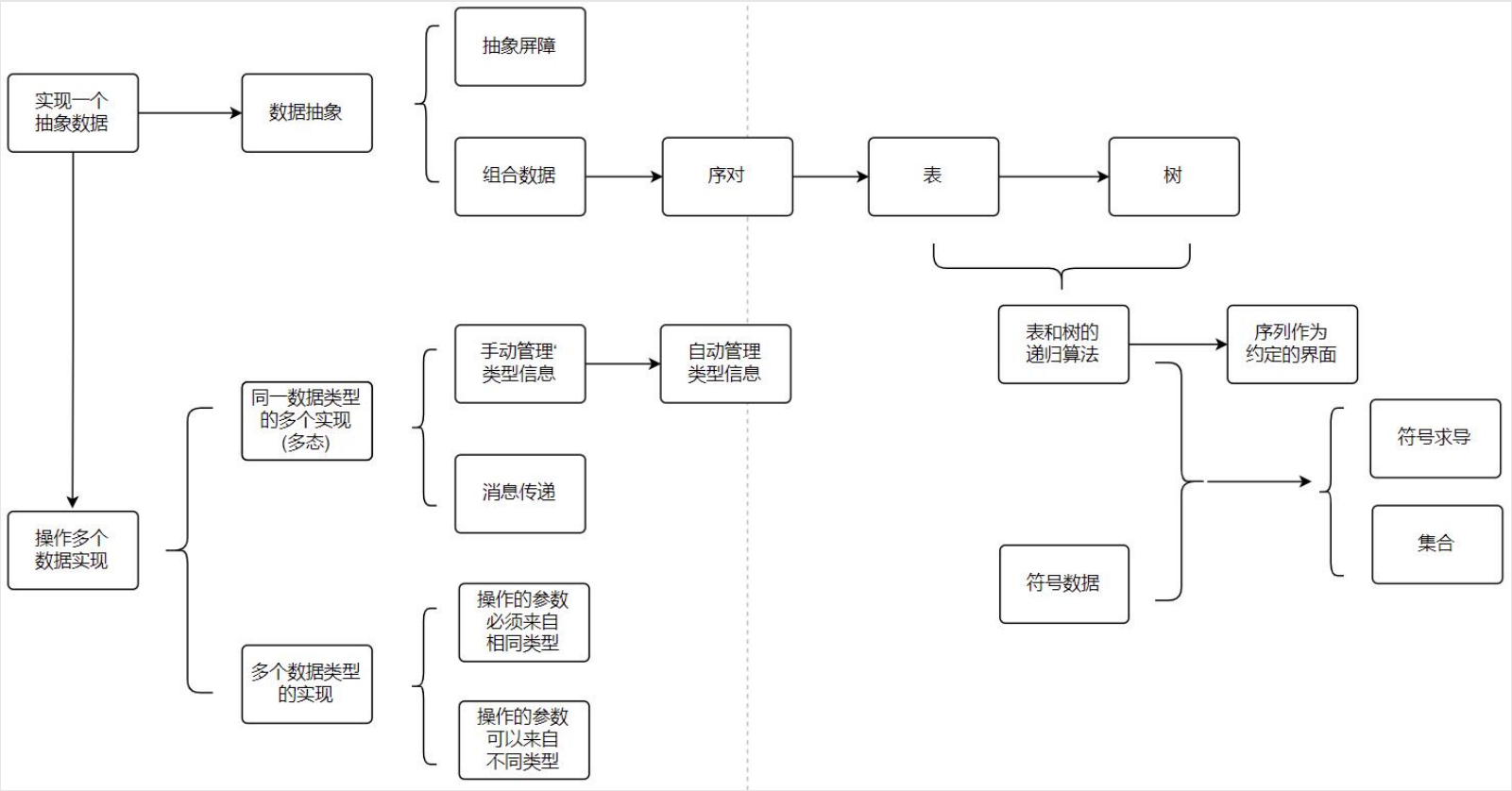
计算机科学导论-SICP
第二章 构造过程抽象
2.2小节 层次性数据和闭包性质

王超

Center for Research and Innovation in Software Engineering (RISE), Southwest University

2022 年 11 月 3 日

第二章的学习思路


实现一个
抽象数据
数据抽象
抽象屏障
组合数据
序对
表
树
表和树的
递归算法
序列作为
约定的界面
符号求导
集合
符号数据
操作多个
数据实现
同一数据类型
的多个实现
(多态)
手动管理
类型信息
自动管理
类型信息
消息传递
多个数据类型
的实现
操作的参数
必须来自
相同类型
操作的参数
可以来自
不同类型

2.2节讲了什么故事

我们开始接触包含多于两个数字的数据

- 表：我们终于有了构造包含多个元素的数据单元的方法
- 树：以嵌套的表的形式实现
- 使用递归算法处理表和树
- 序列作为约定的界面：以表处理为核心的编程
- 多个例子

类似1.2小节在第一章中的作用和篇幅。本小节是本章的重要内容，篇幅较多

2.2节的PPT涉及哪些内容

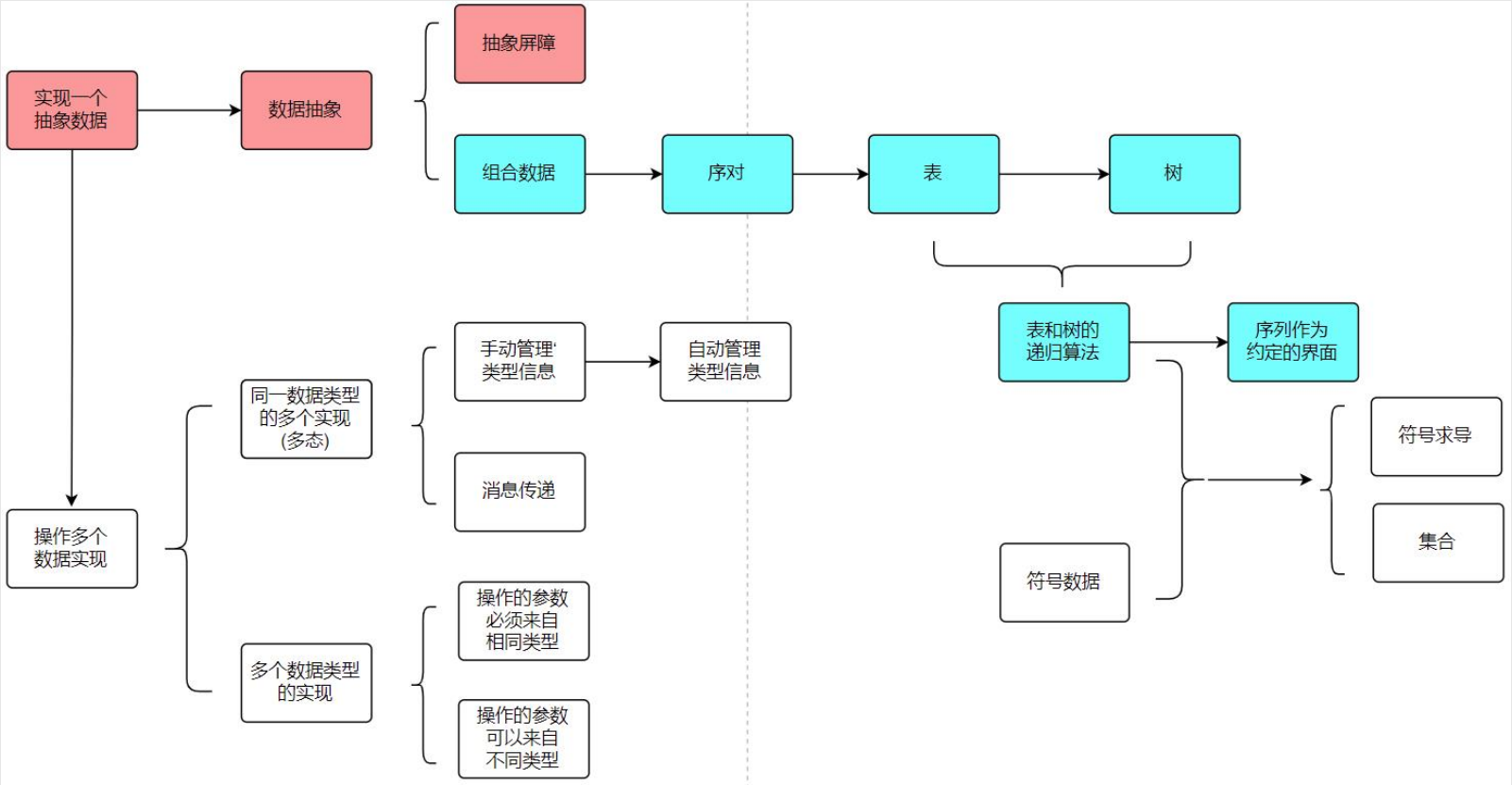
- 表的定义和操作
- 树的定义和操作
- 序列作为约定的界面
- 嵌套映射

内容调整

- 2.2.4节不讲
- 嵌套映射不会在考试中以写代码的形式考察

2.2节的内容

红色为已讲解内容，蓝色为本次要讲解的内容



大纲

闭包性质

- 序对的一个元素可以是一个数字，一个函数，甚至是另一个序对
- 由于序对的元素可以是序对，序对因此可以用来保存多于两个元素
- 例子：(cons (cons 1 2) (cons 3 4))

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
- 这种性质被称作闭包性质
- 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

- 满足闭包性质的组合子具有很强的能力
- 这样的组合子可以构造出层次性的结构
- 例如本节后面讲到的，使用序对构造表和树

很多语言的操作子没有闭包性质

- C语言中的数组没有闭包性质，数组的元素不能是数组
- C语言中的结构的成员可以是结构，但必须是固定种类的结构

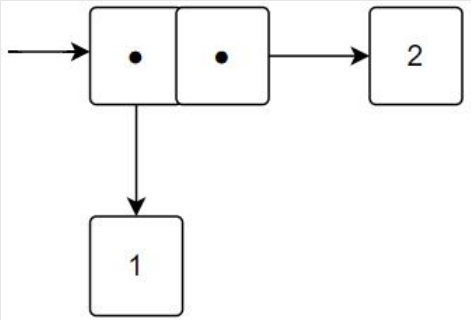
盒子和指针表示

- 嵌套序对可能很复杂，难以理清元素之间的关系
- 图像表示往往更加直观

为了更清晰的展示（嵌套）序对的内容，引入序对的盒子和指针表示

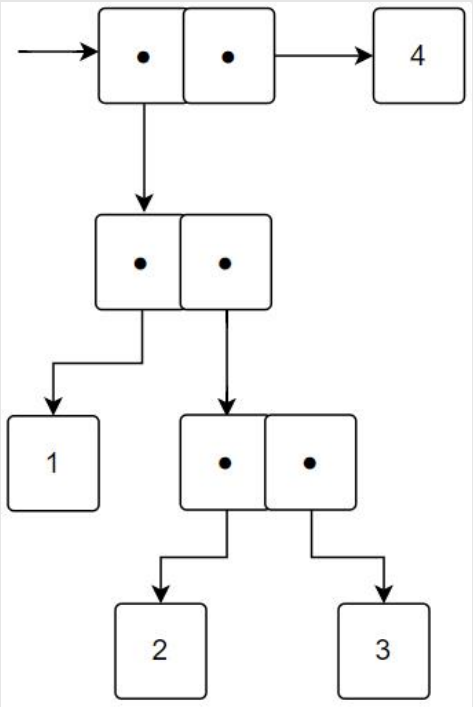
- 一个序对被表示为一对紧邻的方块，每个方块内有一箭头（指针）
- 顺着第一个方块内的箭头可以找到序对首元素
- 顺着第二个方块内的箭头可以找到序对第二个元素
- 当序对元素是一个数字时，我们绘制包含数字的单方块。否则，我们继续画方块对

