumulate

主要代码：

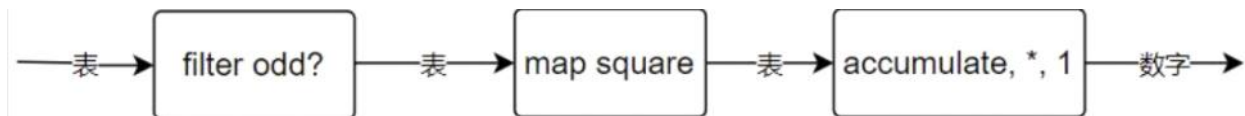
```
(define (even-fibs n)
  (accumulate cons
    nil
    (filter even?
      (map fib
        (enumerate-interval 0 n))))))
```

例3：计算前n个整数的斐波那契值的平方和



```
(define (list-fib-square n)
  (accumulate cons
    nil
    (map square
      (map fib
        (enumerate-interval 0 n))))))
```

例4：计算给定表中那些奇数值的平方的乘积



```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate *
    1
    (map square
      (filter odd? sequence))))
```

**注意：**应用时对表的处理方法并不固定，用几次filter和map，每次的顺序及作为参数的过程，都

是不固定的

# 嵌套映射

2023年2月10日 15:47

**嵌套映射：**有时，使用(map proc)改变表的每个元素时，proc较为复杂，甚至需要使用map来实现proc。

例子：

构造(prime-sum-pairs n)：求值结果是一张表，表的每个元素是一个包含三个元素的表(i j i+j)，满足  $(1 \leq j < i \leq n)$  且i+j为质数。

例如n=4时，求值结果为((2 1 3) (4 1 5) (3 2 5) (4 3 7))。

算法：以n=4为例

①通过enumerate-interval过程，枚举出表(1 2 3 4)。

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high)))))
```

②使用map+某个过程（暂时称为g）处理表(1 2 3 4)，g(k)的值为一张表，其中每个元素为表(k u)且  $1 \leq u < k$ 。例如，g(4)=((4 1) (4 2) (4 3))。

令X1=(map g (enumerate 1 4))=( g(1) g(2) g(3) g(4) )= ( ( ) ( (2 1) ) ( (3 1) (3 2) ) ( (4 1) (4 2) (4 3) ) )

方法：以(g 4)为例，先构造表(1 2 3)，再使用map，把(1 2 3)中的每个x变为(list 4 x)

代码：

```
(lambda (i)
  (map (lambda (j) (list i j))
       (enumerate-interval 1 (- i 1)))))
```

③这个表格不理想，因为( (3 1) (3 2) )多了一层括号，那么g(1) append g(2) append g(3) append g(4)，就可以得到以长度为二的表为元素的表：

令X2=(accumulate append nil X1)= ( (2 1) (3 1) (3 2) (4 1) (4 2) (4 3) )

④在得到了这张表之后，构造一个过程(prime-sum? pair)，判断pair（令为(a b)）中两个数字的和是否是质数：

令X3=(filter prime-sum? X2)= ( (2 1) (3 2) (4 1) (4 3) )，过滤掉和不是质数的二元组。

```
(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair))))
```

⑤构造一个过程(make-pair-sum pair)，把每个(a b)变为表(a b a+b)：

(map make-pair-sum X3)，得到最终结果

```
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
```

嵌套在一起：

```

(define (prime-sum-pairs n)
  (map make-pair-sum
    (filter prime-sum?
      (accumulate append
        nil
        (map (lambda (i)
              (map (lambda (j) (list i j))
                (enumerate-interval 1 (- i 1))))
            (enumerate-interval 1 n))))))

```

把映射+使用append累计定义为一个高阶过程flatmap可以简化一些代码

