2. 构造数据抽象(1)

本节讨论

- 数据抽象的意义
- 建立数据抽象
- 序对：Scheme 语言的基本组合结构
- 复杂的数据，层次性数据
- 表和表操作
- 表映射和树映射
- 以序列作为程序模块之间的约定接口

数据抽象的意义

- 第一章的过程处理简单数据，对解决许多问题都是不够的
- 处理和模拟复杂现象时，常需在程序中构造和处理复杂的计算对象
- 本章关注针对复杂结构的数据的计算，讨论
  - 如何将数据组织起来形成复合数据对象
  - 复合数据对象的处理
- 与构造复合过程一样，构造复合数据也能
  - 提高编程概念的层次
  - 提高设计的模块性
  - 增强语言的表达力
  - 为处理计算问题提供更多手段

有理数计算问题

- 设要实现过程 add-rat 计算两个有理数之和。在基本数据层面上，一个有理数可看作两个整数。因此可以设计两个过程
  - add-num 基于两个有理数的分子/分母算出结果的分子
  - add-den 算出结果的分母
- 显然这种做法很不理想：
  - 如果有多个有理数，记住哪个分子和哪个分母是一对太麻烦
  - 相互分离的两个调用容易写错
  - 还需更多运算，实现/使用都有同样问题
- 应把一个有理数的分子分母粘在一起做成一个复合数据（单位）
  - 有了复合对象，就可以在更高概念层次上定义和使用操作（处理的是有理数而不是两个整数），更加清晰、易理解和使用
  - 隔离了数据抽象的定义（表示细节和操作实现细节）和使用，提高了程序模块性。两边都可以独立修改演化，提高了可维护性

数据抽象：组合

- 考虑实现线性组合 a x + b y。对基本数据写出的过程是：

```
(define (linear-combination a b x y)
  (+ (* a x) (* b y)))
```

- 要想表述线性组合的一般思想，希望用于各种数据，写出的过程：

```
(define (linear-combination a b x y)
  (add (mul a x) (mul b y)))
```

其中 add 和 mul 是对实际数据的适当操作

线性组合过程并不关心具体数据是什么，只要求对它们做适当操作，只要求具体的数据支持 add 和 mul

- 数据抽象能大大提高语言的表达能力

数据抽象的意义

- 实现复合数据和数据抽象，也是建立适当的数据屏障（隔离）。要实现数据抽象，对程序语言有基本要求，需要有：
  - 粘合机制，可用于把一组数据对象组合成一个整体
  - 操作定义机制，定义针对组合数据的操作
  - 抽象机制，屏蔽实现细节，使组合数据能像简单数据一样使用
- 处理复合数据的一个关键概念是闭包：组合数据的粘合机制不仅能用于基本数据，同样能用于复合数据，以便构造更复杂的复合数据
- 本章将讨论：
  - 复合数据如何支持建立以“匹配和组合”方式工作的编程接口
  - 定义数据抽象，也会进一步模糊“过程”和“数据”的差异
  - 处理符号表达式，其基本部分是符号而不是数
  - 通用型（泛型）操作，同样操作可能用于不同的数据
  - 数据制导（导向，驱动）的程序设计，易于加入新的数据种类

数据抽象入门

- 一个过程抽象地描述一类计算的模式，它又可以作为元素用于实现其他（更复杂的）过程，因此是一个抽象。过程抽象
  - 屏蔽了过程的实现细节，可用任何功能/使用形式合适的过程取代
  - 规定了过程的使用方式，使用方只依赖于不多的使用方式规定
- 数据抽象情况类似。一个数据抽象实现一类数据所需要的所有功能，可作为其他数据抽象的元素，就像基本数据元素一样。数据抽象
  - 屏蔽了一种复合数据的实现细节
  - 提供一套抽象操作，使组合数据就像是基本数据
  - 使用接口（界面）包括两类操作：构造函数和选择函数。构造函数基于一些参数构造这类数据，选择函数提取数据内容
- 后面可知，要支持基于状态的程序设计，还要增加一类操作，变动操作（mutation，修改操作）
- 下面用有理数作为例子讨论数据抽象的构造

