2. 构造数据抽象(1)

本章关注数据抽象，这一节讨论

- 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
- 把序列用作程序模块之间的接口

数据抽象的意义

- 前面讨论过程时只考虑简单的数据
  - 解决复杂问题，处理和模拟复杂现象的时候，常需要构造和处理复杂的计算对象
- 第二章关注具有复杂结构的数据的计算，讨论
  - 构造复合数据对象（通过数据的组合）
  - 处理复合数据对象
- 与构造复合过程类似，构造复合数据也能
  - 提高编程的概念层次（可能在比基本数据更高的层面上工作）
  - 提高设计的模块性（基于复合数据来组织程序，很重要）
  - 增强语言的表达力（增加了“新”数据类型）
  - 为处理计算问题提供更多手段和方法
- 下面用一个简单问题讨论有关情况

有理数计算

- 假设要实现过程 **add-rat**，计算两个有理数之和。在基本数据层，一个有理数可以看作两个整数。可以考虑实现两个过程
  - **add-num** 计算结果的分子
  - **add-den** 计算结果的分母
- 这种做法显然很不理想：
  - 如果有多个有理数，记住成对的分子和分母是很大麻烦
  - 相互分离的两个调用很容易写错
  - 所有运算的实现/使用都有同样问题
- 应该把一个有理数的分子分母粘在一起，做成复合数据（一个整体）
  - 有了复合数据对象，就能在更高概念层上定义和使用操作（处理有理数，而不是处理两个整数），更清晰，更容易理解和使用
  - 数据抽象的定义（表示和操作实现细节）与使用分离，提高了程序模块性。两边都可以独立演化，更容易维护修改

数据抽象的意义

- 实现数据抽象，编程语言需要提供：
  - 粘合机制，支持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：组合数据粘合机制应该不仅能用于基本数据，也能用于复合数据，以支持构造更复杂的复合数据
- 本章还要讨论：
  - 复合数据如何支持以“匹配和组合”方式工作的编程接口
  - 通过定义数据抽象，进一步模糊“过程”和“数据”的差异
  - 符号表达式的处理，这种表达式的基本部分是符号而不是数
  - 通用型（泛型）操作，使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，方便新数据类的加入

数据抽象入门

- 一个过程描述了一类计算的模式，又可以作为元素用于实现其他（更复杂的）过程。因此过程是一种抽象，过程抽象
  - 屏蔽计算的实现细节，可以用任何功能/使用形式合适的过程取代
  - 规定了使用方式，使用方只依赖于抽象的使用方式约定
- 数据抽象的情况类似。一个数据抽象实现一类数据所需的所有功能，又像基本数据元素一样可以作为其他数据抽象的元素
  - 屏蔽一种复合数据的实现细节
  - 提供一套抽象操作，使用组合数据的就像是使用基本数据
  - 数据抽象的接口（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，选择函数提取其内容
- 后面将说明，如果需要支持基于状态的程序设计，那么就需要增加另外一类变动操作（**mutation**，修改操作）
- 下面以有理数为例讨论数据抽象的构造

有理数算术

- 为了使用有理数功能，需要有一个基于分子和分母构造有理数的过程，以及分别提取分子和分母的过程。设它们分别是：

  (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

  这三个过程构成有理数数据抽象的接口

- 有理数的计算规则：

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

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

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

有理数算术

- 很容易基于有理数数据抽象，定义实现有理数算术的过程（虽然现在还没考虑有理数的实际表示和基本操作实现）：

```
(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))))
```