```

(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))

```

代码可以改进为：

```

(define (prime-sum-pairs2 n)
  (map make-pair-sum
    (filter prime-sum?
      (flatmap (lambda (i) (map (lambda (j) (list i j))
                                (enumerate-interval 1 (- i 1))))
              (enumerate-interval 1 n))))))

```

计算排列：（无法理解）

如何生成集合S的所有排列S' ？

集合S以表的形式给出，且假定S中数字各不相同，S' 应该是一个表，每个元素是一个排列（表示为表）

方法：

令S=(a,b,c)，则**(a b c)的排列 = a打头的排列 + b打头的排列 + c打头的排列**

**= [在(b c)的每个排列前加上a] + [在(a c)的每个排列前加上b] + [在(a b)的每个排列前加上c]**

构造(permutations s)：以s=(1 2 3)为例

①使用map+某个过程（暂时称为g）处理表(1 2 3)，g(k)的值为k打头的排列，即在S - {a}的每个排列前加上k。例如，g(1)=(1 2 3) (1 3 2))

令X1=(map g s)=(g(1) g(2) g(3)) = ((1 2 3) (1 3 2)) ((2 1 3) (2 3 1)) ((3 1 2) (3 2 1)) )

代码实现：

第一步：构造过程(remove item sequence)：从表sequences中去掉item

```

(define (remove item sequence)
  (filter (lambda (x) (not (= x item)))
    sequence))

```

第二步：过程(g x)：已知集合s以及元素x，构造s中x打头的排列的表

使用map实现这个匿名过程

以s=(1 2 3)，x=1为例

首先构造表(permutations (remove x s)) = ((2 3) (3 2))，即S - {1}的排列们

再使用map，把排列(2 3)变为(1 2 3)，把排列(3 2)变为(1 3 2)。得到表((1 2 3)(1 3 2))

```

(map (lambda (x)

```



```

    (map (lambda (p)(cons x p))
      (permutations (remove x s))))
  s)

```

②这个表不理想，因为((1 2 3) (1 3 2))多了一层括号，g(1) append g(2) append g(3)即为我们需要的结果

(accumulate append nil X1)=(1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))

总代码：

```

(define (permutations s)
  (if (null? s)
      (list nil)
      (accumulate append
        nil
        (map (lambda (x)
          (map (lambda (p)(cons x p))
            (permutations (remove x s))))
          s))))

```

# 符号数据 符号求导

2023年2月10日 20:42

## '操作:

使用: ' + 字符串, 或' + (其他内容)。

求值时, 将'后面的内容视为文字而不是变量, 求值结果是字符串

## ' + 字符串

求值结果是后面紧接着的文字, 直到遇到空格为止。

例如, 'swu的求值结果是字符串swu。而不使用'时swu被认为是一个变量, 其求值结果是变量swu的值。

## '+(s)

求值结果是: (以文字形式出现的s)

例如, '(a 1 2)的求值结果是表(a 1 2),

注意, 这里不会代入a的值。

注意, 求值结果(a 1 2)是一个表:

(car '(a 1 2))=a, (cadr '(a 1 2))=1, (caddr '(a 1 2))=2

'可以对表的部分元素使用 当变量a和c的值分别为1和2时, (list a 'b c)的求值结果是(1 b 2)

## (memq item x)

用于判断字符串item是否是表x的元素,

如果是, 则返回表自item第一次出现时余下的部分。否则, 返回false

例如, (memq 'abc '(a b abc d e))=(abc d e)

注意, memq的求值结果并不是同一“类型”。这样做的用处是, 只要其求值结果不是false, 就一定是一个表

代码:

```
(define (memq item x)
  (cond ( (null? x) false)
        ( (eq? item (car x) ) x)
        ( else (memq item (cdr x)))))
```

eq?判断字符串的相等, 但无法判断字符串表的相等

比如会出现以下问题:

