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

- 第一章讨论过程时只考虑了简单的数据
  - 在解决复杂问题，处理和模拟复杂的现象时，通常需要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的手段和方法
- 下面用一个简单问题讨论这方面的情况

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。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
- 这种做法显然不理想：
  - 如有多个有理数，记住对应的分子和分母实在太麻烦
  - 相互分离的两个调用容易写错
  - 更多运算的实现/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，做成一个复合数据（单位）
  - 有复合数据对象，就能在更高概念层次上定义和使用操作（是处理有理数而不是两个整数），更清晰、更易理解和使用
  - 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)))
```

  这里用的是具体的数值运算

- 如果想表述线性组合的一般思想，希望用于各种数据，过程是：

```
(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，只要求具体的数据支持 **add** 和 **mul**，基于它们完成对数据的适当操作

- 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，程序语言需要提供：
  - 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
- 本章将讨论问题：
  - 复合数据可以支持以“匹配和组合”方式工作的编程接口
  - 定义数据抽象将进一步模糊“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，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，易于加入新数据类

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
  - 屏蔽了计算的实现细节，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
- 后面将说明，要支持基于状态的程序设计，还需要增加一类操作：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

- 要使用有理数，需要有基于分子和分母构造有理数的过程，以及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：

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} \qquad \frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

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

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

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

- 需要实现有理数的几个基本操作

  首先必须能把分子和分母结合为一个整体，构成一个有理数

- **Scheme** 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

- 可以直接用序对表示有理数，有基本过程定义：

```
(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)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

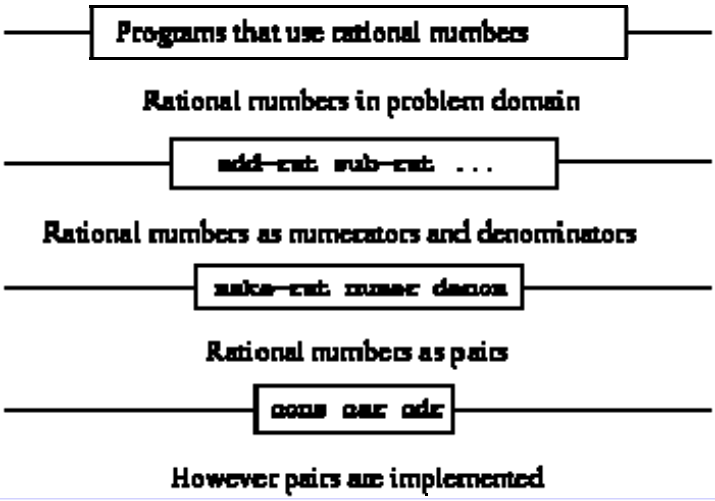
- 为处理方便，可考虑把有理数都化简到最简形式，这样分子分母的值达到最小，相等判断谓词可以简化。修改定义

```
(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，也说明了抽象的价值）

抽象屏障

- 总结有理数算术系统，工作中的问题和相关思想
- 有理数运算都基于基本过程 **make-rat**、**numer** 和 **denum** 实现。一般说，实现数据抽象时需首先确定一组基本操作，其余操作都基于这些基本操作实现，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

- 建立层次性的抽象屏障的价值：
  - 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

- 有理数实现中，有理数运算是基于三个开始并无定义的过程，根据被操作的数据对象（分子/分母/有理数）的需要。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
- 并不是任意三个过程都构成有理数的实现。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**

  （书上定义引进局部过程，并无必要）
- 这种序对表示满足序对构造函数和选择函数的所有条件，完全可用

  Scheme 用存储直接实现序对，主要为了效率
- 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年的重要发展就是数据抽象机制
  - 语言 Modula、Modula-2、Ada 等提供数据抽象机制
  - 从 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的方式构造数据抽象
  - 早期的一些语言只支持构造数据抽象，后来的语言都支持构造支持数据抽象的类型，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
  - 可通过编程技术，隐蔽数据抽象的实现细节