在没有考虑具体实现之前先考虑基本过程的使用，是非常好的设计方法

通过这些基本使用可以检验数据抽象的基本过程的设计是否合理、有效

序对

- 还要实现有理数以及它的几个基本操作

  首先必须有办法把分子和分母结合为一个整体，构造出有理数

- Scheme 的基本复合结构称为“序对”，基本过程 cons 把两个参数结合起来构造一个序对，过程 car 和 cdr 取出序对的两个成分

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

- 序对也是数据对象，可以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

- 可以直接用序对表示有理数。基本过程的可行定义：

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

输出有理数的过程（display 是一个基本输出函数）

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

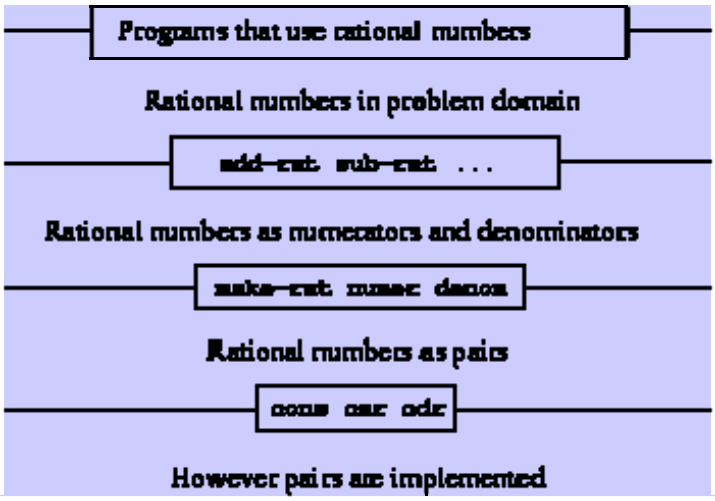
- 为处理方便，可以考虑把有理数都化简到最简形式，使分子分母达到最小，简化相等判断。修改定义

```
(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：这一修改对有理数使用没有任何影响，这也说明了抽象的价值）

抽象屏障

- 总结一下有理数算术系统，问题和相关思想
  - 运算都基于基本过程 make-rat、numer 和 denum 实现
  - 一般而言，实现数据抽象时要首先确定一组基本操作，其余操作都基于它们实现，不直接访问基础数据表示
- 有理数系统的结构，注意各层次间的抽象屏障：



抽象屏障

- 建立层次性抽象屏障的价值：
  - 数据表示和使用隔离，两部分可以独立演化，容易维护和修改
  - 实现好的数据抽象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
- 复杂的数据抽象可能有多种实现方式，各有特点。不同选择的影响经常要在开发进行了一段后才能看清，抽象屏障可大大降低修改的代价
- 例如，把有理数最简化可以推迟到访问时再做，得到另一实现：

```
(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)))
```

对有理数而言这种实现未必好。但许多时候不同实现的优劣并不清晰。究竟那种技术更优，要根据具体情况考虑。例子：

链表是否显式表示元素个数？

实例：区间算术

- 考虑实现一个工程问题辅助求解系统，做不精确物理量（如测量值）的计算。参数包含已知误差，结果也应该是包含误差信息的数值
- 例如，电子工程师用下面公式计算并联电阻的阻值

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

电阻通常标注为“xxxΩ 误差 10%”

- 考虑实现一套区间值运算，首先需要“区间”数据对象。例如
  - 构造函数 make-interval
  - 选择函数 lower-bound 和 upper-bound
- 在此基础上，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

- 乘法实现为上下界的最小和最大可能值构成的区间：

```
(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))))
```

min 和 max 是基本过程，求任意多个参数中的最小值/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)))))
```