有理数算术

- 为使用有理数，需要基于分子和分母构造有理数的过程，还要有取分子和分母的过程。分别是：

  (make-rat <*n*> <*d*>) 构造以 n 为分子 d 为分母的有理数

  (numer <*x*>) 取得有理数 x 的分子

  (denom <*x*>) 取得有理数 x 的分母

  这三个过程构成了有理数数据抽象的接口

- 有理数计算规则：

$$\frac{n_1}{d_1}+\frac{n_2}{d_2}=\frac{n_1d_2+n_2d_1}{d_1d_2} \qquad \frac{n_1}{d_1}\cdot\frac{n_2}{d_2}=\frac{n_1n_2}{d_1d_2}$$

$$\frac{n_1}{d_1}-\frac{n_2}{d_2}=\frac{n_1d_2-n_2d_1}{d_1d_2} \qquad \frac{n_1}{d_1}/\frac{n_2}{d_2}=\frac{n_1d_2}{n_2d_1}$$

$$\frac{n_1}{d_1}=\frac{n_2}{d_2} \text{ iff } n_1d_2=n_2d_1$$

有理数算术

- 基于有理数数据抽象，很容易定义实现有理数算术的过程：

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

序对

- 有理数的几个基本操作也需要实现。为此必须有办法能把分子和分母结合为一个整体，构成一个有理数
- Scheme 提供的基本复合结构是“序对”，基本过程 cons 把两个参数结合构造成一个序对，过程 car 和 cdr 取序对中的两个成分

```
(define x (cons 1 2))
(car x)
1
(cdr x)
2
```

- 序对也是数据对象，可用于构造更复杂的数据对象，如：

```
(define y (cons 3 4))
(define z (cons x y))
(car (car z))
1
(car (cdr z))
3
```


有理数的表示

- 考虑直接用序对表示有理数，基本过程很容易定义：

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

定义一个输出有理数的过程（display 是输出值的基本函数）

```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

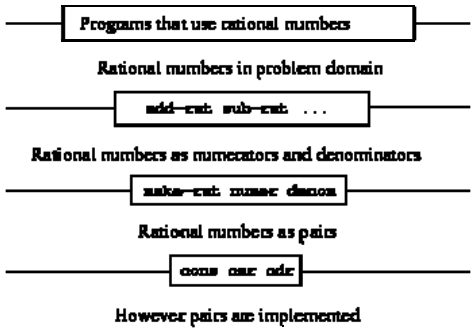
- 为了处理方便，最好把有理数都化简到最简形式，这样分子分母的值也达到最小，相等判断谓词也可以简化。修改定义

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
```

过程 gcd 见 1.2.5节（注意：这一修改对使用完全没有影响）

抽象屏障

- 总结一下有理数算术系统，以及工作中遇到的问题
- 有理数运算都基于基本过程 make-rat、numer 和 denum 实现。一般而言，实现数据抽象，要首先确定一组基本操作，这类数据的其余操作都基于基本操作来实现，不访问基础数据表示
- 下图是有理数系统的结构，可以看到各层次的抽象屏障：



抽象屏障

- 建立层次性的抽象屏障带来的利益：
  - 数据表示和使用隔离，两部分可以独立演化，容易维护修改
  - 数据抽象的实现可以用于其他程序和系统，可能做成库
  - 一些设计决策可以推迟，直到有了更多实际信息后再处理
- 复杂数据抽象有多种实现方式，各有不同特点。不同选择的影响常要到开发一段后才能看清，抽象屏障可大大降低改变实现的代价
- 例如，有理数系统的最简化可以在访问时做，得到另一套实现：

```
(define (make-rat n d) (cons n d))

(define (numer x)
    (let ((g (gcd (car x) (cdr x))))
      (/ (car x) g)))

(define (denom x)
    (let ((g (gcd (car x) (cdr x))))
      (/ (cdr x) g)))
```