    支持有关技术的基本语言结构包括指针，不完全的 struct 定义，typedef，动态存储分配等

实例：区间算术

- 考虑另一实例：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。参数已知误差，结果应是知道误差的数值
- 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%”

- 要实现一套区间值运算，需要“区间”数据对象。为此需要
  - 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))))
```

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

- 假设用户后来提出希望能处理“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))
```

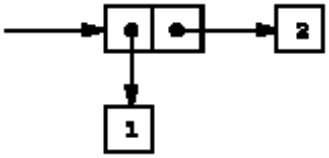
- 一些工程师希望处理由数值加百分比误差表示的数据。不难增加新的构造和选择函数满足这种需求
- 这些都说明了抽象的价值（支持系统的扩展/修改/变化）
- 练习要求分析和改进这个系统

  提出了一下问题和建议

  这样做下去，可以完成一个功能完整的区间计算系统

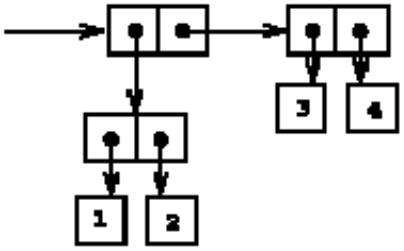
层次性数据和闭包

- 序对是构造复合数据的基本粘合机制。常用图形式表示，右图表示 **(cons 1 2)**

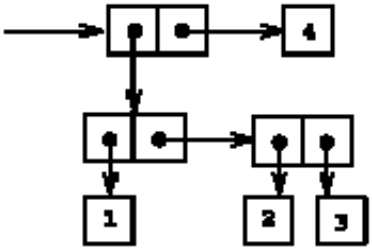


这种图示称为盒子指针表示

- **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))

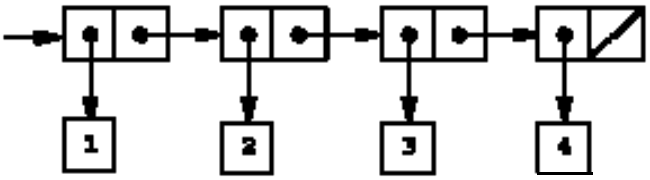


(cons (cons 1
(cons 2 3))
4)

- 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）。只需 **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，直接方式如下，这个序列包含元素 **1, 2, 3, 4**



构造本序列的表达式：

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

nil 是特殊变量，它绝不是序对，当作空序列（空表）

- 用 **cons** 构造出的序列称为表。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

- 表被输出为加括号的元素序列。例如

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

- 注意区分 **(list 1 2 3 4)** 和输出的 **(1 2 3 4)**

  分别是表达式，和表达式求值的结果

- 对于表，**car** 取其第一项，**cdr** 取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)
```


序对和表

- 应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** 的图示自然也不一样
  - 表的通过 **cdr** 连接起来的，以 **nil** 结尾的盒子序列
- 注意：表示序对和表，需要在内存中安排它们的存储
  - 实际上，每次 **cons** 都需要做动态存储分配
  - 在做各种表操作时，可能导致一些序对单元失去引用。**Scheme** 系统包含内置的废料收集系统，能自动把这些单元回收重用

表操作：取元素

- 序列表示一组元素，不断求 cdr 能顺序得到对表中的内容
- 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 值过大（大于表元素个数）或小于 0，上面过程会出错（对非序对的对象求 cdr，将出错）

  一般说，在不断找 cdr 时应判断是否遇到空表

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** 过程定义：

```
(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
- 可以自己定义这类过程，用带点尾部记法的参数表：