练习中提出了一些与区间表示和基本操作实现有关的问题

实例：区间算术

- 假设用户又提出需要处理以“数值加误差”形式表示的数据。由于有数据抽象，很容易加入新构造函数和选择函数

```
(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

  这些都说明了抽象的价值（支持系统的扩展/修改/变化）

- 练习要求分析和改进这个系统

  提出了一些问题和建议，请参考，自己做做

  这样做下去，可以完成一个功能完整的区间计算系统

数据是什么

- 在考虑有理数的实现时
  - 各种有理数运算都基于三个当时没定义的过程，基于数据对象（分子/分母/有理数）的情况定义
  - 有理数对象就是由三个基本过程刻画的，没看到“数据”。
  - 这提出了一个问题：“数据到底是什么”
- 首先，不是任意三个过程都构成有理数的实现。正确实现要满足

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

  任一组满足这一条件的三个函数，都能作为有理数表示的基础

  一般说，一种数据对象的构造函数和选择函数都要满足一组条件
- 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　有前提，**x** 必须是序对

数据是什么

- 理论结果：
  - 只需要有过程就可以定义出序对，不需要用任何数据结构
  - 对其他数据对象，也同样可以做到
  - 例如，只需过程就可以第一有理数数据抽象
- 计算机科学先驱 **Alonzo Church** 研究 λ 演算证明了这个结论，他只用λ表达式（相当于完全基于过程）构造了整数算术系统

数据是什么

- 在 Scheme 里可以完全基于过程定义序对，如下面过程定义

```
(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

（书上引进局部过程，并不必要。error 结束执行并输出各参数）

- 这种序对表示满足序对构造函数和选择函数的所有条件，可以用

  但实际 Scheme 系统用存储表示直接实现序对，主要是为了效率

- 本例说明：过程和数据之间没有绝对界限，完全可以用过程表示数据，用数据表示过程。后面将会看到这样做的实际价值

常规语言里的数据抽象

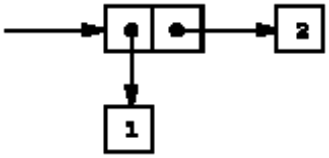
- 常规语言近 30 年的最重要发展就是数据抽象机制
  - Modula、Modula-2、Ada 等研究和发展了数据抽象概念和机制
  - 由 Simula/Smalltalk 发展起来的面向对象思想，从 C++ 开始进入常规语言。面向对象：支持以递增方式构造数据抽象
  - 早期有些语言只支持构造具体的数据抽象，后来的语言都支持构造数据抽象的类型，如 C++ 等的类也是类型
- C 语言没有支持数据抽象的专门机制，但可通过技术来模拟：
  - 用结构作为数据成分的粘结机制
  - 定义一组相关操作，作为被抽象的数据和外界的接口。可以通过头文件，代码文件里的 static 函数区分接口函数和内部函数
  - 通过编程技术，隐蔽数据抽象的实现细节

    技术基础：指针/不完全 struct 定义/typedef/动态存储分配等

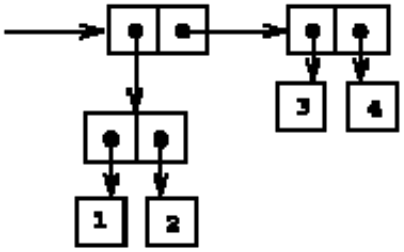
层次性数据和闭包

- 序对常用图示，右图表示 (cons 1 2)

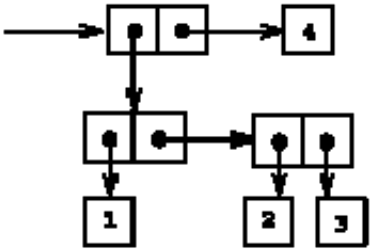
  称为盒子指针表示



- cons 也可以组合复合数据。组合 1,2,3,4 的两种不同方式：



(cons (cons 1 2)
(cons 3 4))

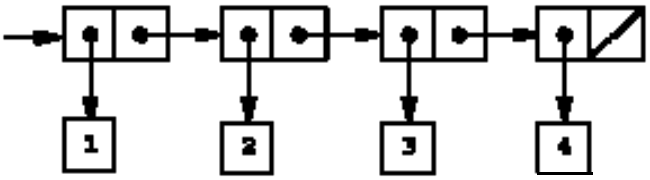


(cons (cons 1
(cons 2 3))
4)