- 以(cons 1 2)为例



- 注意，这里的指针（pointer）指的是绘图上的箭头，而不是C语言中的指针
- 箭头水平指向，斜线指向或垂直指向没有分别

- 使用这个方法绘制复杂的序对
- 以(cons (cons 1 (cons 2 3)) 4) 为例

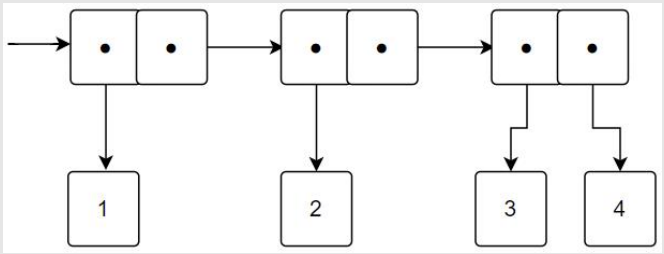


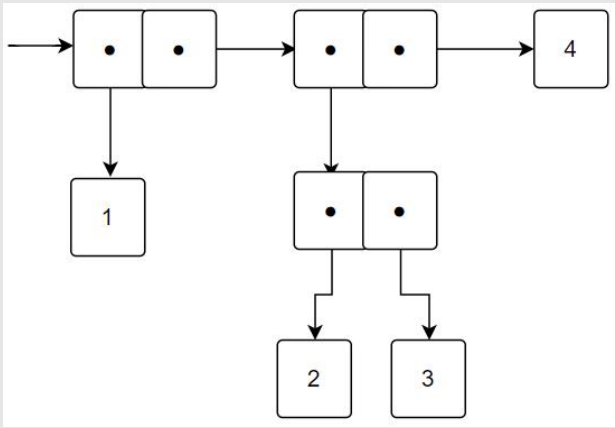
多样的嵌套可能

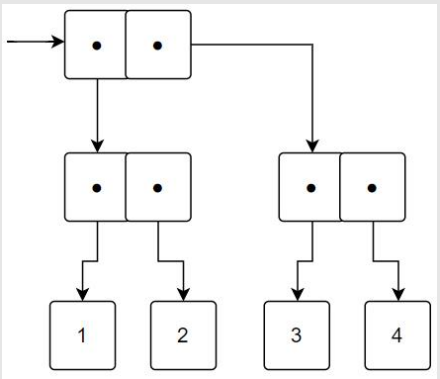
编码数据时有着多种编码方案

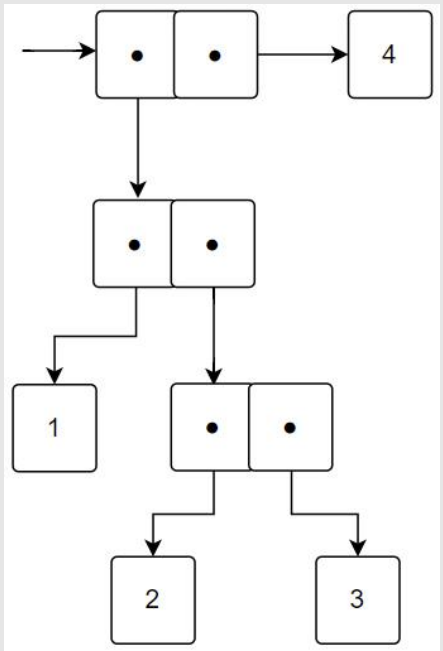
- 允许嵌套，序对中存储四个数字1、2、3和4，且代码中四个数字总按照从小到大顺序出现
- 一共有五种方案
- 序对首元素为1：(cons 1 (cons 2 (cons 3 4)))，(cons 1 (cons (cons 2 3) 4))
- 序对首元素为(cons 1 2)：(cons (cons 1 2) (cons 3 4))
- 序对首元素包含1、2和3：(cons (cons 1 (cons 2 3)) 4)，(cons (cons (cons 1 2) 3) 4)

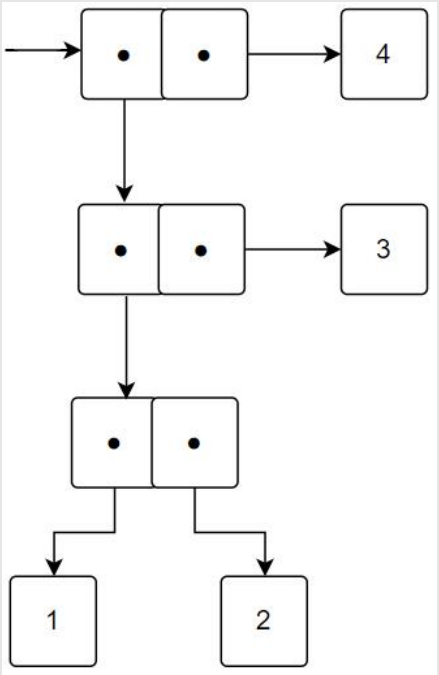
- 盒子和指针表示可以更好的区分这五种情况











如何存储一串数据

- 不采用上一页的五种方案
- 因为有时需要通过car读到数据，有时需要通过cdr读到数据，方式不统一

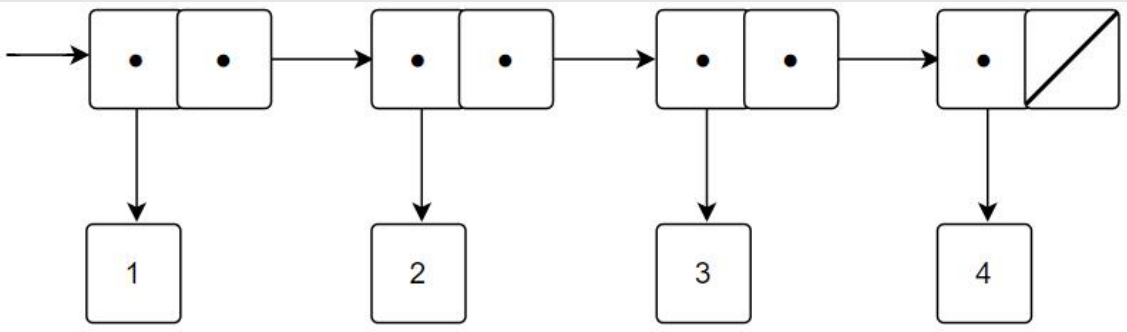
解决方案

- 类似方案1，但要求所有元素都以相同风格保存，为此需明确标识串的结束
- 串中的数据只能通过car读取
- 为了保证构成序对，引入一个特殊的值nil，表示串的结束
- nil只出现在最内层序对的次元素位置

表

- 以特定的格式，用嵌套的序对存储一个串
- 最外层序对（假定名为items1）的首元素存储串的首元素
- 最外层序对的第二元素是另一个序对（假定名为items2），即items2=(cdr items1)，items2存储串除了首元素以外的部分
- items2的情况类似。它的首元素是原串的第二个元素
- (cdr items2)=(cdr (cdr items1))是另一个序对，存储原串除了前两个元素以外的部分
- ...
- 串的最后一个元素a存储在序对(a nil)中
- 这种结构称为表
- 串的每个元素，都可以通过对“存储以这个元素开始的串的子部分”执行car过程得到

- 例子：内容为1、2、3、4的表，表示为(cons 1 (cons 2 (cons 3 (cons 4 nil))))



注意图中序对之间的包含关系

- 令此表名为items
- (car items)=1，(cdr items)=(2 3 4)
- (car (cdr items))=2，(cdr (cdr items))=(3 4)
- (car (cdr (cdr items)))=3，(cdr (cdr (cdr items)))=(4)
- (car (cdr (cdr (cdr items))))=4，(cdr (cdr (cdr (cdr items))))=()

缩写

- 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))))

list语句

- 用来简化表的构造
- 格式：(list a_1 ... a_k)
- 构造元素为a_1 ... a_k的表
- 相当于(cons a_1 (cons a_2 (cons ... (cons a_k nil) ...)

上一页的表可以由语句(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（在1的位置），参数为2、3、4，进而报错

对数时间的斐波那契算法

- 在1.2节我们讲过，斐波那契数列也可以表示为幂函数形式

$$\begin{bmatrix} fib(n+1) & fib(n) \\ fib(n) & fib(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

- 现在我们可以实现这个算法
- 使用表实现2*2的矩阵的乘法
- 一个2*2矩阵表示为列表(a_{11} a_{12} a_{21} a_{22})

- 令fibBase为表(1 1 1 0)对应的2*2矩阵
- 过程(mul22Matrix a b)计算两个2*2矩阵的乘积
- 需习惯let语法

```
;2*2矩阵1 1 1 0
(define fibBase (list 1 1 1 0))

;计算两个2*2矩阵的乘法
(define (mul22Matrix a b)
  (let ( (a11 (car a))
         (a12 (cadr a))
         (a21 (caddr a))
         (a22 (cadddr a))
         (b11 (car b))
         (b12 (cadr b))
         (b21 (caddr b))
         (b22 (cadddr b)))
    (list (+ (* a11 b11) (* a12 b21))
          (+ (* a11 b12) (* a12 b22))
          (+ (* a21 b11) (* a22 b21))
          (+ (* a21 b12) (* a22 b22)))))
```

- (fibMatrixExp n)计算fibBase矩阵的n次方，使用二分法计算
- (fibMatrixExp n)计算出的2*2矩阵正是上上张PPT中的矩阵，(fib n)正是其右上角元素，即(cadr (fibMatrixExp n))
- 为了不处理计算计算矩阵的0次方的情况，(fib 0)直接返回0
- 由于采用了二分法的思想，算法的时间增长阶为对数

```
;计算fibBase的n次方
(define (fibMatrixExp n)
  (cond ( (= n 1) fibBase)
        ( (even? n)
          (let ( (a (fibMatrixExp (/ n 2))) )
            (mul22Matrix a a)))
        (else (mul22Matrix fibBase (fibMatrixExp (- n 1))))))


(define (fib n)
  (if (= n 0)
      0
      (cadr (fibMatrixExp n))))
```

表的操作

- 表的格式形如(表头 余下部分)，天然适合递归求解

使用递归访问表items的各个元素

- =通过(car items)访问表的首元素+访问表(cdr items)的各个元素
- “拆开”一个表

使用递归构造内容为$a_1, \ldots, a_k$的表

- =a_1 + 递归构造内容为$a_2, \ldots, a_k$的表

1. 递归地拆开一个表

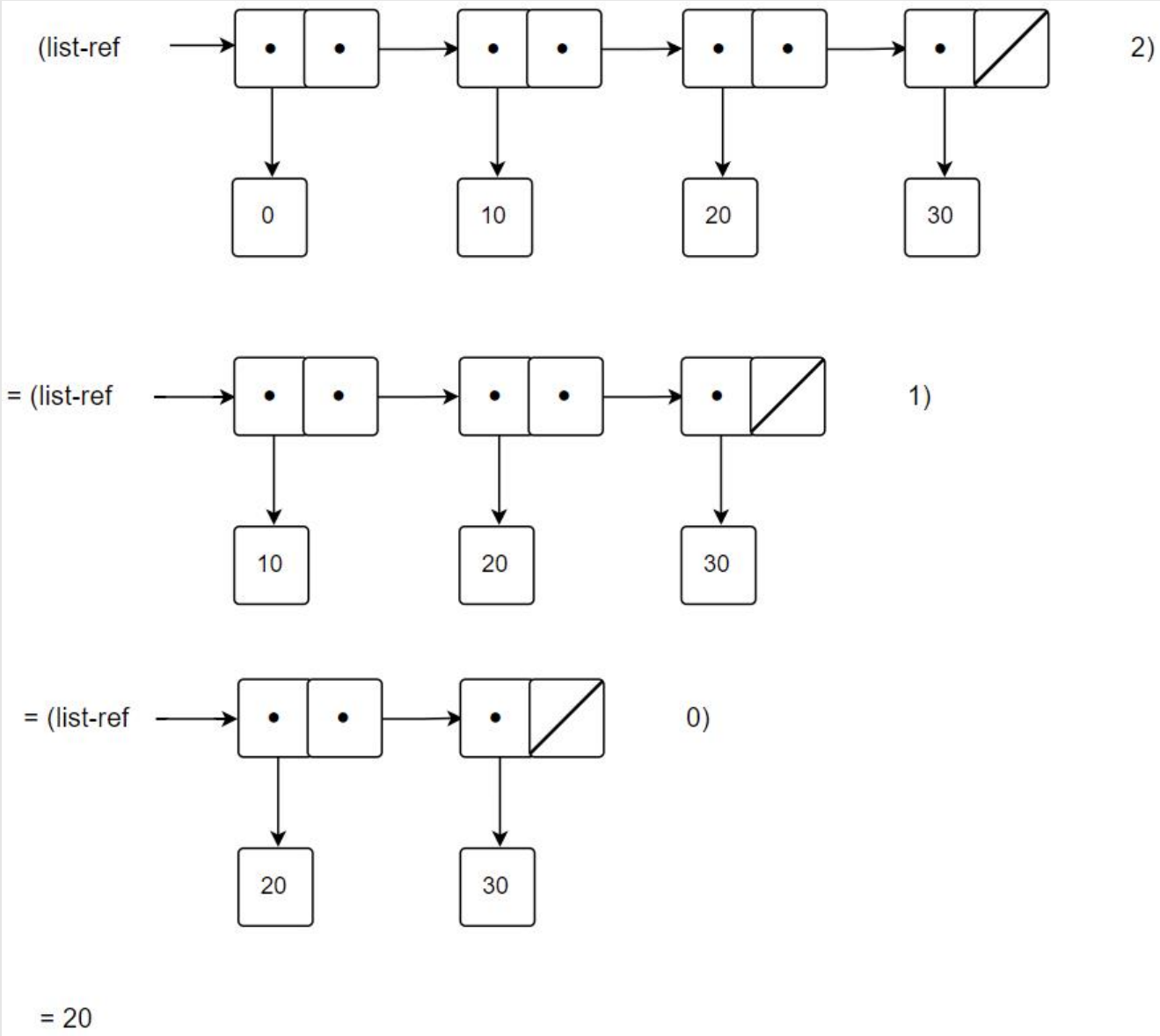
- 需要从表的头元素开始依次读表元素，直到找到特定元素并返回
- 使用递归依此访问表的各个元素

list-ref

- (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
- 注意，需用户保证n在合法的范围内，否则执行会出错