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

圆点前根据需要写任意多个形参，它们将与实参一一匹配。圆点后的一个形参在应用时关联于其余实参的表

- 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。作为自己的课下练习

其他语言里的表

- 常规过程性语言里没有内部的表数据类型
  - 如 C、Pascal 语言等
  - 只能通过自行编程来实现（数据结构课基本内容之一）
  - 实现支持机制包括结构、指针、动态存储分配等
  - 注意：Scheme 并不限制一个表里元素的类型（是“泛型”的，支持任何元素类型的表，且允许混合类型的表），而常规语言里实现的表，按规范的方式只能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
- 一些脚本语言提供了表作为基本数据机制，多数数学软件（如 Maple、Mathematica 等）都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
  - 已经成为常规编程工作的基本技术手段
  - 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

- 现在讨论一类重要表操作：把某过程统一应用于表中元素得到结果表

```
(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)
```

- 总结这一计算中的模式，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)
```


表的映射

- 用 **map** 给出 **scale-list** 的定义：

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

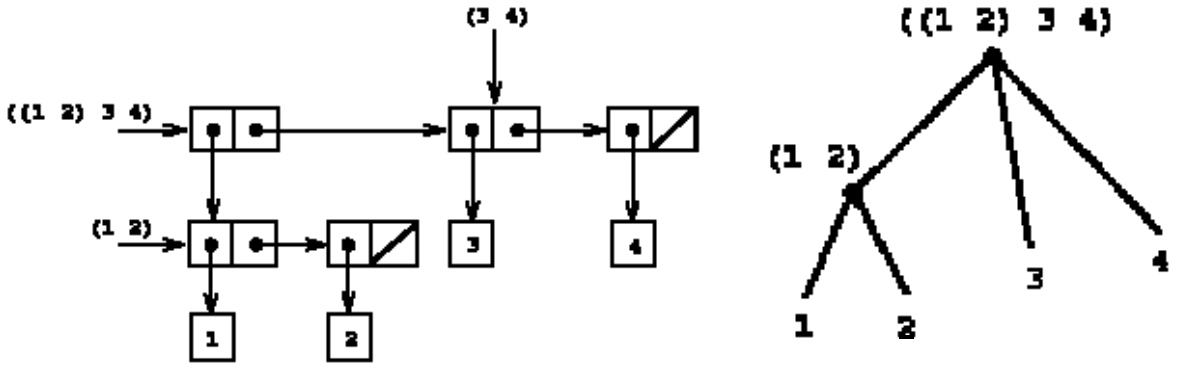
- **map** 很重要，是一种表处理的高层抽象，代表一种公共编程模式：
  - 它把对元素的映射（计算）提升为对整个表的映射
  - 在 **scale-list** 的原定义中，元素操作和对表元素的遍历混在一起，使这两种操作都不够清晰
  - 新定义中通过 **map** 抽象，使元素操作和对表的变换（对表的遍历和作为结果的表的构造）得到很好的分离
  - 这是一种很有价值的思路
  - 下面讨论如何把这种方式扩充为具有普遍意义的程序组织框架

层次结构

- 用表表示序列可以自然推广到元素本身也是序列的情况，如

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

  包含3个项，其中第1项又是个表。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通过递归处理。树的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，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，都可以用这种计算模式
- 在这种递归过程中需要判断是否到达树叶。**Scheme** 的基本谓词 **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

- 例：将树叶（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，就可以基于 **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

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（易维护、易修改）
  - 实现时可以采用不同的具体表示
- 相关问题：使用约定的接口。可用高阶过程实现各种程序模式
  - 但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：

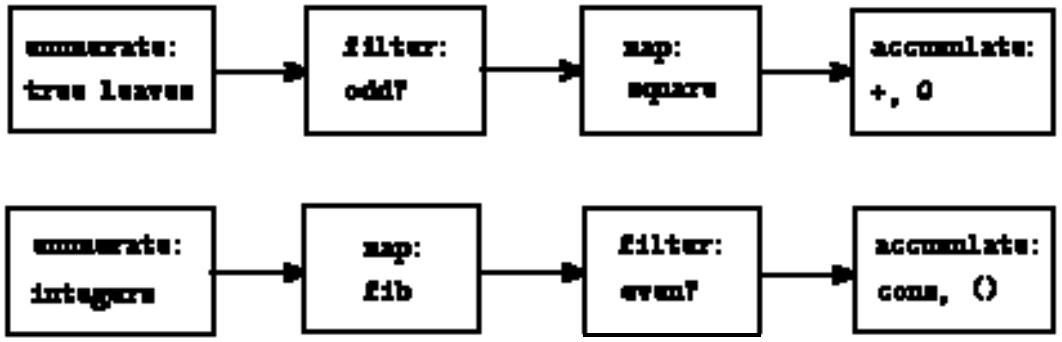
```
(define (even-fibs n)
    (define (next k)
      (if (> k n)
          nil
          (let ((f (fib k)))
            (if (even? f)
                (cons f (next (+ k 1)))
                (next (+ k 1))))))
  (next 0))
```

- 两个过程看起来差别很大，但相关计算的抽象描述很类似：

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

- 两个过程的处理过程都可以看作是串联的一些步骤，每步完成一项具体工作，信息在步骤间流动：



- 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？

- 重新组织程序，反映这种信息流动，有可能使程序变得更清晰

序列操作

- 为更好反映上述信息流结构
  - 应注意表示和处理从一个步骤向下一步骤流动的信息
  - 用表可以方便地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：
- 重定义的 sum-odd-square：

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

- 重定义的 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，得到了更规范的模块。这里用序列（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，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系列的序列操作
- 这里，用表表示序列，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**）里统一使用的表结构，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))))
```

