[读书笔记]SICP：5[B]层次性数据和闭包性质

本章对应于书中的2.2。

- **表**是指那些有表尾结束标记的序对的链；**表结构**是指所有有序对构造起来的数据结构，不仅是表。
- 在使用`define`定义过程时，如果定义成如下形式

```
(define (<name> <v1> <v2> . <v3>) <body>)
```

则前面的各个形参将以前面的各个实际参数为值，而最后一个形参将以所有剩下的实际参数的表为值。更进一步，以下形式定义的过程可以接收任意数量的参数调用

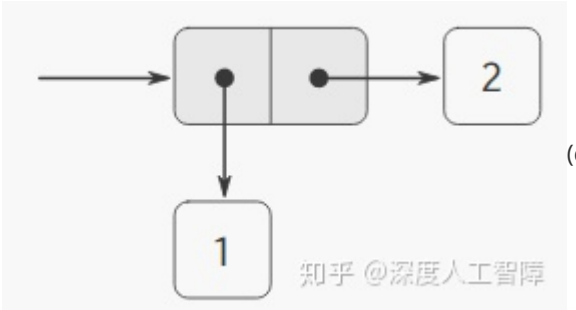
```
(define (<name> . <v>) <body>)
```

- 数据抽象使我们能设计出不会被数据表示的细节纠缠的程序，使程序能够保持很好的弹性
- 对不同数据结构的执行进行规范化，指定一种数据结构作为信号，通过对该数据结构的操作来使得信号在不同处理阶段中传递。从而得到能处理不同数据结构的程序。
- **精华：**选择一种合适的约定界面来连接不同的标准部件，使得我们能对这些标准部件进行模块化设计而无需考虑具体的数据结构（因为连接部件的界面是约定好的），由此来控制程序复杂度，使得我们能通过级联标准部件的方式来构建复杂的系统。
- **精华：**由于我们选择了序列作为约定界面，可以利用这种界面组合起各种处理模块。且序列作为统一的界面，使得程序对数据结构的依赖性局限到不多的几个序列操作上，通过修改这些操作，就能在序列的不同表示之间进行转换，而保持程序的整体设计不变。（就是用一个过程将不同的数据结构转化为统一的界面，而后不同的部件都是对该约定的界面进行操作，从而无需考虑具体的数据结构）

---

1 数据结构

从之前可以看出，序对为我们提供了一种用于构造复合数据的基本“粘合剂”，可以用**盒子和指针表示方法**进行表示。

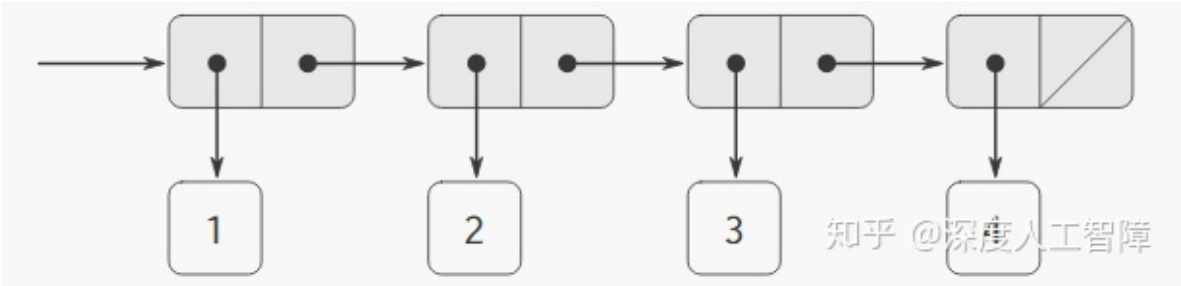


(cons 1 2)的盒子和指针表示

此外，我们可以建立元素本身也是序对的序对，这称为`cons`的**闭包性质**：某种组合数据对象的操作满足闭包性质，则它组合起数据对象得到的结果本身还可以通过同样的操作再进行组合。比如组合式的成员本身还可以是组合式。而`cons`的闭包性质使得我们能够通过不同嵌套`cons`的方法来建立起不同的**层次性**的结构。

1.1 序列的表示

我们可以通过序对构造如下的**序列**，其中每个序对的`car`指向序列中的一个元素，而`cdr`指向下一个序对，而最后一个序对的`cdr`指向一个特殊的表示结尾的元素`nil`，而空表可以直接表示为`nil`。这种带有表尾结束标记的序列也称为**表**。