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

以(list-ref (list 0 10 20 30) 2)为例



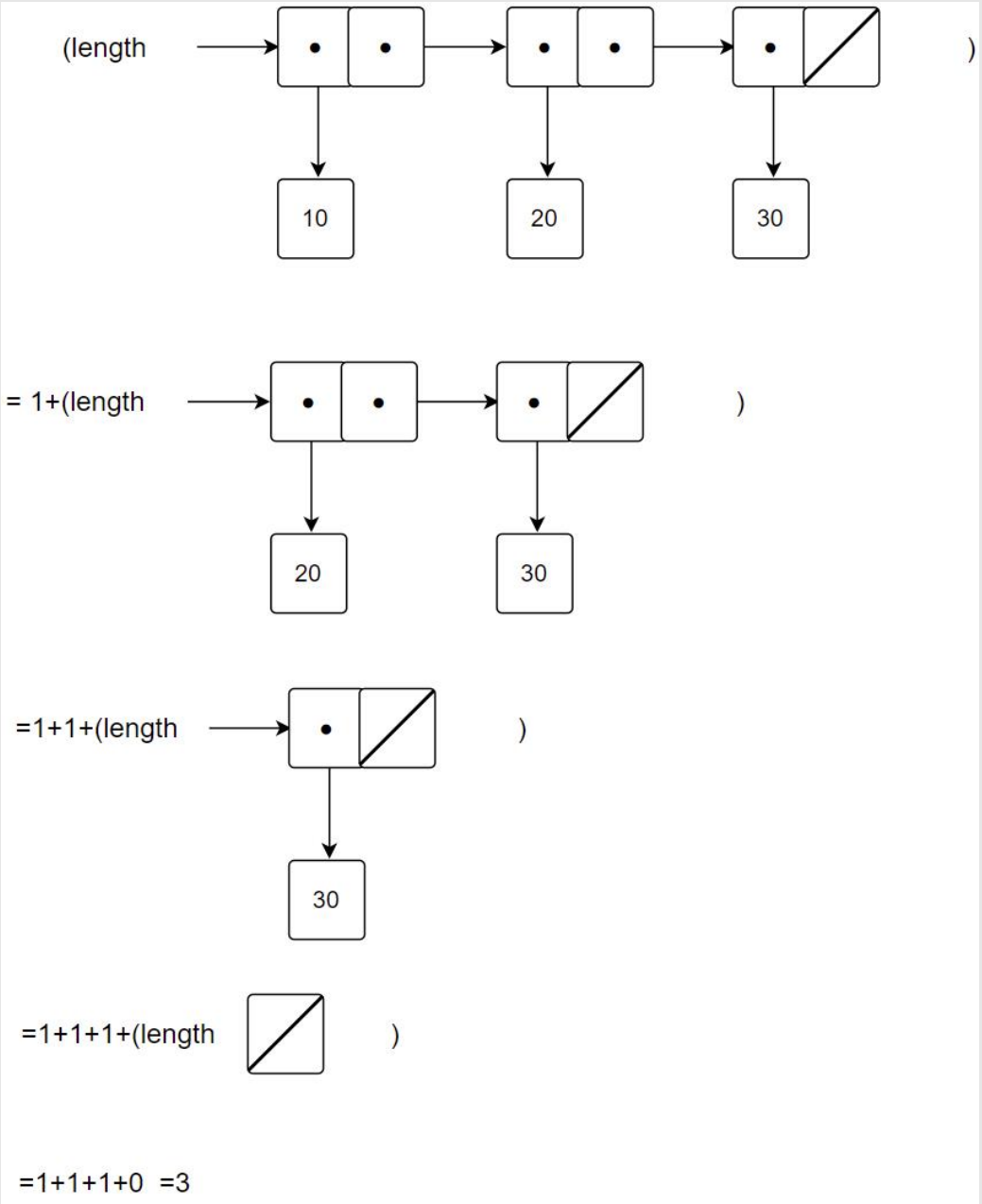
2. 判断到达表尾

- 仍然是递归地拆开表。不同于使用car访问元素，这次需要判断是否到达了表尾
- Scheme提供了一个过程null?
- (null? items)用来判断items是否等于nil，即是否是表的“最右边”

例子：length

- (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，(length items) = 1+(length (cdr items))

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


(length
)
10
20
30
= 1+(length
)
20
30
=1+1+(length
)
30
=1+1+1+(length
)
=1+1+1+0 =3

3. 递归地构造表

- 之前关注“自上而下地拆开表”。而现在关注如何构造一张新表
- 从旧表构造新表，新表元素来自于原表的元素
- 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
- 原表拆解完成时，这些“递归的中间信息”会自动完成新表的构造工作

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

- 例如，(duplicateElement (list 1 2 3))=(1 1 2 2 3 3)
- 注意，构造出的结果要符合表的格式

duplicateElement

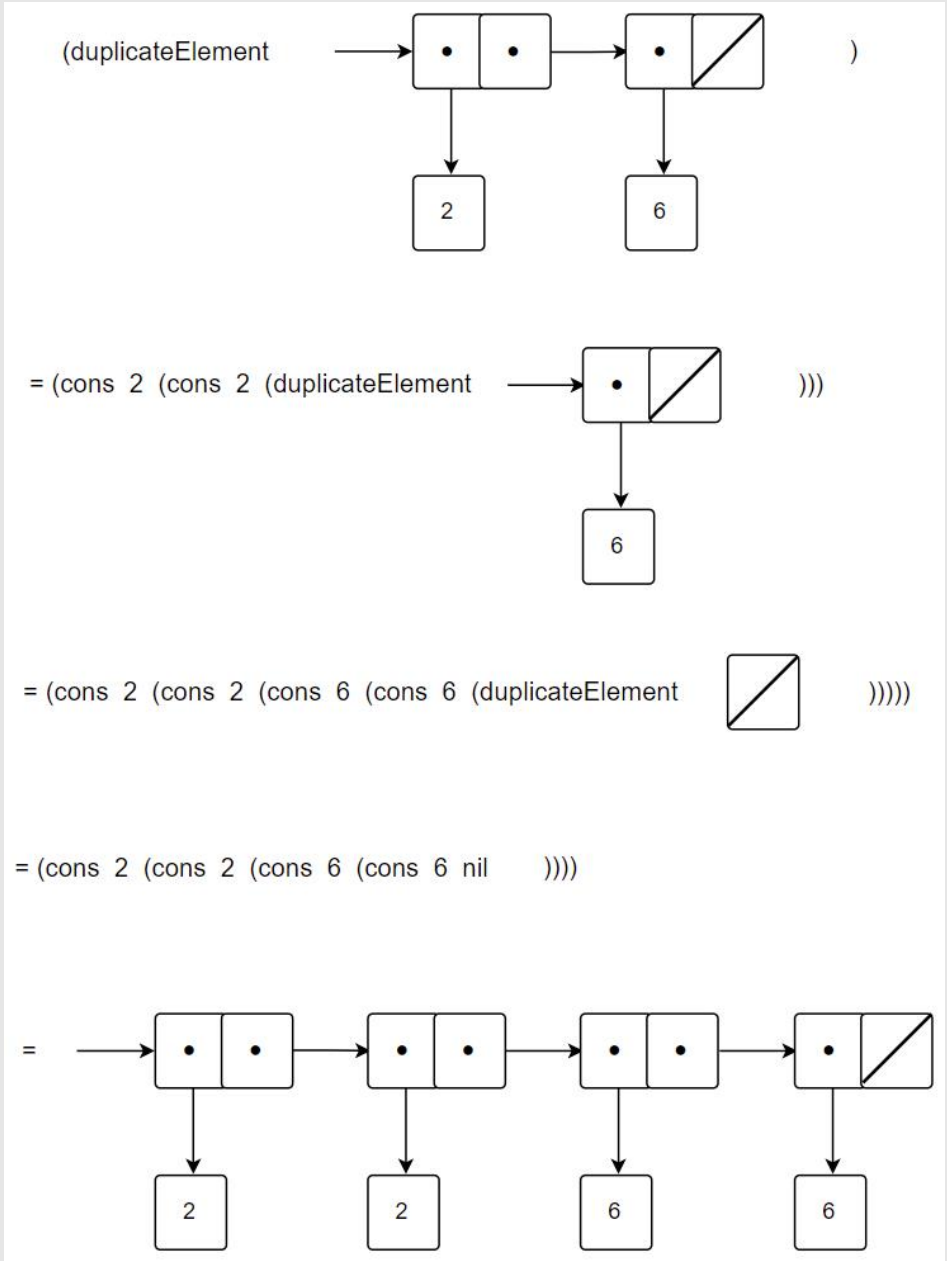
- 递归出口：(duplicateElement nil)=nil
- 递归等式：(duplicateElement (cons a s))=(cons a (cons a (duplicateElement s)))
- 递归地拆开原表头的元素a，并把两个a添加到新构造的表头
- 拆开原表到达表尾的时候，新构造的表生成其结尾
- 根据递归时记忆的中间信息，把这些元素拼接为完整的新表
- 拆开表时从左到右拆，拼接表时从右到左拼接

- 递归地拆开表+通过递归调用时生成的中间信息拼接表

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

本次PPT中另一个递归地构造表的典型例子是后面的(enumerate-interval low high)

- 构造表(low . . . high)
- 不是从旧表构造新表，而是从数字构造表


(duplicateElement
)
2
6
= (cons 2 (cons 2 (duplicateElement
)))
6
= (cons 2 (cons 2 (cons 6 (cons 6 (duplicateElement
)))))
= (cons 2 (cons 2 (cons 6 (cons 6 nil
))))
=
2
2
6
6

4. 递归地黏贴两个表

(append items1 items2)

- 构造一个新表，其内容先是items1的内容，然后是items2的内容
- 例如，(append (list 1 2) (list 3 4))=(1 2 3 4)

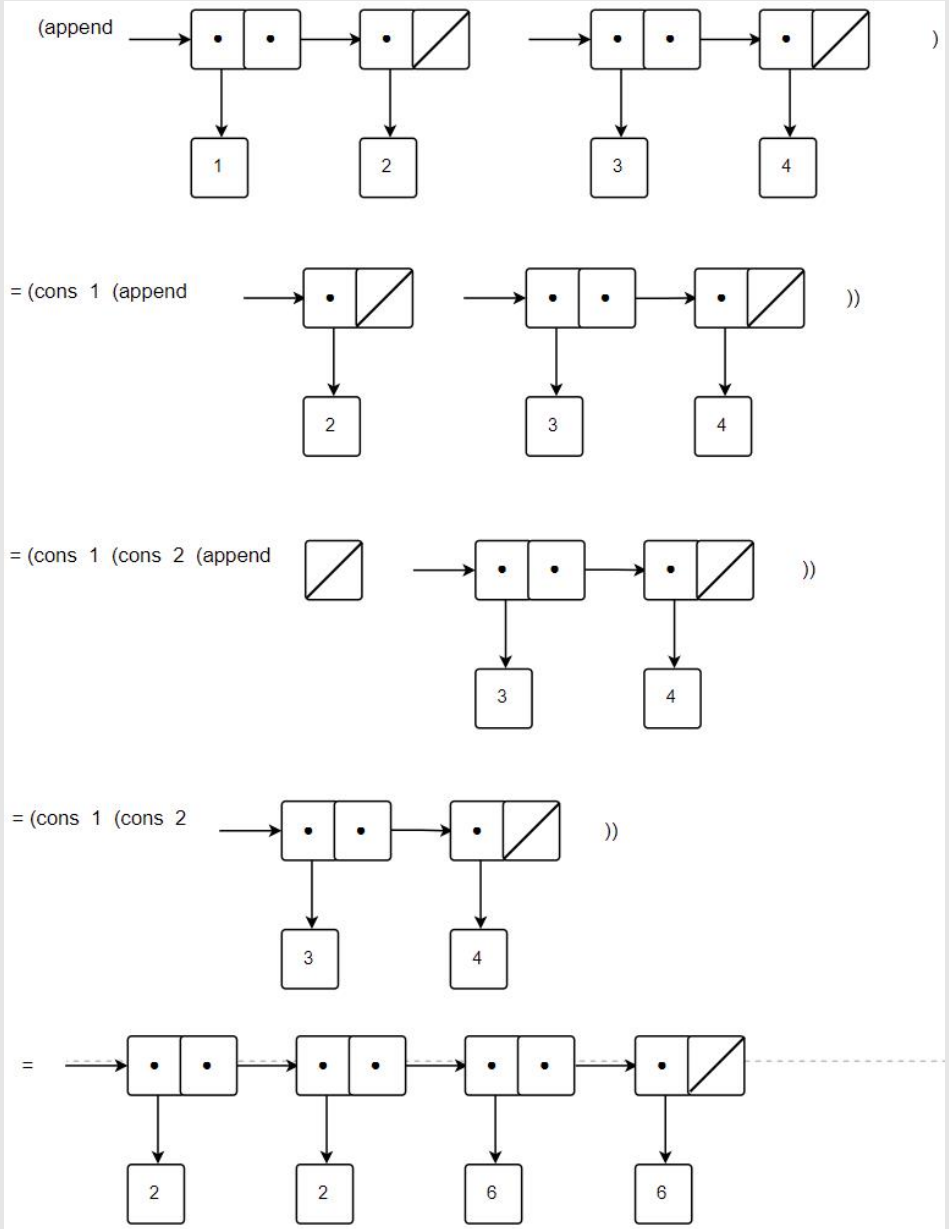
如何构造：不好的方案

- items1=nil时：(append items1 items2)=items2
- 否则：计算items1的末尾元素a（不是nil），令items1'为items1去掉末尾元素a得到的表，令items2'=(cons a items2)，然后递归执行(append items1' items2')
- 即(append items1 items2)=(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
- 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