- 任何序对结构都可作为 cons 的参数（cons 的闭包性质，序对的 cons 还是序对）。只有 cons 就可以构造结构任意复杂的数据对象

序列的表示

- 用序对构造的最常用结构是序列：一批数据的有序汇集
- 一般用最直接的方式。表示包含元素 1, 2, 3, 4 的序列的结构：



构造本序列的表达式：

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

nil 是特殊变量，它不是序对，当作空序列（空表）

- 使用 cons 顺序构造起来的序列称为表。list 是构造表的过程：

(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

- 表输出为带括号的元素序列。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)

  前者是表达式，后者是表达式求值的结果

- car 取出表是第一项，cdr 取得去掉第一项后其余项的表：

```
(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)
```


序对和表

- 注意序对和表的不同。设

```
(define x (cons 1 2))
(define y (list 1 2))
```

- 这时有：

  (car x) → 1　　(cdr x) → 2

  (car y) → 1　　(cdr y) → (2)

- x 和 y 的图示（指针盒子表示）也不同

  表是通过序对的 cdr 连接的，以 nil 结尾的盒子序列

- 注意：序对和表都需要在内存里安排存储
  - 每次 cons 都要做一次动态存储分配
  - 做表操作时可能导致一些序对单元失去引用。Scheme 系统有内置的废料收集系统，能自动回收这种单元

表操作：取元素

- 一个序列表示一组元素，不断求 **cdr** 就能顺序得到表中的内容
- 定义过程 **list-ref** 返回表 **items** 中第 **n** 项元素（**n** 是 **0** 时返回首项）

  递归考虑：**n** 是 **0** 时返回表的 **car**，否则返回表的 **cdr** 的第 **n-1** 项
- 基于上面想法的过程定义：

```
(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** 太大（大于表元素个数）或小于 **0**，过程出错（执行中要对非序对对象求 **cdr**，报告错误）

  一般的，在不断求 **cdr** 的程序中应该判断是否遇到空表

表操作：求表长度

- 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
- 过程定义：

```
(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 **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
- **append** 的定义：

```
(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

  我们也可以定义这种过程，只需用带点尾部记法的参数表：

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

  圆点之前可以根据需要写多个形参，它们将一一与实参匹配。圆点后写一个形参，应用时关联于其余实参的表

- 举例：下面是一个求任意多个数的平方和的过程：

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

- 如果需要处理的是 **0** 项或任意多项，参数表用 **(square-sum . y)**，过程体也需要相应修改

其他语言里的表

- 常规过程性语言都没有内部的表数据类型
  - C、Pascal 等语言开始考虑对表结构的支持

    需要自己编程实现（数据结构课基本内容之一）

    基础机制包括结构、指针、动态存储分配等
  - 注意：Scheme 不限制表的元素类型（是“泛型”的，支持任何元素类型的表，而且允许混合类型的表）。在常规语言里实现表，规范方式是只允许以同一类型的数据为元素
- 一些 OO 语言通过标准库提供这类结构
  - C++ 的标准 STL 库里的 list，Java 的标准库
  - 基于“继承”，可以以一组相关类型为元素
- 一些脚本语言提供表作为基本数据机制（如 Python），多数数学软件（如 Maple、Mathematica 等）以表作为重要数据机制

其他语言里的表

- 表及其相关概念是从 Lisp 开始开发
  - 动态存储管理已经成为日常编程工作的基本支持
  - 链表的定义和使用是常用技术
  - 有关表的使用和操作，以及各种操作的设计和实现，都可以从 Lisp 的表结构学习许多东西
  - 在设计实现表数据结构时，这里的讨论都值得参考
  - 注意下面讨论的高阶函数 map 等，注意其思想和技术
  - 高阶表操作对分解程序复杂性很有意义

表的映射

- 讨论一类重要表操作：把某过程统一应用于表元素，得到结果的表
- 例：构造把所有元素等比例放大或缩小的表

```
(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)
```

- 基本模式
  - 对每个元素都采用同样操作得到结果表的元素
  - 结果表的结构（元素个数）与原表一样
  - 对应元素的顺序也保持不变

表的映射

- 总结这一计算的模式，可以得到一个重要的（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)
```

