[读书笔记]SICP：6[B]符号数据

本章对应于书中的2.3。

- 开发一个系统时，通常会采用同样的数据抽象策略。首先并不考虑抽象对象是如何实现的，根据wishful thinking，可以假设能获得该抽象对象的各种成分，先定义一套在抽象对象上操作的目标过程。其次考虑如何实现抽象对象的细节，由此过后只需要修改抽象对象的实现而无需修改目标过程。
- 尽量使用已封装好的过程

之前的复合数据都是基于数值进行构造的，本章将用任意符号作为数据。

1 引号

```
(define a 1)
(define b 2)
(define c 3)
```

当我们定义了以上符号时，如果我们尝试对`(list a b c)`进行求值，则会得到`(1 2 3)`。我们这里需要将表和符号标记为应该作为数据对象，而不应该作为求值的表达式，可以在想要作为数据对象的符号前加上一个`'`。比如

```
(list 'a b c) ;结果为(a 2 3)
```

同样也能将`'`用于复合对象

```
'(a b c)
```

这里的`'()`将不对括号内的所有符号进行求值，且返回的是一个序列数据`(a b c)`，可以通过`car`和`cdr`进行操作。由此我们也可以直接用`'()`来表示空表。

对于符号，Scheme提供了一个基本过程`eq?`来判断两个符号是否相同，比如

```
(define (memq item x)
  (cond ((null? x) false)
        ((eq? item (car x)) x)
        (else (memq item (cdr x)))))
```

2 实例

开发一个系统时，通常会采用同样的数据抽象策略。首先并不考虑抽象对象是如何实现的，根据wishful thinking，可以假设能获得该抽象对象的各种成分，先定义一套在抽象对象上操作的目标过程。其次考虑如何实现抽象对象的细节，由此过后只需要修改抽象对象的实现而无需修改目标过程。

2.1 符号求导

设计一个过程以一个代数表达式和一个变量为参数，返回这个表达式相对该变量的导数。这里只关注由两个参数的加和乘法运算构建起来的表达式。求导规则为

$$\frac{dc}{dx} = 0, \quad \text{for } c \text{ a constant or a variable different from } x,$$

$$\frac{dx}{dx} = 1,$$
$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$
$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}.$$

2.1.1 wishful thinking和目标过程

首先假设：

- 如果有一种表示代数表达式的方式，则有判断该表达式是否为和式、乘式、常量或变量，并且能提取出表达式的各个部分
- 希望能提取出和式的加数和被加数，乘式的乘数和被乘数
- 希望能从几部分触发构造出整个表达式
- 假定已实现了以下过程

```
(variable? e)           是否为变量
(same-variable? v1 v2)  v1和v2是否为相同变量
(sum? e)                e是否为和式
(addend e)              和式e的被加数
(augend e)              和式e的加数
(make-sum a1 a2)        构造起a1和a2的和式
(product? e)            e是否为乘式
(multiplier e)          乘式e的被乘数
(multiplicand e)        乘式e的乘数
(make-product m1 m2)    构造起m1和m2的乘式
```

基于以上假设，我们就能构建起目标过程

```
(define (deriv exp var)
  (cond ((number? exp) 0) ;第一条求导规则
        ((variable? exp) (if (same-variable? exp var) 1 0)) ;第二条求导规则
        ((sum? exp) (make-sum (deriv (addend exp) var) ;第三条求导规则
                              (deriv (augend exp) var)))
        ((product? exp) (make-sum ;第四条求导规则
                              (make-product (multiplier exp)
                                            (deriv (multiplicand exp) var))
                         (make-product (deriv (multiplier exp) var)
                                       (multiplicand exp))))
        (else (error "unknown expression
                      type: DERIV" exp))))
```

2.1.2 抽象对象

接下来我们就能考虑抽象对象的具体实现，这里使用和Lisp相同的前缀表达式方法，传入的会是符号的list。此时我们就能定义上面假设的内容了

```
; 构造函数
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
; 选择函数
```

```
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))
; 基本函数
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
(define (product? x)
  (and (pair? x) (eq? (car x) '*)))
```

但是我们会发现通过这种定义方式会使得求导出来的内容十分冗余

```
(deriv '(+ x 3) 'x) ;(+ 1 0)

(deriv '(* x y) 'x) ;(+ (* x 0) (* 1 y))

(deriv '(* (* x y) (+ x 3)) 'x)
;(+ (* (* x y) (+ 1 0))
;   (* (+ (* x 0) (* 1 y))
;      (+  x 3)))
```

此时不应该去该目标过程，应该通过修改抽象对象的表达来进行简化，可以对`make-sum`和`make-product`修改为以下形式

```
(define (=number? exp num)
  (and (number? exp) (= exp num)))
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
```

此时结果就会好一些

```
(deriv '(+ x 3) 'x) ;1
(deriv '(* x y) 'x) ;y
(deriv '(* (* x y) (+ x 3)) 'x) ;(+ (* x y) (* y (+ x 3)))
```