上面的这种结构我们可以通过以下方式构建

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

而Scheme为了简便，提供了一种等价的基本操作

```
(define l (list 1 2 3 4))
```

注意：当考虑`list`构建起来的序列如何操作时，将其转为等价的`cons`表示就理解了。

当我们运行`(car l)`时，表示取第一个元素`1`，当我们运行`(cdr l)`时，表示取序对的下一部分，也就是`(1 2 3)`。我们也可以通过`(cons 0 l)`在开头插入一个`0`，从而得到`(0 1 2 3 4)`。

我们也可以对序列执行各种操作：

- 返回表中的第`n`个元素

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

- 返回序列的长度，Scheme提供`null?`来判断序列是否为空表

```
; 递归计算过程
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))
; 迭代计算过程
(define (length items)
  (define (iter a n)
    (if (null? a)
        n
        (iter (cdr a) (+ n 1))))
  (iter items 0))
```

- 合并两个序列

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

- 返回表中的最后一个元素

```
(define (last-pair l)
  (let ((next (cdr l)))
    (if (null? next)
        l
        (last-pair next))))
```

- 逆序

```
;; 迭代计算过程
(define (reverse1 l)
  (define (it-rev lat ans)
    (if (null? lat)
        ans
        (it-rev (cdr lat) (cons (car lat) ans))))
  (it-rev l nil))

;; 递归计算过程
(define (reverse2 lat)
  (if (null? lat)
      nil
      (append (reverse2 (cdr lat)) (list (car lat)))))
```

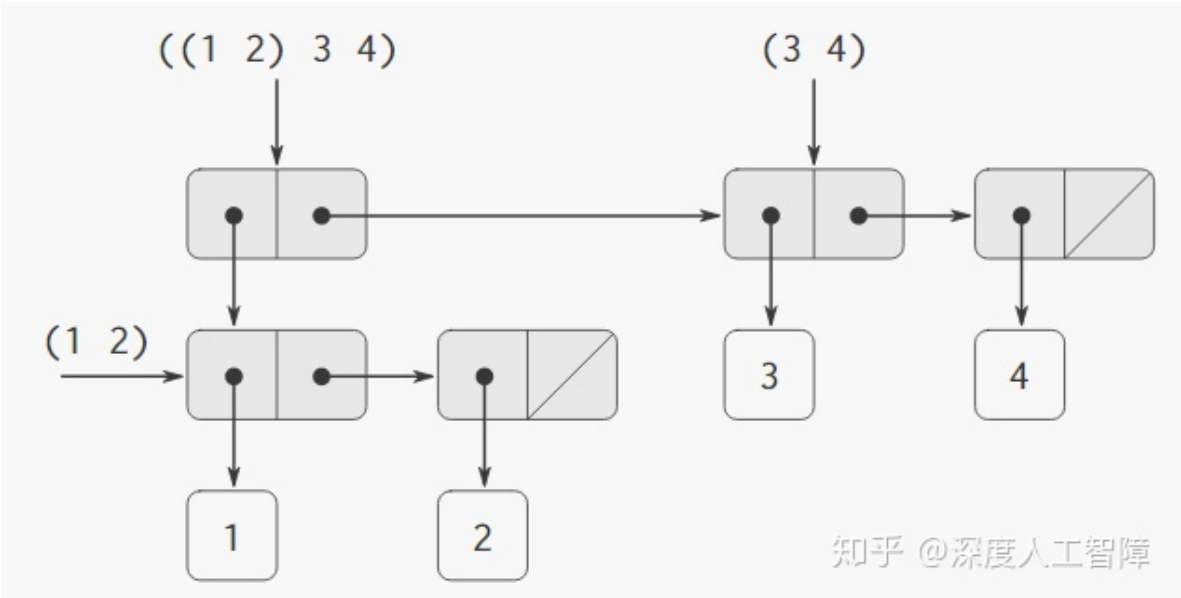
此外，我们也可以定义一个`map`过程，将某个过程应用于表中的所有元素，并返回结果形成的表