- **map** 把处理元素的过程应用于作为其参数的表里的每个元素
  - 得到的是通过该过程的应用得到的所有新元素的表
  - 可以看作操作的“提升”：把元素层的映射（由一个元素得到另一新元素）提升为表层的操作（由一个表得到另一新表）

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，它是一类表处理的高层抽象，代表一种公共编程模式：
  - 在 **scale-list** 原定义里，元素操作和对表元素的遍历交织在一起，使这两种操作都不够清晰
  - 在新定义里，通过使用 **map** 抽象分离了对元素的操作和对表的变换（对表的遍历和作为结果的表的构造）

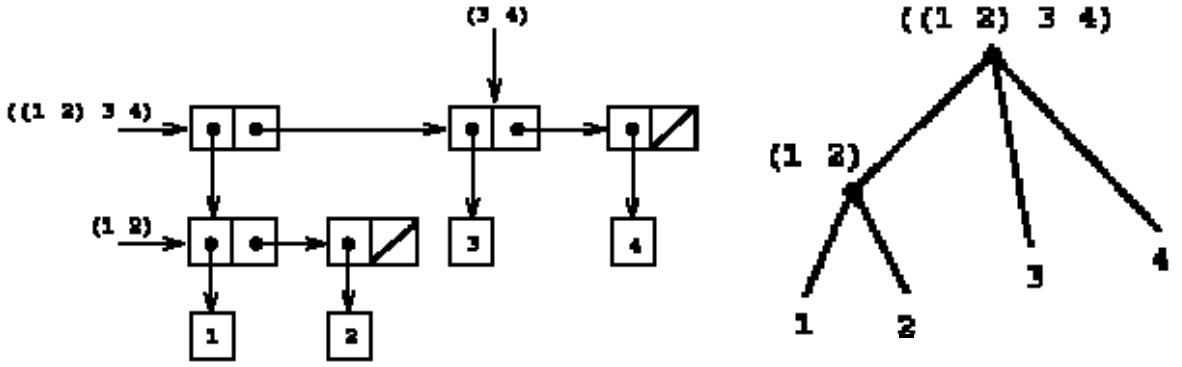
    两部分都可以独立变化，可以换一个操作或者换一种操作模式
  - 这是一种很有价值的思路。下面要讨论如何把这种方式扩充为一种具有普遍意义的程序组织框架

层次结构

- 用表表示序列，可以自然推广到元素也是序列的情况，如

  把 **((1 2) 3 4)** 看作是用 **(list (list 1 2) 3 4)** 构造的

  表里有 **3** 项，其中第 **1** 项又是表。结构如下图



- 这种结构可以看作是树，其中的子表是子树，基本数据是树叶

  树形结构可以自然地用递归方式处理

  树操作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，具有同样的处理模式）

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
```

- 递归地考虑 **count-leaves** ：
  - 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

- 如果需要对层次性结构递归，都可以用这种计算模式

  递归过程中需要判断是否到达了树叶

  基本谓词 **pair?** 可以判断是否序对

- **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表的通用技术。可以考虑推广为一般模式（请考虑）
- 下面采用表映射过程 **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）出另一棵结构相同的树

树的映射

- 例：把树叶（假设是数）按 factor 等比缩放。可以用 count-leaves 的方式遍历树，在遍历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))
```

- 把树看作子树序列，也可以基于 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

以序列作为程序接口

- 数据抽象在复合数据处理中起着重要作用
  - 屏蔽数据的表示细节，使程序更具灵活性（易维护、易修改）
  - 具体实现可以采用不同的具体表示
- 相关问题：使用约定的接口。可以用高阶过程实现各种程序模式
  - 对复合数据做类似操作时，需要考虑具体数据结构的操作
  - 下面基于两个例子考察相关情况和问题，寻找有用抽象
- 例 1：求一棵树里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：