如何构造：好的方案

- items1=nil时：(append items1 items2)=items2
- 否则：(append items1 items2) = (cons (car items1) (append (cdr items1) items2))
- 使用递归（递归地拆开表items1，直到items1为nil）来正向读items1的每个元素，并预留表的构造操作（cons）
- 在把items1拆完之后，再执行递归调用时保存的中间信息（cons），将items1的元素从右到左依次粘贴在items2的开头
- 在这个方法中，将items1中元素粘贴到items2的过程，是由递归调用时保存的中间信息完成的

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


(append
)
1
2
3
4
= (cons 1 (append
))
2
3
4
= (cons 1 (cons 2 (append
))
3
4
= (cons 1 (cons 2
))
3
4
=
2
2
6
6

5. 迭代地构造表

教材练习2.18：构造过程reverse

- (reverse items)把表items取反
- 例子：(reverse (list 2 6 3))=(3 6 2)
- 要保证构造出来的数据有着表的格式

reverse使用迭代更容易构造

- 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
- 读items中下一个元素，成为新表的倒数第二个元素，...
- 使用迭代完成这个过程

```
(define (reverse items)
    (cond ((null? items) items)
          ((null? (cdr items)) items)
          (else (reverse-iter (cdr items) (cons (car items) nil)))))
;args: a是items中剩下未处理的表，b是items中已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))))
```

以(reverse (list 2 6 3))为例

- 调用(reverse-iter (cons 6 (cons 3 nil)) (cons 2 nil))

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

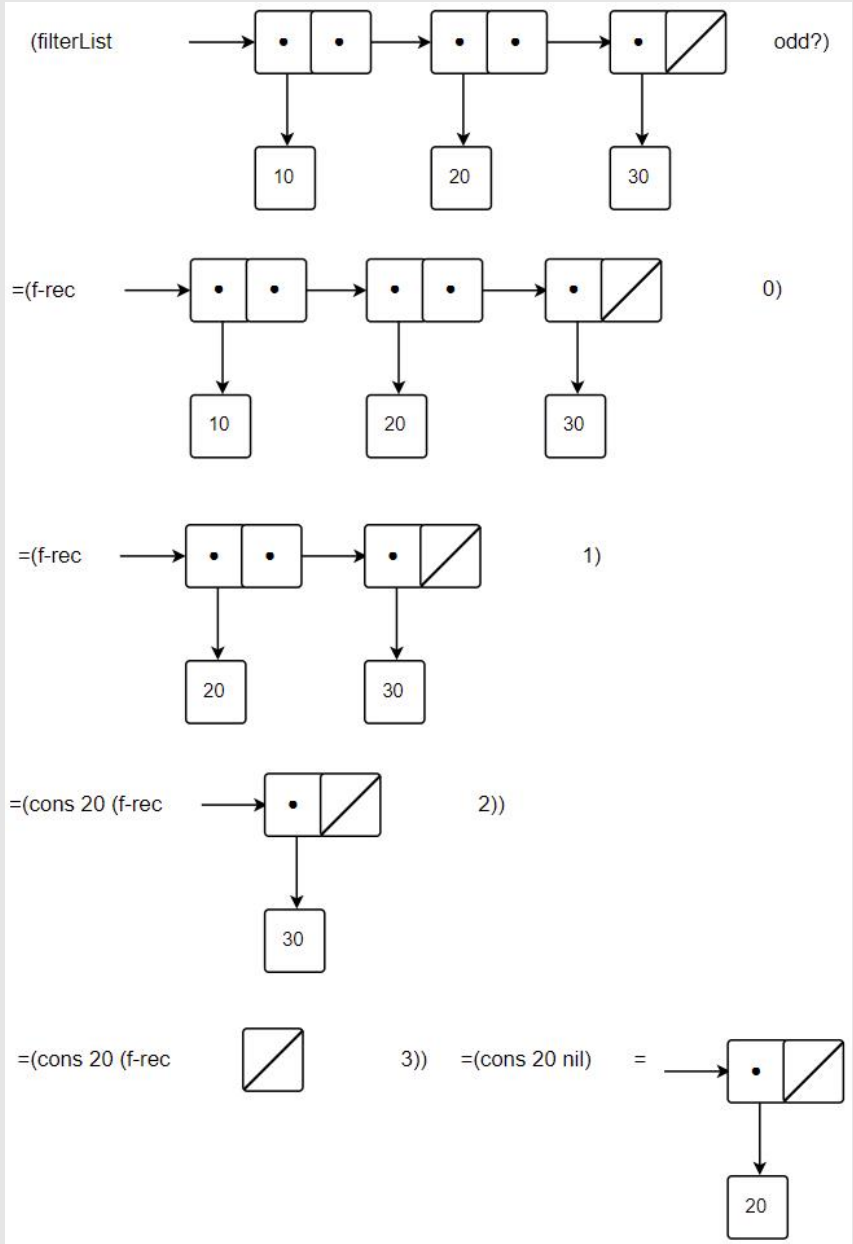
- 一些操作需要依据下标来处理表中元素
- 使用递归遍历表时，可以额外记录当前表的首元素在原表中的下标

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

- 参数pred是一个过程，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
- 注意，判断依据是下标，而不是表的元素的值

- 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
- f-rec递归地拆开表，根据当前元素在items中的下标是否满足pred，选择是否把这个元素放入新构造的表

```
(define (filterList items pred?)
  ;args: 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))
```


(filterList
odd?)
10
20
30
=(f-rec
0)
10
20
30
=(f-rec
1)
20
30
=(cons 20 (f-rec
2))
30
=(cons 20 (f-rec
3))
=(cons 20 nil)
=
20

课堂思考题

- 实现过程f，反向输出由表中下标为3的倍数的元素组成的表
- 例子：(f (list 1 3 5 7 9 11))的结果为(7 1)
- 提示：使用已实现的过程
- 教材练习2.19

对表的映射

表的操作的一种共同模式

- 以同一种方式改变表的每个元素，除了nil

这种操作并不罕见

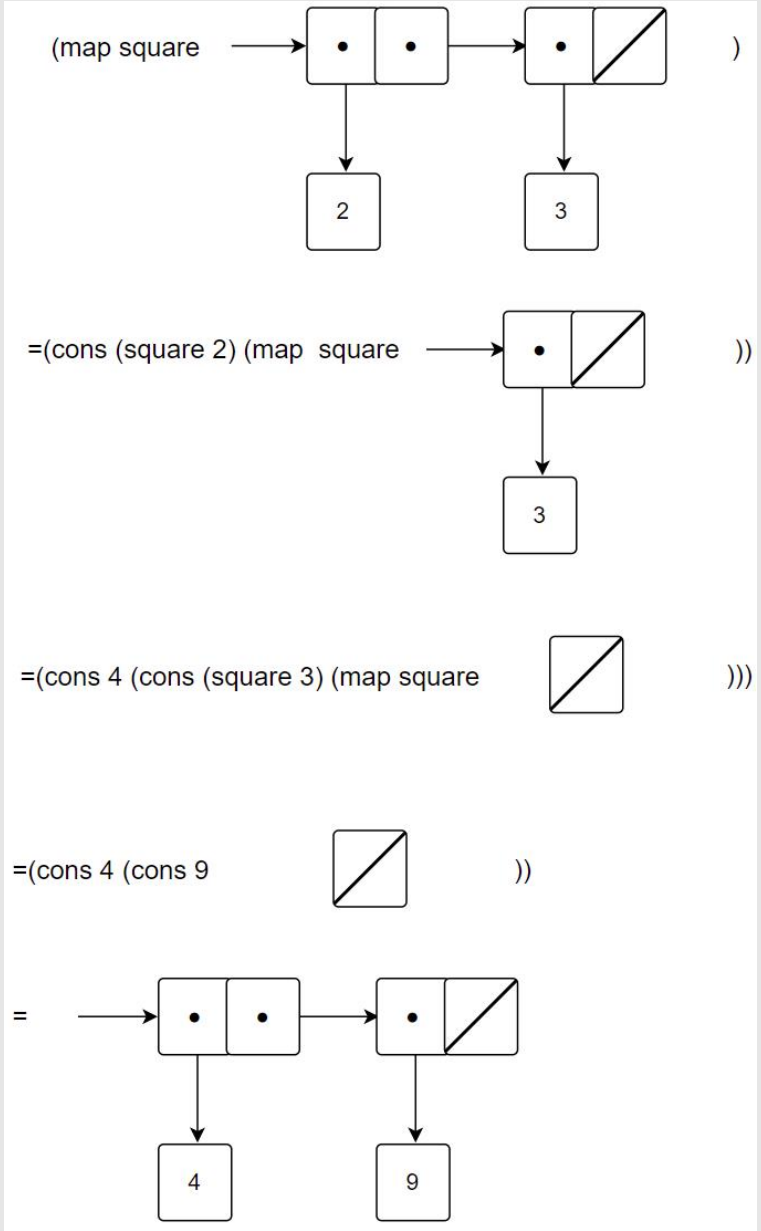
- 涨价：表的元素是(商品ID 价格)，把表中每件商品的价格增加5%
- 选课：表的元素是(学号 选课列表)，把表中每个学生的选课列表部分增添一门课

使用高阶过程实现

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)
- 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拼接表

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


(map square
)
2
3
=(cons (square 2) (map square
))
3
=(cons 4 (cons (square 3) (map square
)))
=(cons 4 (cons 9
))
=
4
9

大纲

- 表的定义和操作
- 树的定义和操作
- 序列作为约定的界面
- 嵌套映射
- 结束

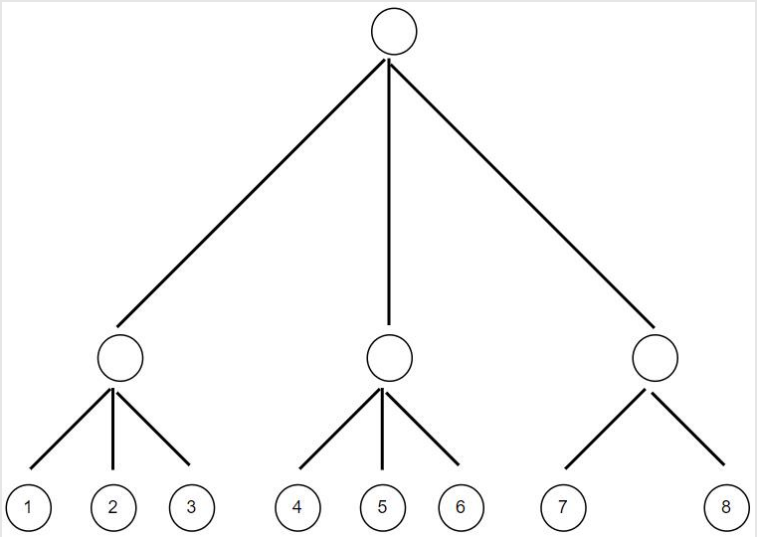
树

- 不同于表，树有着多层结构
- 这里介绍一种把树编码为表的方式
- 2.2中的树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
- 对其他格式的树，在后续章节中会以类似的方式编码

树编码为表的方式

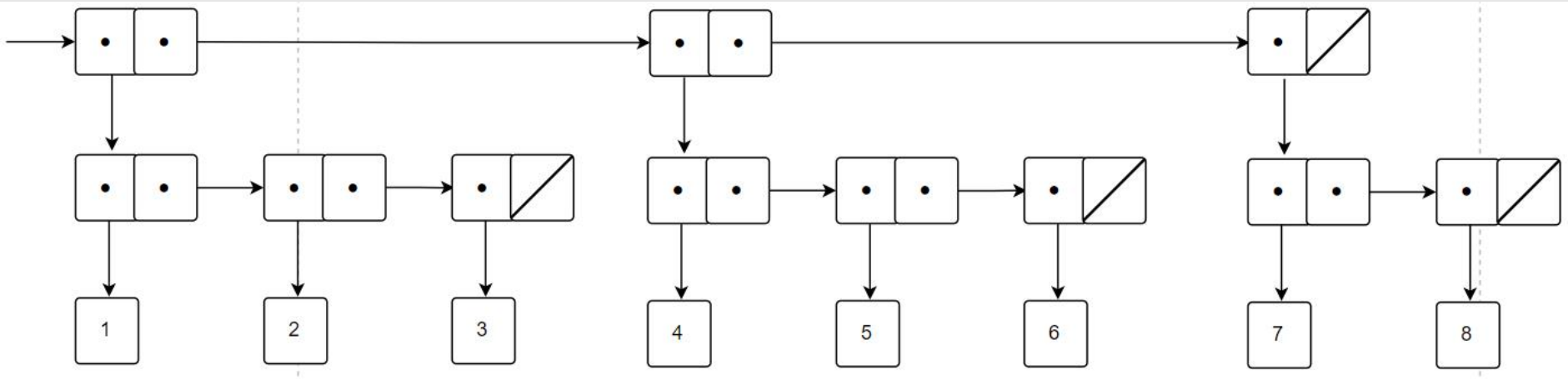
- 一棵树是一个表(list $a_1, \ldots, a_k$)
- a_i是树的第i棵子树
- 树的第i棵子树是叶节点时：a_i是一个数字
- 树的第i棵子树不是叶节点时：a_i是编码这棵子树得到的表