- 组合即可得到所需过程

```
(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))
```

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

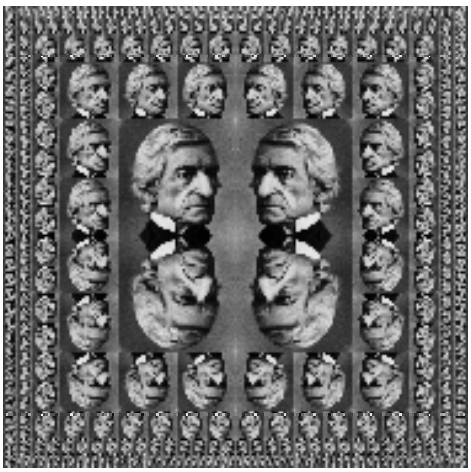
- 如认为顺序处理的步骤还不够清晰，可以考虑定义 **pipeline**，用法是

  (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)) ))
```


一个图形语言

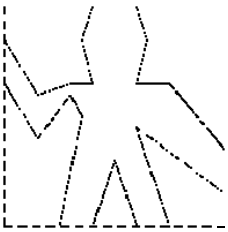
- 以一个构造图形的简单语言为例，展示数据抽象和闭包的作用和威力
  - 其中高阶过程起了关键作用
  - 主要功能：构造重复元素的图形，元素可以按规则变形变大小
- 两个这种图形的例子：



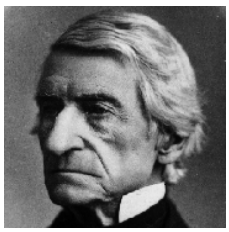
有点像 **Escher** 画的图（当然，远没有那些画复杂、深刻）

图形语言：基本想法

- 基本元素：**painter**。一个 **painter** 画出一种特定图像
  - 可根据要求对所画图像进行操作（改变大小和变形）
  - 画出的图像依赖于具体框架。例如：



wave 画的图



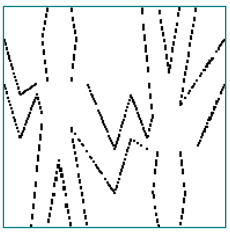
rogers 画的图

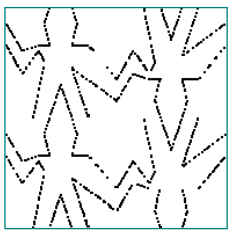
- 图像组合操作：（假设定义了几种组合操作，具体实现见后）
  - **beside** 使用两个 **painter**，让它们分别在左右半区域画图
  - **below** 使用两个 **painter**，让它们分别在上下半区域画图
  - **flip-vert** 使用一个 **painter**，画出上下反转后的图
  - **flip-hozil** 使用一个 **painter**，画出左右反转后的图

图形语言：组合

- **painter** 的组合还是 **painter**，例：