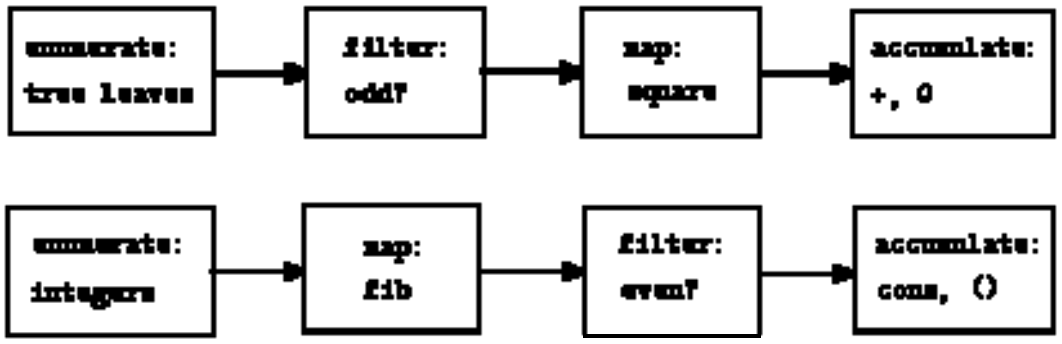
```
(define (even-fibs n)
    (define (next k)
      (if (> k n)
          nil
          (let ((f (fib k)))
            (if (even? f)
                (cons f (next (+ k 1)))
                (next (+ k 1))))))
    (next 0))
```

- 两个过程看起来很不同，但相关计算的抽象描述却类似：

❖ 枚举树中所有树叶	➢ 枚举从 0 到 n 的整数
❖ 滤出其中的奇数	➢ 对每个数 k 算出 Fib(k)
❖ 对选出的数求平方	➢ 滤出其中的偶数
❖ 用 + 累积它们，从 0 开始	➢ 用 cons 累积它们，从 nil 开始

序列作为一种约定的接口

- 两过程实现的处理都可以看作串联起的一些步骤，每步完成一项具体工作，信息在步骤之间流动：



- 问题：前面的程序都没有很好地反映这种信息流动的结构，过程的实现中不同步骤交织在一起，缺乏结构性。

  以 even-fibs 为例：哪些代码是枚举？map 映射？过滤？累积？

- 如果基于上面对数据流动情况的认识组织程序，反映其中的信息流动，有可能把程序改造得更清晰

序列操作

- 为更好反映前面看到的信息流结构
  - 需要注意表示和处理从一个步骤向下一步骤流动的信息
  - 表适合表示和传递这些信息，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

基于这组基础设施，很容易重新构造前面两个过程：

- 重新定义的 **sum-odd-square**：

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

- 重新定义的 **even-fibs**：

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

- 把程序表示为针对序列的一系列操作，得到的模块更规范。这里实际上是用序列（表）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，用模块拼装的方式可以构造出许多程序
- 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

- 问题：从人事记录里找出薪金最高的程序员的工资。假定 **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串序列操作
- 这里用表表示序列，作为操作间的公共接口，作为被处理信息的载体，在不同操作之间传递

  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**）里统一使用表结构，可用于组合各种复杂的程序

嵌套的映射

- 全面想法可以扩展为序列处理的范型，例如加入嵌套循环
- 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，枚举所有满足 j < i 的数对 (i, j)

对 (enumerate-interval 1 n) 中每项 i 做 (enumerate-interval 1 (- i 1))，对该序列中每对 j 和 i 用 (list i j) 生成一个数对，最后用 append 拼接这些数对的序列就得到了所需的基础序列：

```
(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))))
```

- 组合起来就可以得到所需的过程

```
(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 里每个 x 生成的以 x 开头的序列连起来，就得到了结果

```
(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))
```


回顾

- 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

- 如认为顺序处理的步骤还不够清晰，可以考虑定义 pipeline，用法是

  (pipeline operand op1 op2 ... opn) ；任意多个参数

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