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

这也提供了一层抽象屏障，将实现表变换的过程的实现与如何提取表中元素以及组合结果的细节隔离开来。

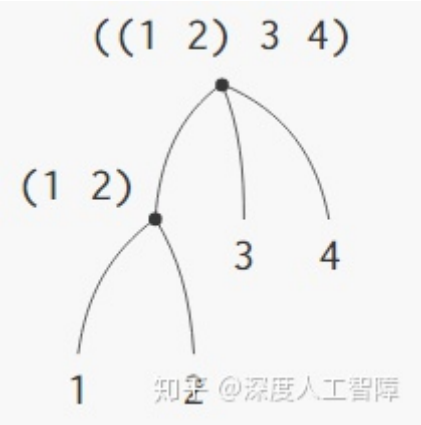
1.2 层次性结构

我们可以通过序对构造如下的**树**，序列的元素本身也是序列。



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

它对应的树结构为如下形式，可以发现，`car`为`list`的构成一个子树，而`cdr`为`list`的则构成若干个分支。



我们也可以对数执行各种操作：

- 统计一棵树中树叶的数目，使用Scheme提供的基本过程`pair?`来检查参数是否为序对。它分别递归树的两个分支`car`和`cdr`，通过`pair?`判断如果其中一个分支不为序对，则说明是元素，则直接返回1，否则继续递归。

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

- 树翻转

```
(define (deep-reverse tree)
  (define (iter t ans)
    (cond ((null? t) ans)
          ((not (pair? t)) t)
          (else (iter (cdr t)
                      (cons (iter (car t) nil)
                            ans)))))
  (iter tree nil))
```

- 统计叶子数目

```
(define (fringe tree)
  (define (iter t ans)
    (cond ((null? t) ans)
          ((not (pair? t)) (cons t ans))
          (else (iter (car t)
                      (iter (cdr t) ans)))))
  (iter tree nil))
```

此外，我们也可以定义一个`map`过程，将某个过程应用于树中的所有叶子元素，并返回结果形成的树

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

我们同样可以将树看成是子树的序列，然后使用序列的`map`过程来完成树的`map`过程

```
(define (tree-map proc tree)
  (list-map (lambda (sub-tree)
              (if (pair? sub-tree)
                  (tree-map proc sub-tree)
                  (proc sub-tree)))
            tree))
```

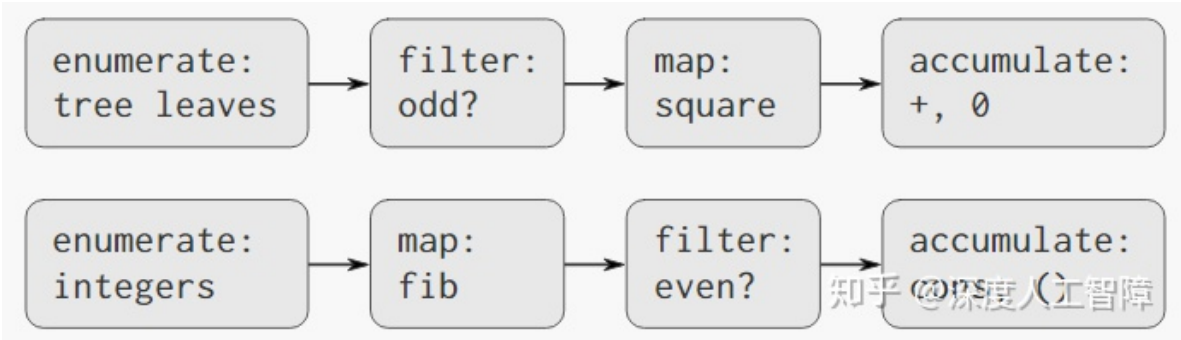
2 序列作为一种约定的界面

第一章中我们通过高阶过程来表示“概念”，使得我们能通过传递不同的过程来实现相同概念的计算。而想要在抽象数据中也到达这种抽象程度，就需要对我们操控数据结构的方式进行调整。比如我们以以下两个程序为例

```
; 计算树中叶子为奇数的平方和
(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))))))

; 计算获得偶数菲波那切数的表
(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))
```

可以发现这两个程序在处理不同数据结构上的风格完全不相同，我们通过信息流图来查看是否存在共性



可以发现，这两个程序的步骤可以归纳为以下几步：

- **枚举**：选出用于计算的元素。程序一中选出所有的树叶，程序二中选出所有的数字。
- **过滤或映射**：对选出的元素进行过滤，或对选出的元素映射到对应的值。程序一中只保留奇数的树叶，程序二中将数字映射为对应的斐波那契额数。
- **过滤或映射**：程序一中对保留的树叶求平方，程序二中保留偶数的斐波那契额数
- **累积**：对之前计算出的元素进行累积。程序一将其加和，程序二将其组合成表。