- 例子：这棵树可以编码为表s_1=(list (list 1 2 3) (list 4 5 6)(list 7 8))
- 表s_1有三个元素(list 1 2 3)，(list 4 5 6)和(list 7 8)，分别编码了有着叶节点1、2、3的树，有叶节点4、5、6的树和有叶节点7和8的树三棵子树



(list (list 1 2 3) (list 4 5 6)(list 7 8))的盒子和指针表示

- 最外层表的各个元素，代表树的各个子树
- 第一棵子树的各个元素，正是1、2、3
- 注意多个nil其各自的意义



递归地拆开一棵树

- 树的格式形如(list 子树1 . . . 子树k)，天然适合递归求解
- 有的任务需要遍历树的每个节点，或每个叶节点

递归访问树的节点

- = 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

注意

- 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

一棵树的子树可能有各种情况

- 为空
- 为数字
- 为子树（单元素子树，多元素子树）

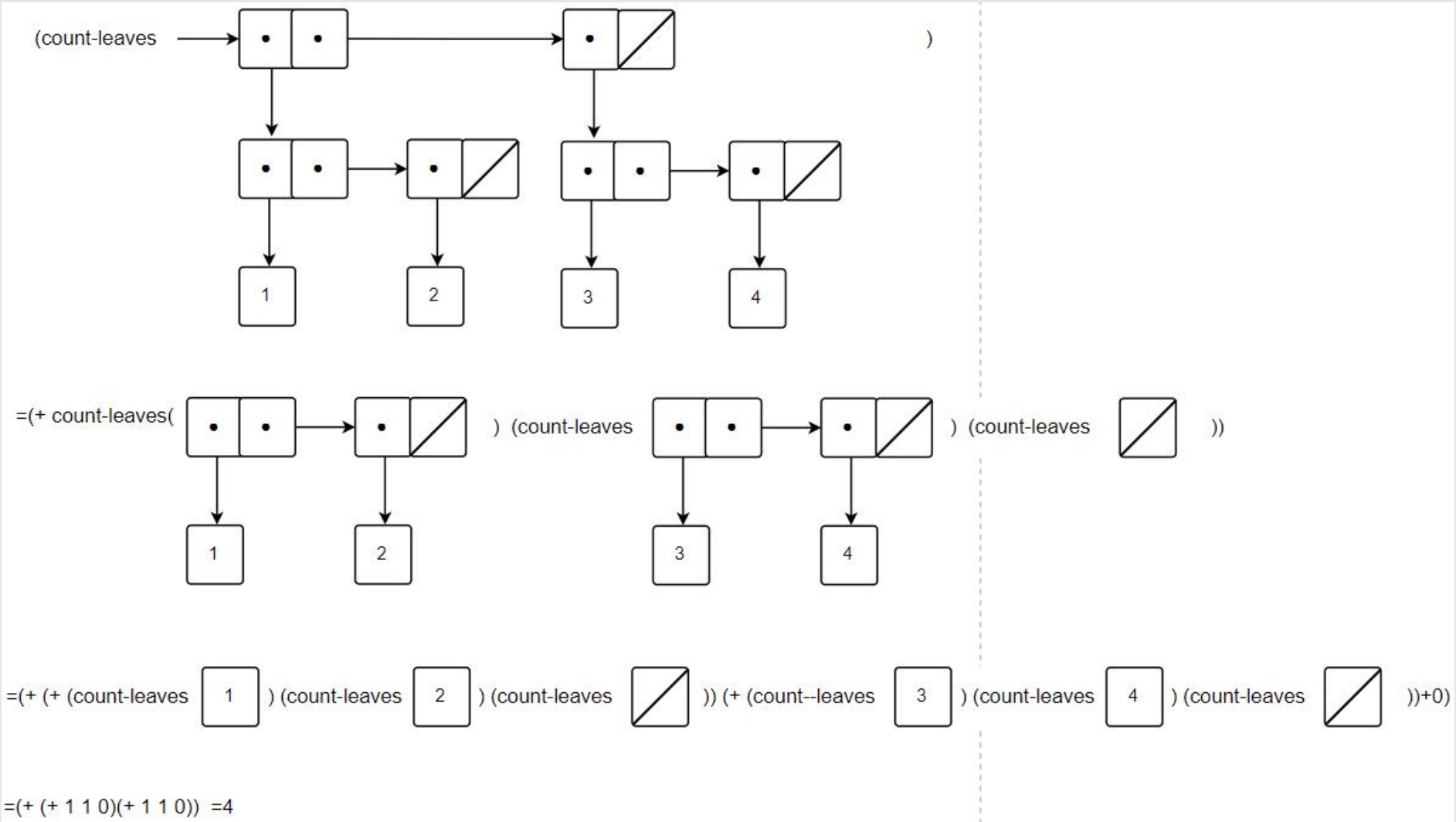
(count-leaves items)

- 返回树items的叶节点个数
- items=nil时：0
- items是一个数字时：1
- 否则：树items的叶节点个数=items的首棵子树的叶节点数+items的其余部分的叶节点数

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

(pair? n)

- Scheme提供的过程
- 判断n是否是一个序对


(count-leaves
)
1
2
3
4
=(+ count-leaves(
) (count-leaves
) (count-leaves
))
1
2
3
4
=(+ (+ (count-leaves
1
) (count-leaves
2
) (count-leaves
)) (+ (count--leaves
3
) (count-leaves
4
) (count-leaves
))+0)
=(+ (+ 1 1 0)(+ 1 1 0)) =4

课堂思考题

- 构造过程sum-leaves，返回一棵树的叶子节点的数值的和
- 例如，(sum-leaves (list (list 1 2) (list 3 4)))=10

对树的映射

树的操作的一种共同模式

- 以同一种方式改变树的每个叶节点（不包含nil）

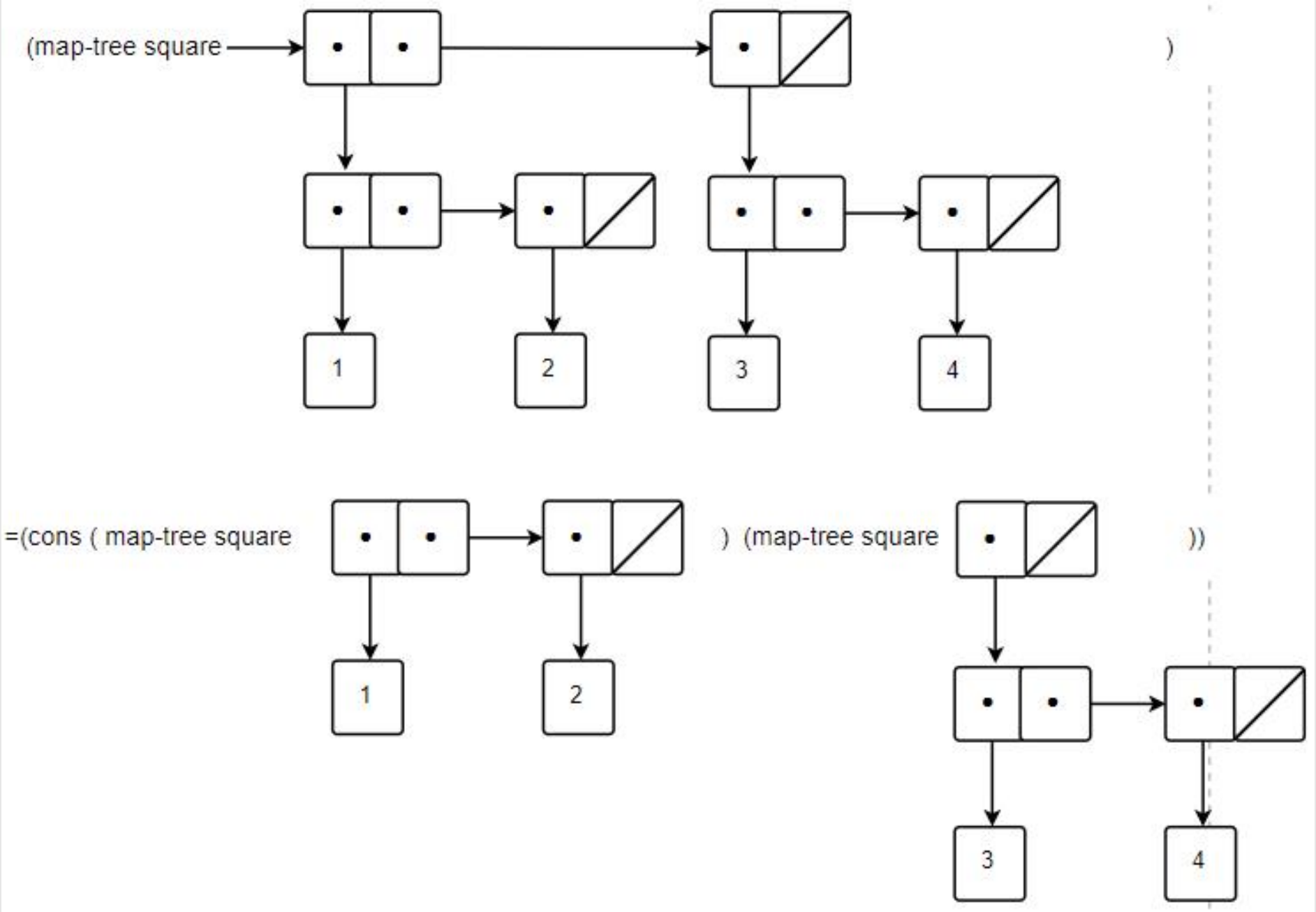
高阶过程map-tree

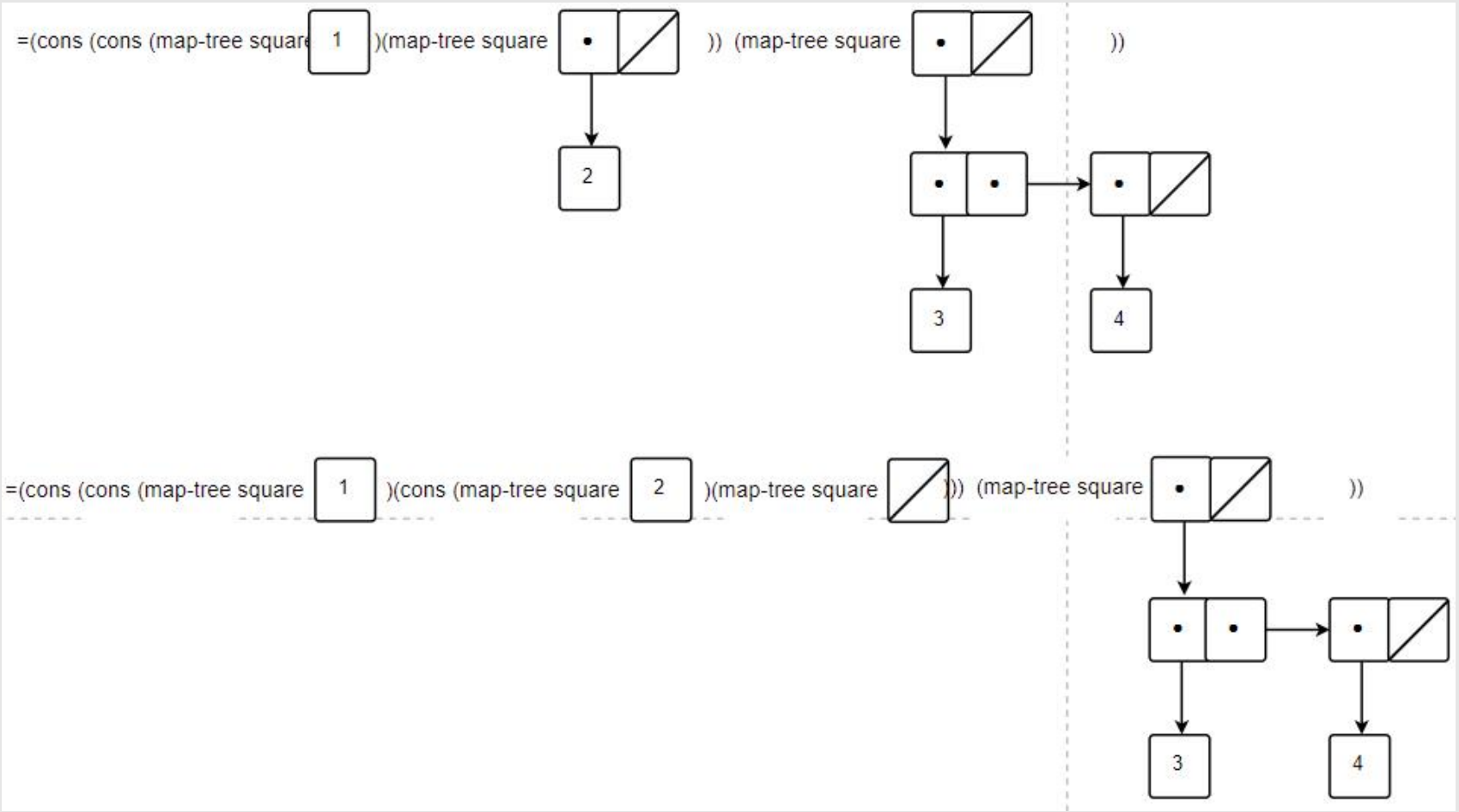
- 求值结果是这样一棵树，它的元素顺序和数目和树items一样，且每个叶节点是通过对树items的相应叶节点做操作proc计算得到的
- 例子：(map-tree square (list (list 1 2) (list 3 4))=((1 4) (9 16))