```
> (define a '(1 2 3))
> (define b '(1 2 3))
> (eq? a b)
#f
```

### 例子1: 符号求导

这之前实现的数值求导不同，对代数表达式的符号求导不需要数值。

输入是文本形式的代数表达式和变量（代数表达式允许常数，变量，以及加法和乘法运算）。

输出是文本形式的求导结果。

运用符号+按愿望思维。

构造(**deriv** **expr1** **var**):

输入是一个字符串形式的代数表达式**expr1**和字符串形式的变量**var**。

输出是一个字符串形式的代数表达式**expr2**，**expr2**是**expr1**对**var**求导后的结果。

例子: (deriv ' (+ (\* 2 (\* x x)) (\* x 3)) ' x)的求值结果应该是(+ (\* 4 x) 3)，即 $2 * x^2 + 3 * x$ 对 $x$ 求导结果为 $4 * x + 3$

输入和输出都是字符串，写成前缀表达式的形式

求导规则：

$$\begin{aligned}\frac{dc}{dx} &= 0 \quad \text{当} c \text{ 是一个常量，或者一个与} x \text{ 不同的变量} \\ \frac{dx}{dx} &= 1 \\ \frac{d(u+v)}{dx} &= \frac{du}{dx} + \frac{dv}{dx} \\ \frac{d(uv)}{dx} &= u \left( \frac{dv}{dx} \right) + v \left( \frac{du}{dx} \right)\end{aligned}$$

根据按愿望思维，我们先构造出这些基本的式子，先不管到底如何实现：

针对第一个式子和第二个式子

(number? exp): 判断a的值是否是数字。由Scheme提供

(variable? exp): 判断表达式exp是否是单个变量

```
(define (variable? x)(symbol? x))
```

(same-variable? v1 v2): 判断变量v1和v2是否是同一个变量

```
(define (same-variable? a b)
  (and (variable? a) (variable? b) (eq? a b)))
```

针对第三个式子

加法被表现为(+ a b)，这是有三个元素的表

(sum? exp): 判断表达式exp是否是最外层运算为加法的表达式

```
(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
```

(addend exp): 返回exp的被加数，exp为最外层为加法的表达式

```
(define (addend s) (cadr s))
```

(augend exp): 返回exp的加数，exp为最外层为加法的表达式

```
(define (augend s) (caddr s))
```

(make-sum a b): 生成被加数为a，加数为b的表达式

```
(define (make-sum a b) (list '+ a b))
```

针对第四个式子

(product? exp): 判断表达式exp是否是最外层运算为乘法的表达式

```
(define (product? x)
  (and (pair? x) (eq? (car x) '*)))
```

(multiplier exp): 返回exp的被乘数, exp为最外层为乘法的表达式

```
(define (multiplier p) (cadr p))
```

(multiplicand exp): 返回exp的乘数, exp为最外层为乘法的表达式

```
(define (multiplicand p) (caddr p))
```

(make-product a b): 生成被乘数为a, 乘数为b的表达式

```
(define (make-product a b)
  (list '* a b))
```

用代码写出这些规则, 其实就构成好几个递归, 乘法转化为求导加法, 加法转化为求导单变量或常数。

代码:

```
(define (deriv exp var)
  (cond ( (number? exp) 0)
        ( (variable? exp)
          (if (same-variable? exp var) 1 0))
        ( (sum? exp)
          (make-sum (deriv (addend exp) var)
                     (deriv (augend exp) var)))
        ( (product? exp)
          (make-sum (make-product (multiplier exp)
                                   (deriv (multiplicand exp) var))
                    (make-product (deriv (multiplier exp) var)
                                   (multiplicand exp))))
        (else error "error in deriv")))
```

结果没错, 只不过它不会化简, 只会保持它本来的模样

```
(deriv '(+ x 3) 'x)
(deriv '(* x y) 'x)
(deriv '(* (* x y) (+ x 3)) 'x)
(deriv '(* x (* x x)) 'x)
(deriv '(* x 3) 'x)
```

---

欢迎使用 [DrRacket](#), 版本 8.1 [cs].

语言: **sicp**, 带调试; memory limit: **128 MB**.