对于有理数，这种实现未必好。但许多时候不同实现的优劣并不那么清晰。究竟那种更优，要根据具体情况考虑。例：

链表是否专门保存元素个数？

数据是什么

- 在有理数实现里，定义各种有理数运算时是基于三个开始并未定义的过程，根据被操作的数据对象（分子/分母/有理数）的需要定义，这些对象的行为由三个基本过程刻画。这提出了“数据是什么”的问题
- 简单说“有理数由几个构造和选择函数确定”不够。不是任意三个过程都能构成有理数的实现。有理数的实现要求

  (make-rat (numer x) (denom x)) = x　　对任何有理数 x

  只要三个函数满足这一条件，就可以作为表示有理数的基础。
- 一般说，一类数据对象的构造函数和选择函数总满足一组特定条件
- 同样看法也适合底层。如序对，cons 和 car、cdr 有如下关系

  (car (cons a b)) = a, (cdr (cons a b)) = b

  (cons (car x)(cdr x)) = x　条件：x 是序对
- 理论证明只用过程就可以定义序对，可以不用任何数据结构。计算机科学先驱 Alonzo Church 在研究 λ 演算证明了这一结论。他只用 λ表达式（过程）构造了整数算术系统

数据是什么

- 序对基本过程的另一定义

```
(define (cons x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- CONS" m)))))
(define (car z) (z 0))
(define (cdr z) (z 1))
```

  不难检查：(car (cons 1 2)) → 1　　(cdr (cons 1 2)) → 2

  （书上定义引进局部函数，没必要）
- 这种序对表示方式满足序对构造函数和选择函数的所有条件，完全可以用。Scheme 用存储直接实现序对，主要为了效率
- 本例想说明：过程和数据之间没有绝对的界限，完全可以用过程去表示数据，用数据表示过程。后面会看到这样做的实际价值

实例：区间算术

- 现在考虑设计一个工程问题辅助求解系统，操作对象是不精确的物理量（如测量值）。被计算量有已知的误差，结果应是已知误差的数值
- 例如电子工程师要用下面公式去计算并联电阻的阻值

$$R_p = \frac{1}{1/R_1 + 1/R_2}$$

  厂家生产的电阻通常标注为“xxxΩ 误差 10%”
- 我们想实现一套区间值运算，如四则运算等。需要“区间”数据对象：构造函数 make-interval/选择函数 lower-bound 和 upper-bound
- 加法实现为上下界分别相加：

```
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                 (+ (upper-bound x) (upper-bound y))))
```


实例：区间算术

- 乘法实现为各个界的最小值和最大值构成的区间：

```
(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval (min p1 p2 p3 p4)
                   (max p1 p2 p3 p4))))
```

  其中 min 和 max 求出任意多个参数的最小或最大值
- 除法用第一个区间乘以第二个区间的倒数：

```
(define (div-interval x y)
  (mul-interval x
                (make-interval (/ 1.0 (upper-bound y))
                               (/ 1.0 (lower-bound y)))))
```

练习中提出了一些问题，包括区间的表示和基本操作的实现

实例：区间算术

- 用户看了完成的系统，提出希望能处理由“数值加误差”的方式表示的数据。加入下面构造和选择函数

```
(define (make-center-width c w)
   (make-interval (- c w) (+ c w)))
(define (center i)
   (/ (+ (lower-bound i) (upper-bound i)) 2))
(define (width i)
   (/ (- (upper-bound i) (lower-bound i)) 2))
```