可以发现这两个程序具有相似的信号处理步骤，但是他们采用了不同的方式分解了这些计算，使得各个步骤都混杂在一起，我们可以尝试对其按照信号流结构进行规范化，来得到能处理不同数据结构的统一过程。这里我们需要关注从一个阶段到下一个阶段的信号，我们这里尝试用表来表示这些信号，则每个步骤其实就是对表的操作。

- **枚举**：枚举中这两个程序是尝试枚举不同的数据，我们这里用两个不同的枚举过程来进行，但是**注意**要产生表来作为信号。

```
; 程序一：枚举出所有叶子
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))
; 程序二：枚举出区间内的所有数
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low
            (enumerate-interval (+ low 1) high))))
```

- **过滤**：当我们以表作为信号时，传入过滤过程的数据就是表，则过滤过程其实就是根据某种给定谓词来对表中元素进行挑选，保留下的元素还是组合成表的形式。

```
; 通过传入谓词predicate来对表sequence进行挑选
;; 程序一中传入的谓词为odd?
;; 程序二中传入的谓词为even?
(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else  (filter predicate (cdr sequence)))))
```

- **映射**：当我们用表来作为信号时，映射其实也就是`map`过程了。
- **累积**：当我们用表来作为信号时，累积过程就是按照给定的累积操作对表中的元素进行累积

```
; 通过传入累积操作op和初始值initial来对表sequence进行累积
;; 程序一中传入的op为cons，initial为nil
;; 程序二中传入的op为+，initial为0
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op  initial (cdr sequence)))))
```

由此我们以序列作为一种在信号处理不同阶段的约定界面，来得到统一的不同阶段的过程，使我们能得到模块化的程序设计。根据以上不同的步骤的过程，我们可以对这两个程序进行重构

```
; 程序一
(define (sum-odd-squares tree)
  (accumulate
   +
   0
   (map square
        (filter odd?
                (enumerate-tree tree)))))
; 程序二
(define (even-fibs n)
  (accumulate
   cons
   nil
   (filter even?
```

```
            (map fib
                 (enumerate-interval 0 n)))))
```

不仅如此，由于我们选择了序列作为约定界面，可以利用这种界面组合起各种处理模块。且序列作为统一的界面，使得程序对数据结构的依赖性局限到不多的几个序列操作上，通过修改这些操作，就能在序列的不同表示之间进行转换，而保持程序的整体设计不变。

此外，我们还能封装以下过程

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

表示通过`proc`对`seq`中的所有元素进行扩展后，通过`append`拼接到一起。相当于是对序列进行扩展。

3 强健设计的语言层次

强调使用**分层设计**来对程序进行设计，也就是说一个复杂的系统应该通过一系列的层次构造出来。我们需要一系列的语言来描述这些层次，设法组合其作为这一层次中部件的各种基本元素，而这些构造出的部件又可以作为另一个层次中的基本元素。每个层次上所用的语言都提供了一些基本元素、组合手段，还对该层次中的适当细节做抽象的手段。并且分层设计使得程序更加强健，使我们可以在给定规范发生一些小改变时，只需要对程序做少量的修改。分层结构中的每个层次都为表述系统的特征提供了一套独特词汇，以及一套修改这一系统的方式。

比如书中的图形语言

- 在最低层次中，基于描述点和直线的语言建立了一个基本元素painter，可以创建出一个画折线图的painter。我们通过将painter设计成过程的形式，使其能满足闭包性质，从而嵌套构建出更加复杂的painter。并抽象出一个高阶过程`transform-painter`来对painter进行变换。
- 再上一层次，基于基本元素painter构建出一些组合方式，比如`beside`和`below`，使其能在不了解painter细节的基础上来构建组合方式（因为painter的细节在上一层次进行封装了）。并且也能在`beside`和`below`的基础上构建其他的组合方式。
- 在最后一层中，可以基于上一层中设计的各种组合方式抽象出一写描述组合模式的高阶过程，比如`square-of-four`，通过将上一层的组合方式作为基本元素传入高阶过程，来创建不同的组合方式。在这一层中无需了解各种组合方式是如何实现的，因为已在上一层中进行封装了。

从中我们可以看出分层设计的形式：在最底层设计出基本元素以及对基本元素进行变换的高阶过程，在第二层中基于基本元素设计出不同的组合方式，在最后一层中抽象出描述组合模式的高阶过程。