- 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拼接树

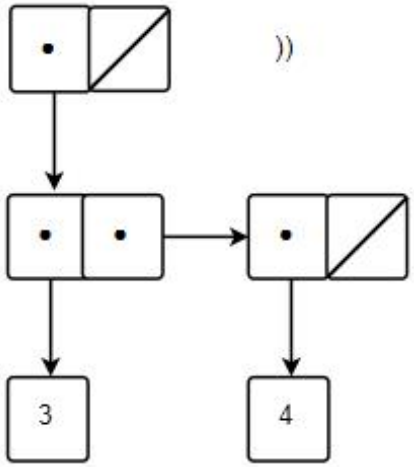
```
(define (map-tree proc items)
  (cond ( (null? items) nil)
        ( (not (pair? items)) (proc items))
        (else (cons (map-tree proc (car items))
                    (map-tree proc (cdr items))))))
```

- 代码中使用cons，见示意图
- 通过递归调用保存的中间信息，生成树


(map-tree square
)
1
2
3
4
=(cons (map-tree square
) (map-tree square
))
1
2
3
4


=(cons (cons (map-tree square
1
)(map-tree square
2
)) (map-tree square
3
4
))
=(cons (cons (map-tree square
1
)(cons (map-tree square
2
)(map-tree square
)) (map-tree square
3
4
))

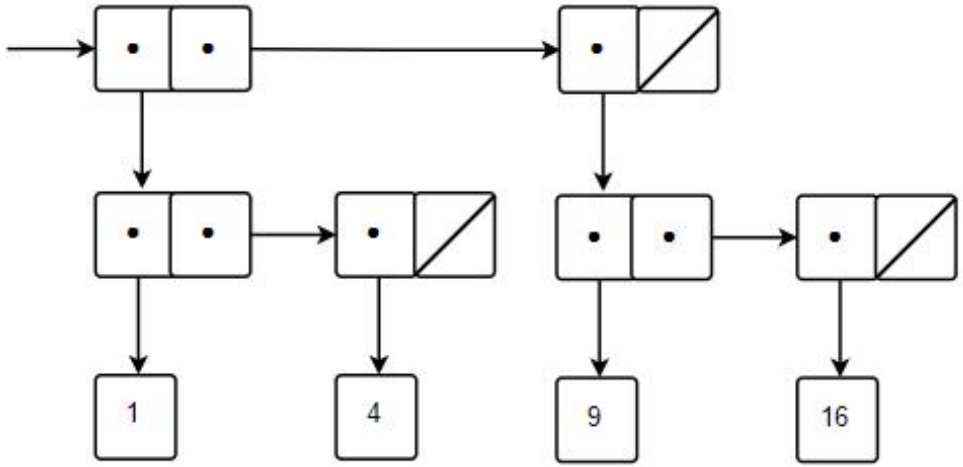
=(cons (cons 1 (cons 4 nil))(map-tree square



))

= (cons (cons 1 (cons 4 nil)) (cons (cons 9 (cons 16 nil)) nil))

=



用另一种方法实现对树的映射

- 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map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

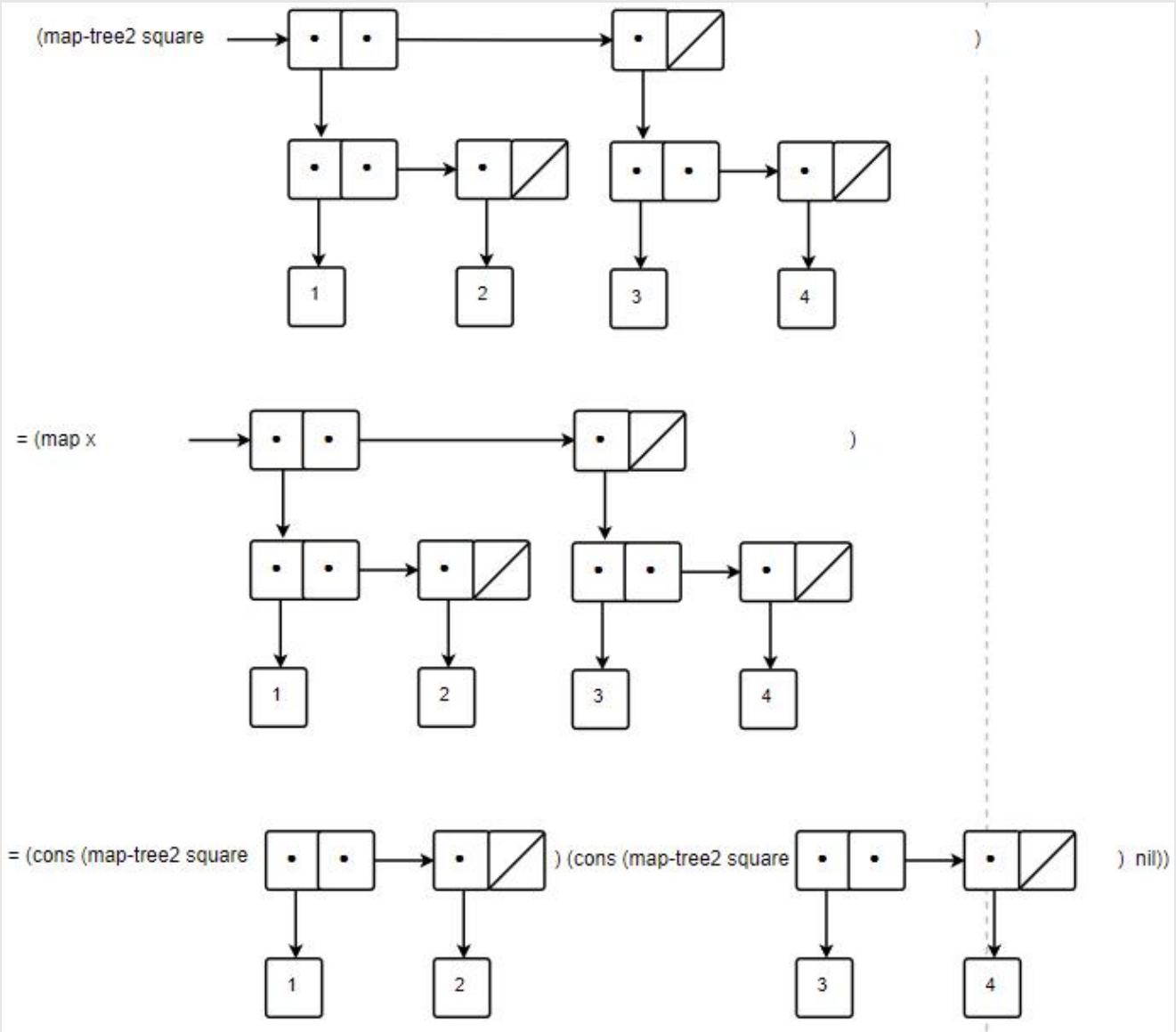
(map-tree2 proc items)

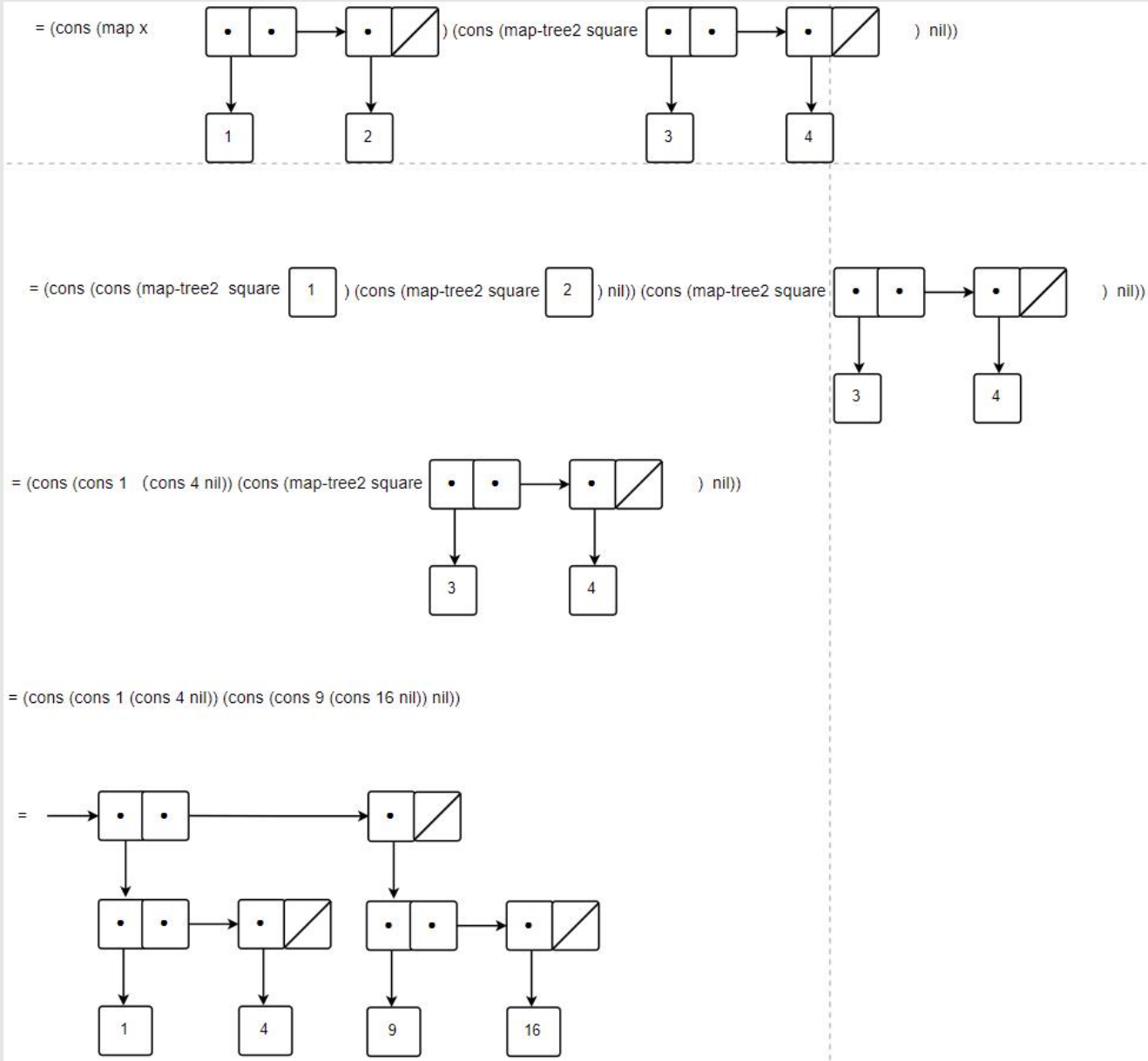
- 如果使用map处理表items，对items的每个元素（子树）subTree执行某个过程f，则这个过程f如何工作
- 由于使用map，无需考虑子树为nil的情况
  - 子树subTree是一棵树时：(f subTree)递归执行(map-tree2 subTree)
  - 子树subTree是一个数字时：(f subTree)的求值结果为(proc subTree)
  - 下面的代码将subTree实现为一个匿名过程

```
(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


(map-tree2 square
)
1
2
3
4
= (map x
)
1
2
3
4
= (cons (map-tree2 square
) (cons (map-tree2 square
) nil))
1
2
3
4


= (cons (map x
) (cons (map-tree2 square
) nil))
1
2
3
4
= (cons (cons (map-tree2 square
1
) (cons (map-tree2 square
2
) nil)) (cons (map-tree2 square
) nil))
3
4
= (cons (cons 1 (cons 4 nil)) (cons (map-tree2 square
) nil))
3
4
= (cons (cons 1 (cons 4 nil)) (cons (cons 9 (cons 16 nil)) nil))
=
1
4
9
16

课堂思考题

- 教材练习2.28
- 构造过程fringe。(fringe items)返回树的叶节点构成的表
- 例子：(fringe (list (list 1 2) 3 (list 4 5))) = (1 2 3 4 5)

大纲

- 表的定义和操作
- 树的定义和操作
- 序列作为约定的界面
- 嵌套映射
- 结束

引入

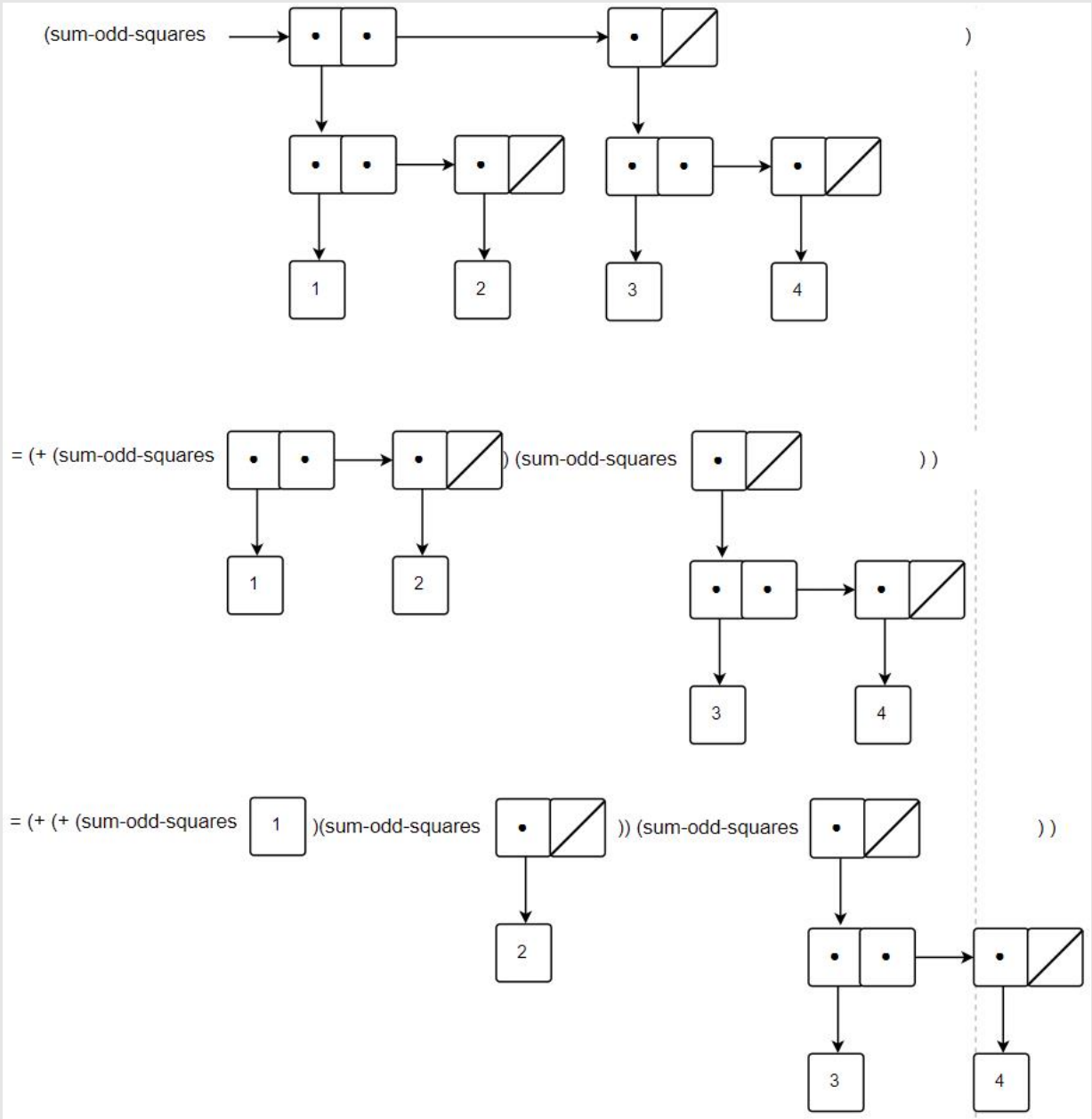
- 处理数据的算法往往依赖于表和树的具体表示
- 递归可以很好地处理表和树，但递归算法和数据的具体表示绑定的很紧
- 寻找数据+计算数据+组合为结果，这三步往往混杂在递归的过程中

问题

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

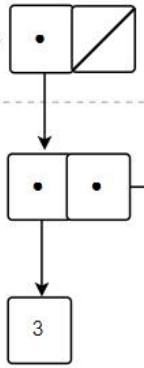
例子：计算树中值为奇数的叶节点的平方和

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

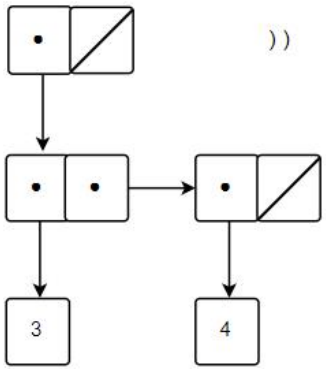

(sum-odd-squares
)
1
2
3
4
= (+ (sum-odd-squares
) (sum-odd-squares
))
1
2
3
4
= (+ (+ (sum-odd-squares
1
)(sum-odd-squares
)) (sum-odd-squares
))
2
3
4

= (+ (+ (sum-odd-squares 

)(+ (sum-odd-squares 

) (sum-odd-squares 
))) (sum-odd-squares 

))

= (+ (+ 1 (+ 0 0))(sum-odd-squares 

))

=(+ (+ 1 (+ 0 0)) (+ 9 (+ 0 0))) = 10

- 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

一种更加自然的解法

- 从图中读出所有叶节点（以表的形式保存）为(1 2 3 4)
- 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

例子：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))
```