一些工程师希望处理由数值加百分比误差表示的数据。不难增加新的构造和选择函数满足这种需求。这也说明了抽象的价值

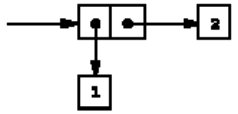
- 有用户基于这一系统，用两种方式计算电阻的并联，发现结果不同：

$$\frac{R_1 R_2}{R_1 + R_2} \qquad \frac{1}{1/R_1 + 1/R_2}$$

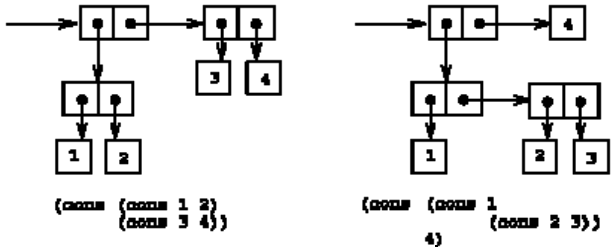
练习要求分析改进这个系统。做下去可完成一个区间计算系统

层次性数据和闭包

- 序对是构造复合数据的基本粘合机制
- 常用图形式表示序对，右图表示 (cons 1 2)。这种图示称为盒子指针表示



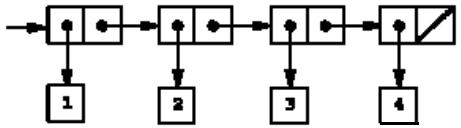
- cons 也可用于组合复合数据，下面是组合 1,2,3,4 的两种不同方式：



- 任意复杂的序对结构都可作为 cons 的参数继续组合，这种能力称为 cons 的闭包性质（序对的 cons 还是序对）。这使 cons 可用于构造结构任意复杂的数据对象

序列的表示

- 可以用序对构造出的最常用结构是序列，即一批数据的有序汇集
- 表示序列的方式很多，最直接的方式如下，其中包含元素 1, 2, 3, 4



构造这个序列的表达式：

```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

nil 是个特殊变量，表示绝不是序对的东西，当作空序列（空表）

- 用 cons 构造出的这种序列称为表。Scheme 有专门构造表的操作：

(list <a_1> <a_2> ... <a_n>)

等价于

(cons <a_1> (cons <a_2> (cons ... (cons <a_n> nil) ...)))

序列的表示

- Scheme 用带括号的元素序列的方式输出表。例如

```
(define one-through-four (list 1 2 3 4))
one-through-four
(1 2 3 4)
```

- 应区分 (list 1 2 3 4) 和输出的 (1 2 3 4)：表达式，和它求值的结果
- 可认为 car 是取表的第一项，cdr 得到去掉第一项后的表：

```
(car one-through-four)
1
(cdr one-through-four)
(2 3 4)
(car (cdr one-through-four))
2
(cons 10 one-through-four)
(10 1 2 3 4)
```

表操作：取元素

- 序列表示一组元素，通过不断求 cdr 能完成对表里所有内容的操作
- 考虑定义 list-ref 返回表 items 中第 n 项元素（n 是 0 时返回首项）

  当 n 是 0 是返回表的 car，否则返回表的 cdr 的第 n-1 项
- 按上述思路写出的过程定义：

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

(define squares (list 1 4 9 16 25))
(list-ref squares 3)
16
```

- 如果参数 n 的值过大（大于表中元素个数）或 n 值小于 0 时，这个函数会出错（对非序对的对象求 cdr 将出错）

表操作：求表长度

- 在向下不断找 cdr 的过程中要判断是否遇到空表
- 例：求表长度的过程：

  空表长度为 0，非空表的长度是其 cdr 的长度加一
- 表长度过程的定义：

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

  谓词 null? 判断参数是否为空表 nil
- 使用实例：

```
(define odds (list 1 3 5 7))
(length odds)
4
```


表操作：拼接

- 另一常用技术：在不断求表的 cdr 的同时用 cons 构造作为结果的表。典型实例：拼接两个表的过程 append：

```
(append squares odds)
(1 4 9 16 25 1 3 5 7)
(append odds squares)
(1 3 5 7 1 4 9 16 25)
```

- append 有两个参数 list1 和 list2。如果 list1 是 nil，直接以 list2 为结果，否则用 cons 把 (car list1) 加在 (append (cdr list1) list2) 前面
- 按这种方式定义的 append 过程：