```
(define wave2 (beside wave (flip-vert wave)))
(define wave4 (below wave2 wave2))
```





- 从一个过程中可能抽取出几个不同模式
- 如 **wave4**，可考虑将反转方式抽取出来作为参数
- 可有其他考虑

- 应考虑 **painter** 的重要组合模式，并将其实现为**Scheme**过程
- 例如，抽象出 **wave4** 里的模式：

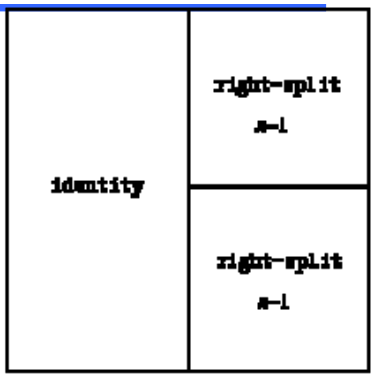
```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))

(define wave4 (flipped-pairs wave))
```

定义好的操作可用于任何 **painter**

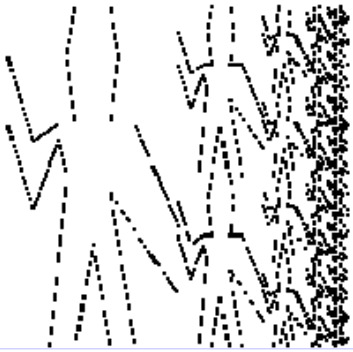
图形语言：组合

- 向右分割和分支：

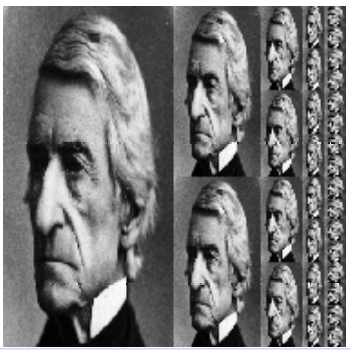
```
(define (right-split painter n)
    (if (= n 0)
        painter
        (let ((smaller (right-split painter (- n 1))))
          (beside painter (below smaller smaller)))))
```



(right-split wave 4)



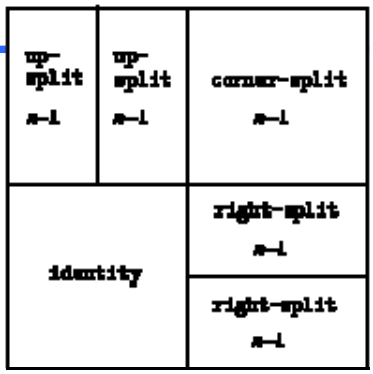
(right-split rogers 4)



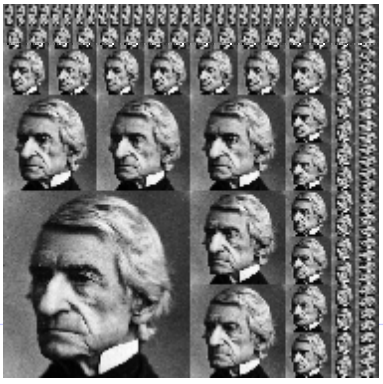
图形语言：组合

- 向右上角分割和分支：