(next k)：从第k个元素到第n个元素的斐波那契数中为偶数的数组成的表

- (even-fibs 3) = (next 0)
- = (cons 0 (next 1))
- = (cons 0 (next 2))
- = (cons 0 (next 3))
- = (cons 0 (cons 2 (next 4)))
- = (cons 0 (cons 2 nil)) = (0 2)

- 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)

这两个算法有着很大的不同

- 以树为参数 vs 以数字为参数
- 扫描树 + 处理叶节点 vs 计算斐波那契+求和
- 很难说这两个算法有什么可以被抽取的共性

但这两个算法确实有着共性

- 共性1：计算涉及多个数字，例如叶节点和斐波那契数
- 共性2：不是每个数字都会被计算，一些数字会被过滤掉
- 共性3：每个参与计算的数字需要经过处理，并且会影响计算结果
- 共性4：多个小的计算结果会被汇集起来

这样的共性存在于很多算法中

序列作为约定的界面

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，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

一种新的算法设计思想

- 需要先设计出把算法转化为以表为核心的操作方法
- 然后，根据不同的步骤，选用上面的四种操作，像搭积木一样构造算法

- 算法分为明确的三段：数据收集+数据处理+结果处理

优点

- 生成表，处理表，处理计算结果，这三部分彼此独立
- 算法设计清晰
- 算法可以更好地修改和移植
  - 改动算法：不改动数据生成和处理结果，只改动数据处理
  - 改变数据源：改动数据生成，不改动数据处理和结果处理

例子：sum-odd-squares的设计方法

- 把树的所有叶节点枚举到一个表中
- 过滤掉表中值为偶数的元素
- 把余下的每个元素变为其平方
- 计算表中元素和

例子：even-fibs的新解法

- 构造表(0 ... n)
- 把表的每个元素k变为(fib k)
- 去掉表中值为奇数的元素
- 返回这张表

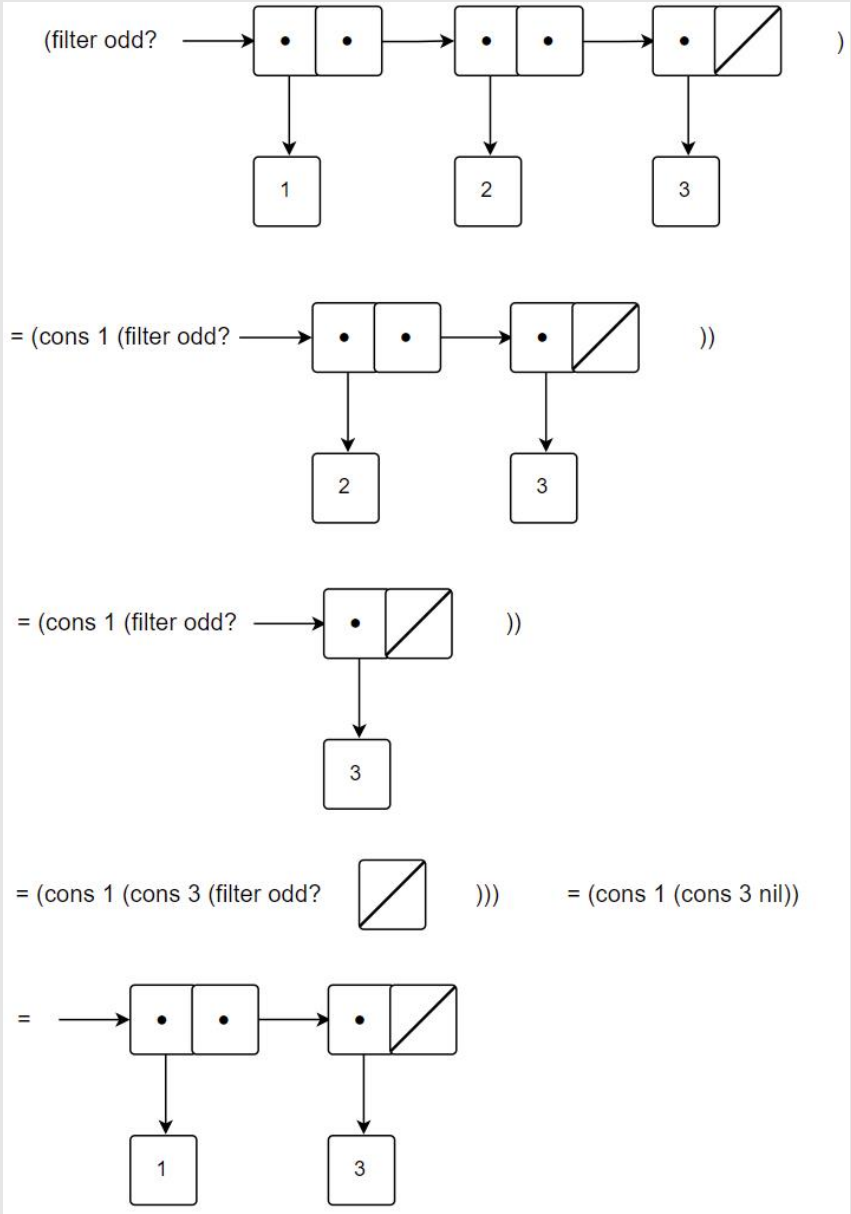
重构代码，明确写出各个部分

- 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

过滤表的元素

- (filter pred items)：过滤表items，只保留满足pred谓词的元素
- filter是基于表元素的值过滤的，而不是基于表元素的下标
- nil保留

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


(filter odd?
)
1
2
3
= (cons 1 (filter odd?
))
2
3
= (cons 1 (filter odd?
))
3
= (cons 1 (cons 3 (filter odd?
)))
= (cons 1 (cons 3 nil))
=
1
3

组合计算结果

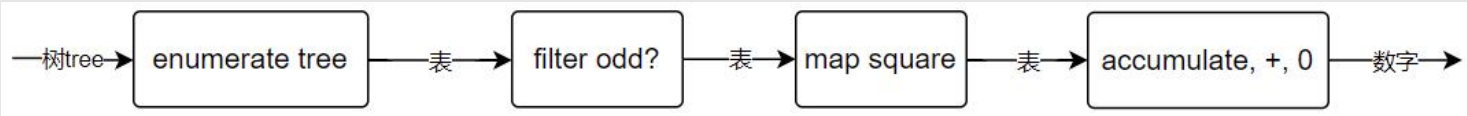
- (accumulate op initial items)：组合表items中已有的计算结果
- 令表items=(a_1, . . ., a_k)
- 则组合出的计算结果为(a_1 op (a_2 . . . (a_k op initial). . .)
- 例如，如果求元素和，则initial=0，op=+
- (a_1 op (a_2 . . . (a_k op initial). . .) = a_1 + a_2 . . . a_k

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

- 如果构造一张和items一样的表，则initial=nil，op=cons
- (a_1 op (a_2 . . . (a_k op initial). . .) = (list a_1 . . . a_k)

- (accumulate op initial (list a b c))
- = (op a (accumulate op initial (b c)))
- = (op a (op b (accumulate op initial (c))))
- = (op a (op b (op c (accumulate op initial nil))))
- = (op a (op b (op c initial)))

重写sum-odd-squares



- (enumerate-tree tree)：生成树tree的叶节点的表
- 计算顺序是enumerate-interval，filter，map，accumulate
- 代码是(accumulate + 0 (map square (filter odd? (enumerate-tree tree))))
- (accumulate + 0 s)用来求表s中元素的和