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr list1) list2))))
```


表操作：任意多个参数的过程

- 基本过程 +、* 和 list 等都允许任意多个参数。我们可以定义这类过程，需要在define 时用带点尾部记法的参数表，形式是：

```
(define (f x y . z) <body>)
```

  圆点前可根据需要写任意多个形参，它们与应用时的实参一一匹配。圆点后的形参（只有一个）关联于其余实参值的表
- 举例：求任意多个数的平方和的过程，可以定义为：

```
(define (square-sum x . y)
  (define (ssum s vlist)
    (if (null? vlist)
        s
        (ssum (+ s (square (car vlist))) (cdr vlist))))
  (ssum (square x) y))
```

- 如果需要处理 0 项或任意多项，参数表用 (square-sum . y)，过程体也需要相应修改。作为课下练习

表的映射

- 一类重要的表操作：把某过程统一应用于表中元素得到结果的表

```
(define (scale-list items factor)
  (if (null? items)
      nil
      (cons (* (car items) factor)
            (scale-list (cdr items) factor))))
(scale-list (list 1 2 3 4 5) 10)
(10 20 30 40 50)
```

- 总结这一计算里的模式，可得到一个很重要的（高阶）过程 map：

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))
(map abs (list -10 2.5 -11.6 17))
(10 2.5 11.6 17)
(map (lambda (x) (* x x)) (list 1 2 3 4))
(1 4 9 16)
```


表的映射

- 高阶过程 map 把对元素的映射提升为对整个表的映射
- 用 map 给出 scale-list 的定义：

```
(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items))
```

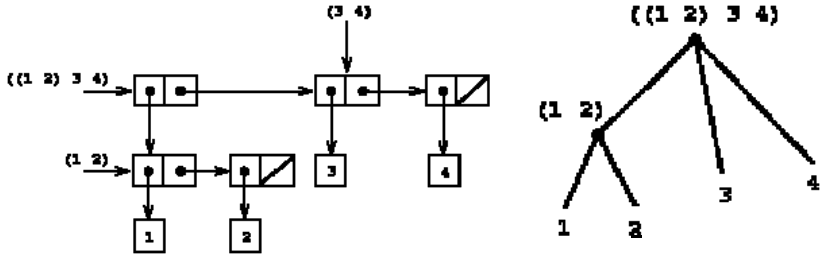
- map 很重要，它代表一种公共编程模式，还是一种表处理的高层抽象：
  - 在 scale-list 的原定义中，对元素的操作和对表元素的遍历混在一起，使这两种操作都不够清晰
  - 新定义中通过 map 抽象，对元素的操作与对表的变换（对表的遍历和作为结果的表的构造）之间很好隔离
  - 这是一种很有价值的思路
  - 下面要介绍如何把这种方式扩充为具有普遍意义的程序组织框架

层次结构

- 用表表示序列可以自然地推广到元素本身也是序列的情况，例如

  认为 ((1 2) 3 4) 是用下面表达式 (list (list 1 2) 3 4) 构造出来的

  这个表包含3个项，其中第1项又是个表。其结构如下面左图



- 这种结构可以看作树，其中的子表是子树，基本数据是树叶
- 树形结构可以很自然地用递归来处理。对树的操作可分为对树叶的具体操作和对子树的递归处理（与对整个树的操作一样，有公共模式）

层次结构

- 考虑统计树叶个数的过程 count-leaves（与 length 不同）：

```
(define x (cons (list 1 2) (list 3 4)))
(length x)
3
(count-leaves x)
4
(list x x)
(((1 2) 3 4) ((1 2) 3 4))
(length (list x x))
2
(count-leaves (list x x))
8
```

- count-leaves 可以递归地考虑：
  - 空表的 count-leaves 值是 0
  - 非序对元素的 count-leaves 值是 1
  - 非空表（序对）的 count-leaves 是其 car 和 cdr 相应的值之和