```
(define (corner-split painter n)
    (if (= n 0)  painter
        (let ((up (up-split painter (- n 1)))
              (right (right-split painter (- n 1))))
          (let ((top-left (beside up up))
                (bottom-right (below right right))
                (corner (corner-split painter (- n 1))))
            (beside (below painter top-left)
                    (below bottom-right corner))))))
```



up-split 与 right-split 类似



(corner-split wave 4)

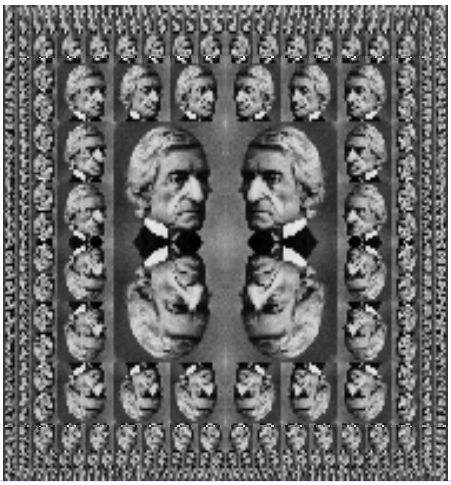
(corner-split rogers 4)

还可以定义更复杂的图形组合过程

图形语言：组合

- 把四个 **corner-split** 按适当方式组合，定义下面 **square-limit**，就可以生成本节开始的两个图形：

```
(define (square-limit painter n)
      (let ((quarter (corner-split painter n)))
        (let ((half (beside (flip-horiz quarter) quarter)))
          (below (flip-vert half) half))))
```



(square-limit rogers 4)

图形语言：高阶操作

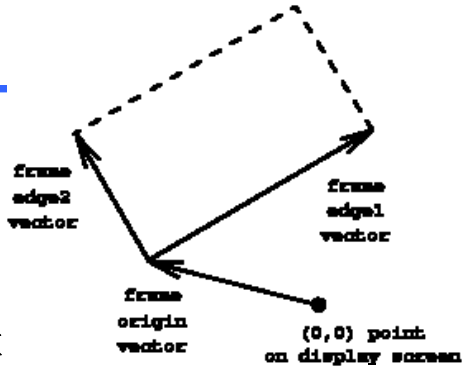
- 前面对 **painter** 的组合模式进行抽象定义了几个过程。同样可以对组合操作进行抽象定义各种高阶过程，它们以对 **painter** 的操作作为参数，产生对 **painter** 的新操作
- 例：**flipped-pairs** 和 **square-limit** 都是将原区域分为4块，而后按不同变换方式摆放四个部分的图像。把这 **4** 个变换抽象为过程参数：

```
(define (square-of-four tl tr bl br)
      (lambda (painter)
        (let ((top (beside (tl painter) (tr painter)))
              (bottom (beside (bl painter) (br painter))))
          (below bottom top))))
```

- 重定义 **flipped-pairs**：

```
(define (flipped-pairs painter)
      (let  ((combine4 (square-of-four identity flip-vert
                                       identity flip-vert)))
        (combine4 painter)))
```

图形语言：框架



现在考虑 **painter** 本身及其技术基础

- 图像的显示框架可以用3个向量表示：
  - 基准向量描述框架基准点的位置
  - 两个角向量描述两个相邻角的相对位置
- 设有框架构造函数 **make-frame**，选择函数 **original-frame, edge1-frame** 和 **edge2-frame**

需要一个映射由给定 **frame** 生成一个过程，它由向量参数生成所需向量：

```
(define (frame-coord-map frame)
    (lambda (v)
      (add-vect
        (origin-frame frame)
        (add-vect (scale-vect (xcor-vect v)
                              (edge1-frame frame))
                  (scale-vect (ycor-vect v)
                              (edge2-frame frame))))))
```

- 平面向量 **v** 有两个分量
- 生成过程用 **frame** 变换给定向量（移基点+两方向的比例变换）

图形语言：painter

- 一个 **painter** 是一个过程，它以一个框架为参数，通过适当位移缩放，把它要画的图形嵌入由参数给定的框架里
- 基本 **painter** 的实现依赖具体图形系统和被画图像的种类。如，假定有画直线的基本过程 **draw-line**，折线图形中的折线用线段的表表示，用下面过程可以画出各种折线图：

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
      (lambda (segment)
        (draw-line
          ((frame-coord-map frame) (start-segment segment))
          ((frame-coord-map frame) (end-segment segment))))
      segment-list) ) )
```