注意

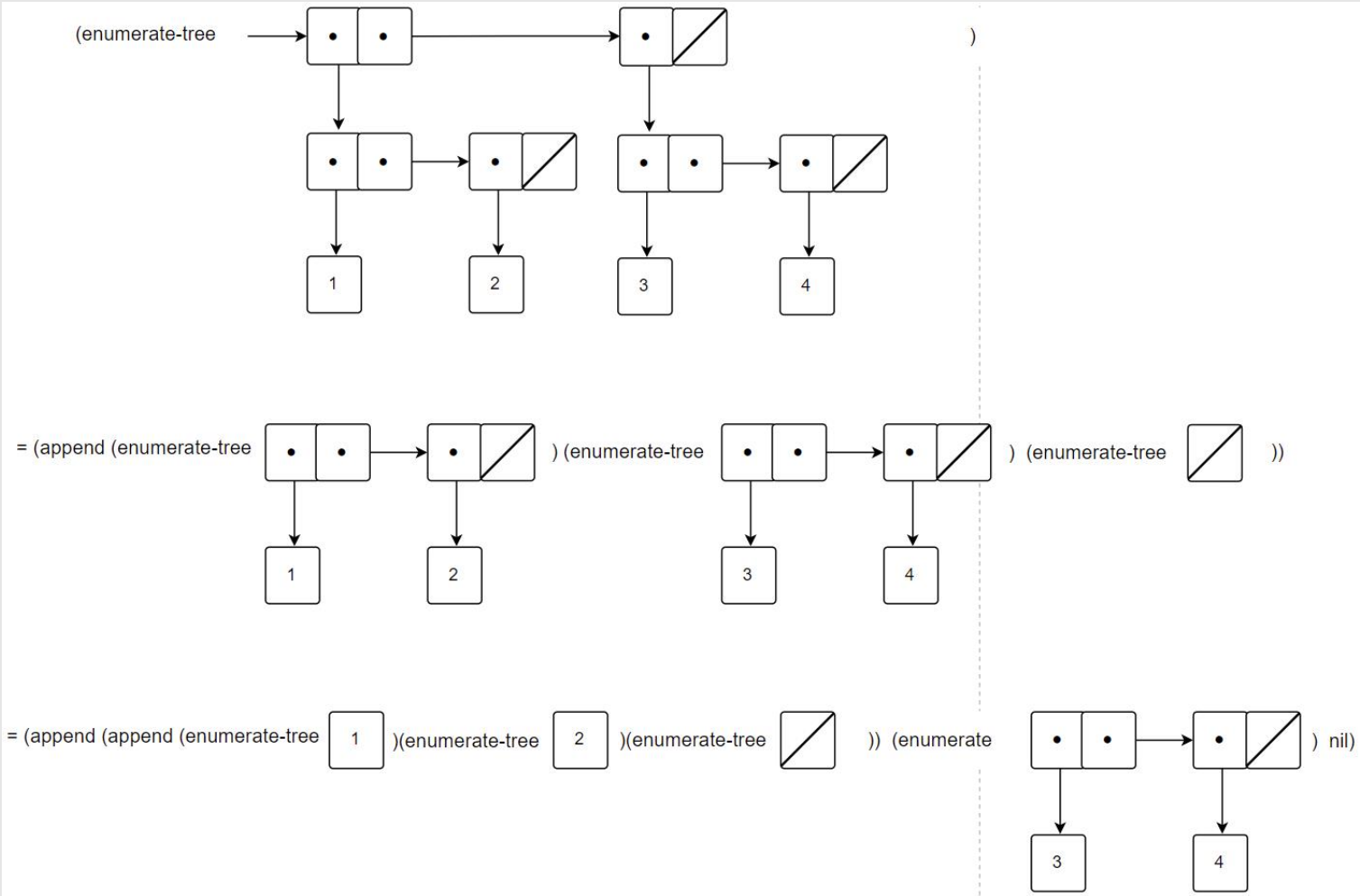
- 新算法的优点是更加清晰，更易于抽象。只从代码长度看的话，之前的算法代码更短

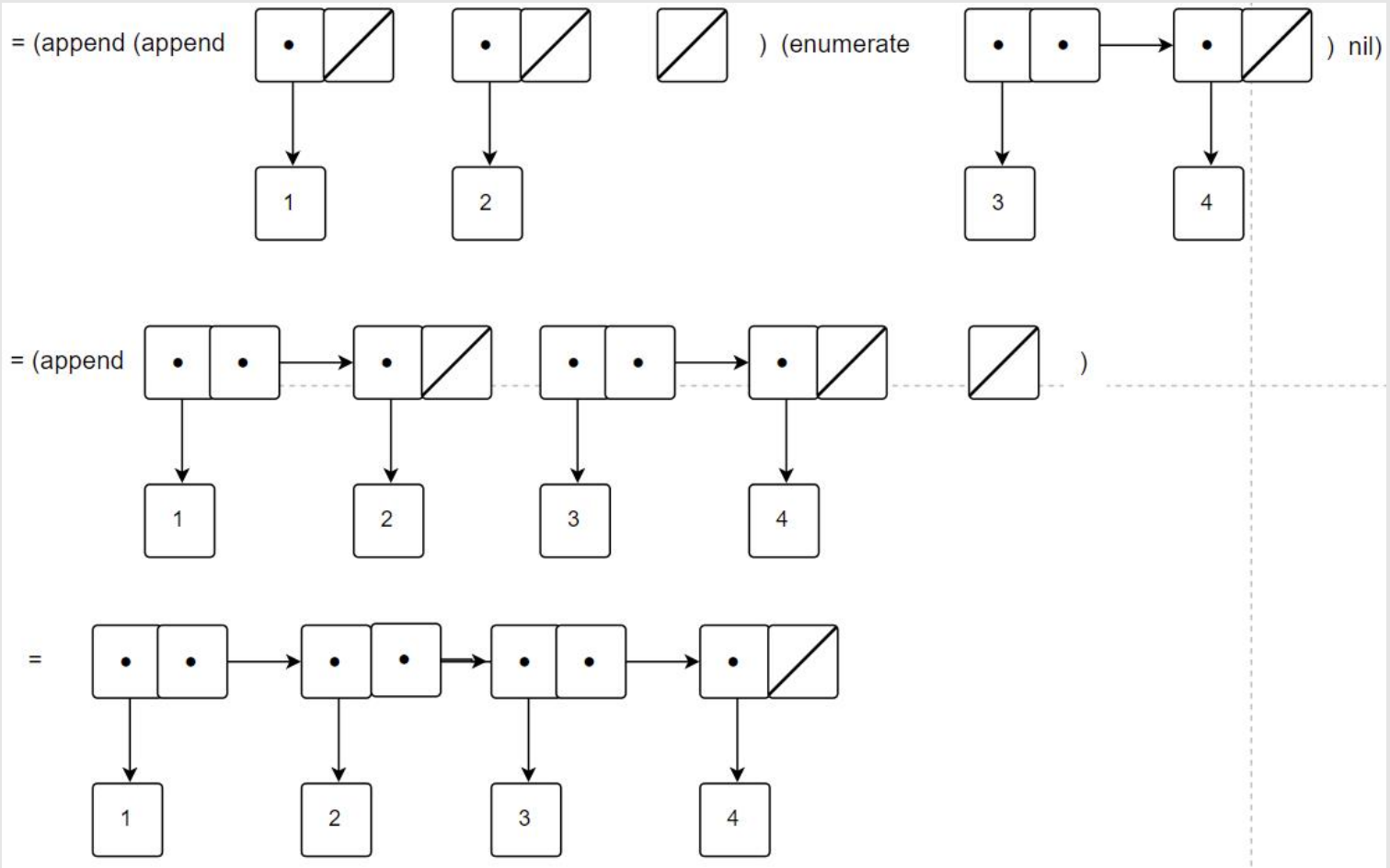
(enumerate-tree tree)

- tree为nil：nil
- tree为一个数字，即叶节点：(list tree)，这里需要结果为表
- 否则，tree的叶节点表=append(tree的第一棵子树的叶节点表，tree余下部分的叶节节点表)
- 之所以使用append，是因为“第一棵子树的叶子节点表”和“余下部分的叶子节点表”都是表，合并表需要append

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

出于绘图简便，这里把(append x (append y z))写为了(append x y z)

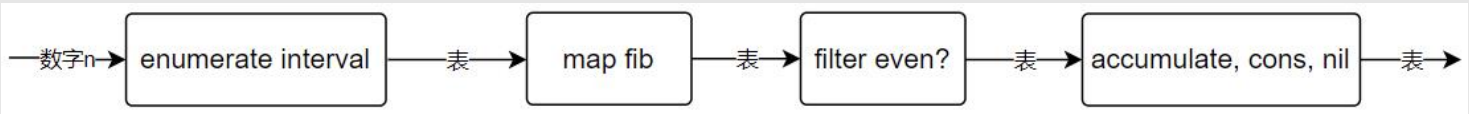



= (append (append
) (enumerate
) nil)
1
2
3
4
= (append
)
1
2
3
4
=
1
2
3
4

新的sum-of-squares代码

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

重写even-fibs



- (enumerate-interval low high)：构造表(low . . . high)
- 计算顺序是enumerate-interval，map，filter，accumulate
- 代码是(accumulate cons nil (filter even? (map fib (enumerate-interval low high))))
- (accumulate cons nil items)的执行结果是表items

- (enumerate-interval low high)

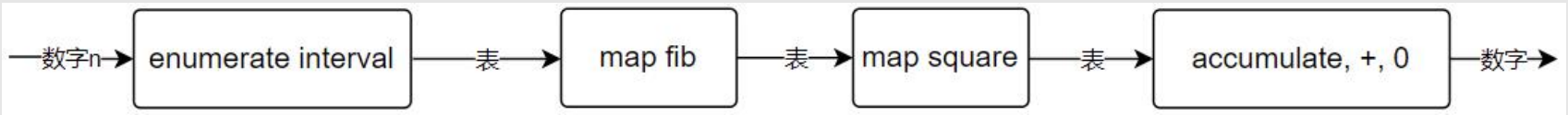
```
(define (enumerate-interval low high)
    (if (> low high)
        nil
        (cons low (enumerate-interval (+ low 1) high))))
```

- (enumerate-interval 0 2)
- = (cons 0 (enumerate 1 2))
- = (cons 0 (cons 1 (enumerate 2 2)))
- = (cons 0 (cons 1 (cons 2 (enumerate 3 2))))
- = (cons 0 (cons 1 (cons 2 nil)))

使用“序列作为约定的界面”

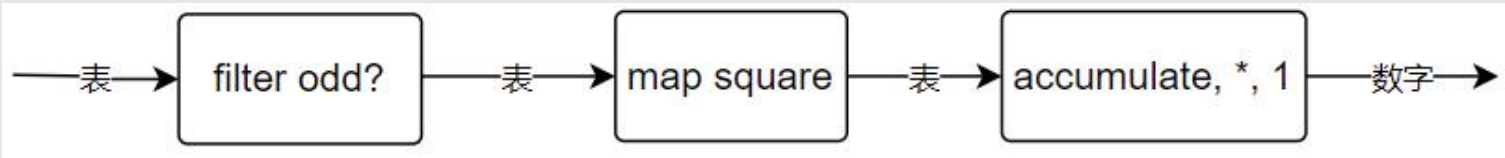
- 构造作为输入的表
- 把算法构造为对表的各种处理
- 构造从处理完的表中计算出结果的方法
- 对表的处理方法并不固定，用几次filter和map，每次的顺序及作为参数的过程，都是不固定的

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)))))
```

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))))
```

课堂思考题

- 教材练习2.33：使用累积定义一些基本的表操作
- (define (map p sequence) (accumulate (lambda (x y) <>) nil sequence))
- (define (append seq1 seq2) (accumulate cons <> <>))
- (define (length sequence) (accumulate <> 0 sequence))

大纲

- 表的定义和操作
- 树的定义和操作
- 序列作为约定的界面
- 嵌套映射
- 结束

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
- 这样的情况称为嵌套映射

例子：(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

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

这个表格不理想，因为((3 1) (3 2))多了一层括号

- g(1) append g(2) append g(3) append g(4)，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))

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)，过滤掉和不是质数的二元组
- X3 = ((2 1) (3 2) (4 1) (4 3))

距离最终结果，还需要把二元组改为三元组

- 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

过程(g i)：构造为匿名过程

- g(i)的值为一张表，其中每个元素为表(i u)且$1 \leq u < i$。例如，g(4)=((4 1) (4 2)(4 3))
- 使用map实现这个匿名过程
- 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)

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

prime-sum?和make-pair-sum

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

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

prime-sum-pairs

```
(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
- (define (flatmap proc seq) (accumulate append nil (map proc seq)))
- 可以简化一些代码

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

(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

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）
- 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]

(permutations s)：以s=(1 2 3)为例

- 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)))

这个表格不理想，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)
- 处理之后，结果为((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))

过程(remove item sequence)：从表sequences中去掉item

- 例如，(remove 3 (list 1 2 3 4))=(1 2 4)
- 使用filter实现

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

过程(g x)：构造为匿名过程

- 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))

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

注意，本例中的p为(2 3)或(3 2)，注意(cons x p)中的顺序

permutations

```
(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))))
```

- 为何(null? s)时结果为(list nil)而不是nil
- 以(permutations (list 1))为例
- (permutations (list 1)) = (append X nil)，其中X = (map (lambda (p) (cons 1 p)) (permutations (remove 1 (1)))) = (map (lambda (p) (cons 1 p)) (permutations nil))
- 如果(permutatios nil)=nil，则无法被map
- 而(permutations nil) = (list nil)则可以被map，此时X=((1))

大纲

- 表的定义和操作
- 树的定义和操作
- 序列作为约定的界面
- 嵌套映射
- 结束

课堂总结

我们掌握了表这一核心数据结构

- 表的定义和操作
- 使用表构造树
- 以表为核心的编程思想
- 嵌套映射

Scheme语法

- list
- null?，pair?

类似于1.2节，学习的难点不在于Scheme的语法，而在于编程思想

本节内容与其他知识的联系

在后续章节中会进一步进展的

- 后面章节会遇到其他形式的树及其编码方法
- 第三章会讲述如何修改表和树，而不是从头构造一个
- 第三章的3.5节介绍流

1.2节涉及数据结构课程，前面已经叙述过

预告

- 目前我们还不能使用文字，只能使用数字
- 在2.3节，我们将引入文字，让我们的程序能力更加强大

2.2节练习题

练习级

- 教材练习2.17
- 教材练习2.21
- 教材练习2.22
- 教材练习2.41

挑战级

- 教材练习2.19
- 教材练习2.27 给了答案
- 教材练习2.32 给了答案
- 教材练习2.42 给了答案

- 2.2节的代码难度变高了
- 需掌握递归、高阶过程、以表为核心的编程
- 需清晰了解程序执行过程中发生了什么

下课