层次结构

- 对层次结构的递归，都具有这种模式
- 基本谓词 pair? 判断其参数是否序对。count-leaves 的定义：

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

count-leaves 实现一种遍历树中所有树叶、积累信息的过程。反映了一种处理多层次的表的通用技术

- 可以考虑将它推广为一般的模式（自己考虑）
- 下面要参考表映射过程 map，把对树的递归处理推广为从树到树的映射，从作为参数的树生成（计算）出另一棵与之结构相同的树

树的映射

- 树中树叶（假设是数）按 factor 等比缩放。可以用与 count-leaves 类似的方式遍历整个树，处理中构造作为结果的树：

```
(define (scale-tree tree factor)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else (cons (scale-tree (car tree) factor)
                    (scale-tree (cdr tree) factor)))))
(scale-tree  (list 1 (list 2 (list 3 4) 5) (list 6 7))  10)
(10 (20 (30 40) 50) (60 70))
```

- 还可以把树看作子树的序列，基于 map 实现 scale-tree：

```
(define (scale-tree tree factor)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
             (scale-tree sub-tree factor)
             (* sub-tree factor)))
       tree))
```

两种观点都可以提炼出实现树映射的高阶过程

请自己作为练习

序列作为一种约定的接口

- 数据抽象在复合数据处理中有重要作用：屏蔽数据的表示细节，使程序更有弹性，实现里可以采用不同的具体表示
- 另一相关问题：使用约定的接口。高阶过程可用于实现各种程序模式。但对复合数据做类似的操作，需要考虑对具体数据结构的操作
- 下面从两个例子观察这方面情况和问题，希望找到有用的抽象
- 例 1：求树中值为奇数的树叶的平方和：

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


序列作为一种约定的接口

- 例 2：构造 Fib(k) 的表，其中 Fib(k) 是偶数且 k <= n。过程：

```
(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))
```

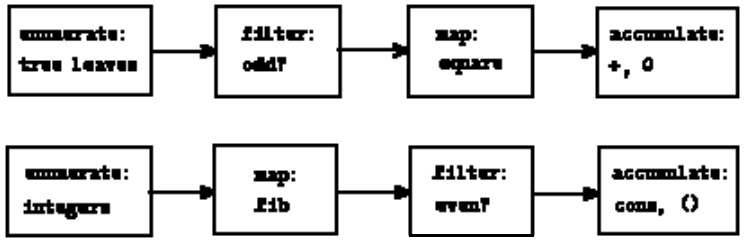
- 两个过程看起来差别很大，但相关计算的抽象描述类似：

- 枚举树中所有树叶
- 滤出其中的奇数
- 对选出的数求平方
- 用 + 累积它们，从 0 开始

- 枚举从 0 到 n 的整数
- 对每个数 k 算出 Fib(k)
- 滤出其中的偶数
- 用 cons 累积它们，从 nil 开始

序列作为一种约定的接口

- 可将两个过程的处理过程设想为串联起的一些步骤，每步完成一项具体工作，信息在步骤间流动。图示：



- 问题：前面程序都没有很好地反映这种信息流动的结构，过程实现中不同步骤交织在一起，缺乏结构性。例如对 even-fibs：

  哪些代码是枚举？map 映射？过滤？累积？

- 重组程序，使之很好反映这种信息流动，可能使程序更清晰

序列操作

- 为更好反映上述信息流结构，必须注意处理中从一个步骤向下一步骤流动的信息。用表来表示这些信息，就可以通过表操作实现各步处理
- 例如，用 map 实现信息流中的映射：

```
(map square (list 1 2 3 4 5))
(1 4 9 16 25)
```

- 对序列的过滤就是选出其中满足某个谓词的元素：

```
(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))
```

  满足谓词的元素留下，不满足的丢掉

序列操作

- 累积操作的过程实现：

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
(accumulate + 0 (list 1 2 3 4 5))
15
(accumulate cons nil (list 1 2 3 4 5))
(1 2 3 4 5)
```


序列操作

- 枚举数据序列是处理的基础。even-fibs 枚举一个区间的整数：

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high))))
(enumerate-interval 2 7)
(2 3 4 5 6 7)
```