实现某种线段表表示（也是数据抽象），给出 **wave** 图形的线段表，就可以用 **segments->painter** 定义出 **wave**

图形语言：painter，变换和组合

- 用过程表示 painter，建立了良好的抽象屏障
  - 容易创建基本 painter，容易通过组合构造复杂 painter
  - 任何以框架为参数，基于它画图的过程都可作为 painter
- 对 painter 的操作都是创建新 painter，如 flip-vert 和 beside，其中用到作为参数的 painter，还涉及框架变换
- 对 painter 的操作都基于 transform-painter 定义。它以 painter 和变换框架的信息为参数，基于变换后的框架调用原 painter。框架变换信息用三个向量描述，分别表示新基准点和两个边向量的终点

```
(define (transform-painter painter origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter
          (make-frame new-origin
                      (sub-vect (m corner1) new-origin)
                      (sub-vect (m corner2) new-origin)))))))
```


图形语言：变换和组合

- 各种 painter 变换都可以基于过程 transform-painter 定义
- 纵向反转 flip-vert ：

```
(define (flip-vert painter)
    (transform-painter painter
                       (make-vect 0.0 1.0)   ; new origin
                       (make-vect 1.0 1.0)   ; new end of edge1
                       (make-vect 0.0 0.0))) ; new end of edge2
```

- 将框架收缩到原区域的右上四分之一区域：

```
(define (shrink-to-upper-right painter)
    (transform-painter painter (make-vect 0.5 0.5)
                       (make-vect 1.0 0.5) (make-vect 0.5 1.0)))
```

- 将图形逆时针旋转 90 度：

```
(define (rotate90 painter)
    (transform-painter painter (make-vect 1.0 0.0)
                       (make-vect 1.0 1.0) (make-vect 0.0 0.0)))
```

图形语言：变换和组合

- 将图像向中心收缩：

```
(define (squash-inwards painter)
    (transform-painter painter (make-vect 0.0 0.0)
                               (make-vect 0.65 0.35) (make-vect 0.35 0.65)))
```

- beside 以两个 painter 为参数：

```
(define (beside painter1 painter2)
    (let ((split-point (make-vect 0.5 0.0)))
      (let ((paint-left
             (transform-painter painter1 (make-vect 0.0 0.0)
                                         split-point (make-vect 0.0 1.0)))
            (paint-right
             (transform-painter painter2  split-point
                                (make-vect 1.0 0.0)  (make-vect 0.5 1.0))))
        (lambda (frame)
          (paint-left frame)
          (paint-right frame)))))
```


语言设计和分层抽象

- 总结：语言里的 **painter** 和基本数据抽象都用过程表示。支持
  - 以统一方式处理各种本质上完全不同的基本画图功能
  - 组合方式有闭包性质，已有 **painter** 的组合仍然是 **painter**
  - 所有过程抽象手段都可以用于组合各种 **painter**
- 复杂系统应该通过分层设计完成
  - 描述这些层次需要一系列语言
  - 通过组合一层次的各种基本元素，得到更高层次的元素
  - 每层提供基本元素、组合手段和抽象手段，支持更高层构造
- 在图形语言实例中：
  - 基本语言提供基本图形功能，如为 **segmant->painter** 提供画线段功能，为 **rogers** 提供画图和着色功能
  - 基本 **painter** 提供基本图形，**beside** 和 **below** 等操作 **painter**
  - 还实现了图形操作的组合，以 **beside** 和 **below** 等为操作对象