```
(+ 1 0)
(+ (* x 0) (* 1 y))
(+ (* (* x y) (+ 1 0)) (* (+ (* x 0) (* 1 y)) (+ x 3)))
(+ (* x (+ (* x 1) (* 1 x))) (* 1 (* x x)))
(+ (* x 0) (* 1 3))
>
```

求导过程中有明显的冗余计算：

① $1+0$ 这样的纯数字计算应该被化简，对乘法亦然。

② $x*0$ 和 $1*y$ 中的乘以0和乘以1应该被化简。 $x*0+1*y$ 的化简结果是 $0+y$ ，进一步化简为 $y$ 。

把一个较长的加法或乘法表达式化简可以在**构造加法和乘法表达式时**执行

化简：

### 1、化简加法

$a$ 和0的和、0和 $a$ 的和显然是 $a$ ；

两个数字的和显然是一个数字；

对其他情况，像原来一样计算。

`(=number? exp n)`用来判断`exp`是一个数字且值为`n`

```
(define (make-sum a b)
  (cond ((=number? a 0) b)
        ((=number? b 0) a)
        ((and (number? a) (number? b))
         (+ a b))
        (else (list '+ a b))))
```

### 2、化简乘法

同理

```
(define (make-product a b)
  (cond ((or (=number? a 0) (=number? b 0)) 0)
        ((=number? b 1) a)
        ((=number? a 1) b)
        ((and (number? a) (number? b))
         (* a b))
        (else (list '* a b))))
```

# 集合的表示

2023年2月11日 0:20

集合作为抽象数据

提供给用户如下接口：

(union-set set1 set2): 计算两个集合set1和set2的并集。

(intersection-set set1 set2): 两个集合set1和set2的交集。

(element-of-set? x set): 判断元素x是否属于集合set。

(adjoin-set x set): 计算把元素x放入集合set后的新集合。

常量empty-set, 代表空集。

```
(define empty-set nil)
```

三种实现方法：

使用未排序的表实现集合

使用排序的表实现集合

使用二叉树实现集合

## 方法一：使用未排序的表实现集合

;使用未排序的表实现集合

```
(define (element-of-set? x set)
  (cond ( (null? set) false)
        ( (equal? x (car set)) true)
        ( else (element-of-set? x (cdr set)))))
```

```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
```

```
(define (intersection-set set1 set2)
  (cond ( (or (null? set1) (null? set2)) nil)
        ( (element-of-set? (car set1) set2)
          (cons (car set1)
                (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

```
(define (union-set set1 set2)
  (cond ( (null? set1) set2)
```

```

    ( (null? set2) set1)
    ( (element-of-set? (car set1) set2)
      (union-set (cdr set1) set2))
    (else (cons (car set1)
                 (union-set (cdr set1) set2)))))

(element-of-set? 'a '(1 2 a b 3))
(adjoin-set 'b '(10 a 20))
(intersection-set '(1 a 2 b 3 c) '(b 1 4 8))
(union-set '(1 a 2 b 3 c) '(b 1 4 8))
(adjoin-set 1 (adjoin-set 'a empty-set))

```

---

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

```

#t
(b 10 a 20)
(1 b)
(a 2 3 c b 1 4 8)
(1 a)
>

```

## 方法二：使用已排序的表实现集合

```

(define (element-of-set? x set)
  (cond ( (null? set) false)
        ( (= x (car set)) true)
        ( (< x (car set)) false)
        ( else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)
  (cond ( (null? set) (list x))
        ( (< x (car set))
          (cons x set))
        ( (= x (car set))
          set)
        ( (> x (car set))
          (cons (car set) (adjoin-set x (cdr set))))))

(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      nil
      (let ( (x1 (car set1)) (x2 (car set2)) )
        (cond ( (= x1 x2)
                  (cons x1 (intersection-set (cdr set1) (cdr set2))))
              ( (< x1 x2)

```

```

        (intersection-set (cdr set1) set2))
      ( (< x2 x1)
        (intersection-set set1 (cdr set2)))))))))

(define (union-set set1 set2)
  (cond ( (null? set1) set2)
        ( (null? set2) set1)
        ((let ( (x1 (car set1)) (x2 (car set2)) )
              (cond ( (= x1 x2)
                      (cons x1 (union-set (cdr set1) (cdr set2))))
                    ( (< x1 x2)
                      (cons x1 (union-set (cdr set1) set2)))
                    ( (< x2 x1)
                      (cons x2 (union-set set1 (cdr set2))))))))))

```