- sum-odd-square 枚举一棵树的所有树叶：

```
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))
(enumerate-tree (list 1 (list 2 (list 3 4)) 5))
(1 2 3 4 5)
```

序列操作

- 现在可以基于这些基础设施，重新构造前面两个过程：
- sum-odd-square 重定义为：

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

- even-fibs 重定义为：

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

- 把程序表示为一系列针对序列的操作，得到了更规范的模块。这里用序列作为不同模块之间的标准接口

序列操作

- 模块化设计还能支持重用，许多程序可能通过模块拼装的方式构造
- 例：前 n+1 个斐波纳契数的平方：

```
(define (list-fib-squares n)
  (accumulate cons  nil
              (map square
                   (map fib (enumerate-interval 0 n)))))
(list-fib-squares 10)
(0 1 1 4 9 25 64 169 441 1156 3025)
```

- 例：一个序列中的所有奇数的平方的乘积：

```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate *
              1
              (map square (filter odd? sequence))))
(product-of-squares-of-odd-elements (list 1 2 3 4 5))
225
```


序列操作

- 一个数据处理问题：从人事记录里找出薪金最高的程序员的工资。假定过程 salary 返回记录里的工资，programmer? 检查是否程序员：

```
(define (salary-of-highest-paid-programmer records)
  (accumulate max
              0
              (map salary (filter programmer? records))))
```

- 启发：许多处理过程可能表示为一系列的序列操作
- 这里用表来表示序列，用作操作之间的公共接口，作为被处理信息的载体在不同的操作之间传递

  Unix 的常用工具用正文文件作为信息载体，基于标准输入和标准输出，通过管道传递。这是 Unix 优于其他 OS 的一个重要因素。问题：字符串不能很好支持复杂的信息结构

  最基础最重要的一种软件体系结构是“管道和过滤器”结构

  Scheme（Lisp）里统一的表结构，是组合复杂程序的利器

嵌套的映射

- 可以扩展序列处理的范型，例如加入嵌套循环的概念
- 例：找出所有不同的 i 和 j 使 1 <= j < i <= n 且 i + j 是素数。对 n = 6 的结果：

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

- 第一步：对每个 i <= n，枚举所有的 j < i，生成数对 (i, j)

  对序列 (enumerate-interval 1 n) 中每项 i 做 (enumerate-interval 1 (- i 1))，对这一序列中每个 j 和 i 执行 (list i j) 生成一个数对。把这样的数对序列用 append 拼接，就能得到所需的基础序列：

```
(accumulate append
            nil
            (map (lambda (i)
                   (map (lambda (j) (list i j))
                        (enumerate-interval 1 (- i 1))))
                 (enumerate-interval 1 n)))
```

- 把用 append 积累的工作定义为一个过程：

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

- 最后的过滤条件是 i + j 是否素数。定义谓词：

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

- 生成结果序列，只需定义一个过程生成 (i, j, i+j)：

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

- 把这些组合起来就得到了所需的过程

```
(define (prime-sum-pairs 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)))))
```


嵌套的映射

- 通过嵌套的映射可以生成各种序列
- 例：生成一组元素的所有排列，即生成所有排序方式的序列。考虑

  对集合 S 里的每个 x 生成 S – {x} 的所有排序的序列，而后将 x 加在这些序列的最前面，就得到以 x 开头的所有排序序列

  把对 S 中每个 x 生成的以 x 开头的序列连起来，就得到所要的结果

-

```
(define (permutations s)
  (if (null? s)                    ; empty set?
      (list nil)                   ; sequence containing empty set
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (remove x s))))
               s)))
```

-

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


关注

- 数据抽象

  以一组基本操作作为接口，操作应满足某些关系

- 序对和 cons, car, cdr
- 表，表操作，map
- 一般的序对结构和树映射
- 用序列作为组织程序的约定接口
- 顺序处理的步骤还不是很清晰，可考虑定义 pipeline，其用法是

```
(pipeline operand op1 op2 ... opn)
```

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