2.2 集合的表示

之前构造的有理数系统和代数表达式系统都是使用表的形式来表示的，而这里我们探讨集合的不同表示形式，并实现不同表示形式下的`union-set`,`intersection-set`,`element-of-set?`和`adjoin-set`过程。

2.2.1 集合作为未排序的表

```
;;; 判断元素 O(n)
(define (element-of-unsorted-set? x set)
  (cond ((null? set) false)
```

```
        ((equal? x (car set)) true)
        (else (element-of-unsorted-set? x (cdr set)))))
;;; 添加元素 O(n)
(define (adjoin-unsorted-set x set)
  (if (element-of-unsorted-set? x set)
      set
      (cons x set)))
;;; 交集 O(n*m)
(define (intersection-unsorted-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-unsorted-set? (car set1) set2)
         (cons (car set1) (intersection-unsorted-set (cdr set1) set2)))
        (else (intersection-unsorted-set (cdr set1) set2))))
;;; 并集 O(n*m)
(define (union-unsorted-set set1 set2)
  (cond ((null? set1) set2)
        ((element-of-unsorted-set? (car set1) set2) (union-unsorted-set (cdr set1)
set2))
        (else (cons (car set1) (union-unsorted-set (cdr set1) set2)))))
```

可以发现，这里的`element-of-set?`过程的增长阶为`O(n)`。而这里要求集合中不允许存在重复的元素，所以在添加元素的`adjoin-set`过程中首先要判断元素`x`是否存在于集合中，所以其增长阶也为`O(n)`。而计算交集和并集的过程都需要依次判断两个集合中的元素是否存在于另一个集合中，所以增长阶都为`O(n^2)`。如果我们放宽不要求集合中的元素可以重复，则`adjoin-set`过程直接通过`cons`就能完成，增长阶为`O(1)`，而并集直接使用`append`过程则增长阶为`O(n)`。

2.2.2 集合作为排序的表

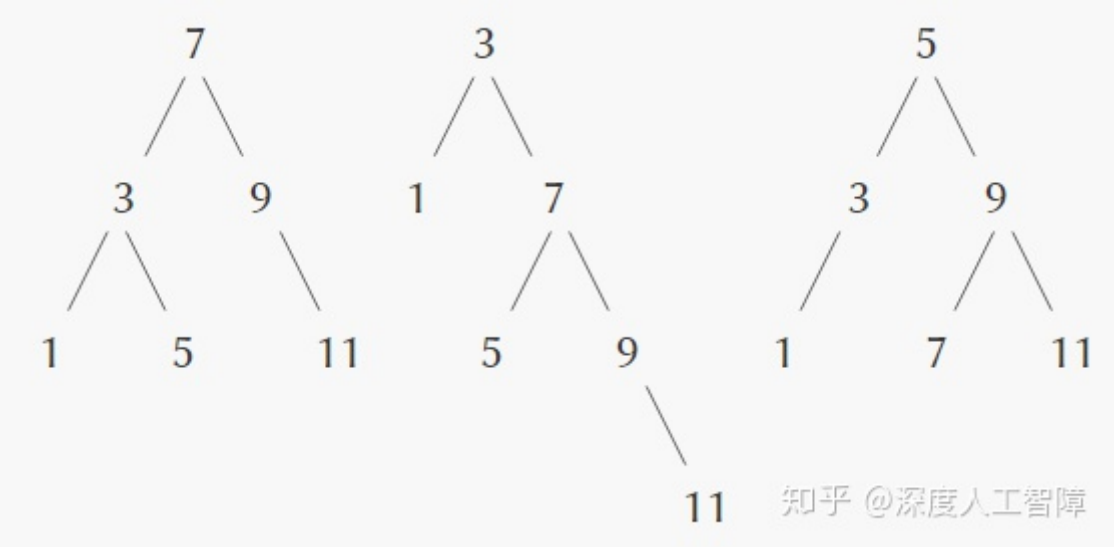
此时需要定义集合中元素的比较方式，这里我们简化集合中只包含数字，然后将集合构建成一个升序的表。对应的代码为

```
;;; 判断元素 O(n)
(define (element-of-sorted-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-sorted-set? x (cdr set)))))
;;; 添加元素 O(n)
(define (adjoin-sorted-set x set)
  (cond ((or (null? set) (< x (car set))) (cons x set))
        ((= x (car set)) set)
        (else (cons (car set) (adjoin-sorted-set x (cdr set))))))
;;; 交集 O(n+m)
(define (intersection-sorted-set set1 set2)
  (if (or (null? set1) (null? set2)) '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2) (cons x1 (intersection-sorted-set (cdr set1) (cdr set2))))
              ((< x1 x2) (intersection-sorted-set (cdr set1)  set2))
              ((< x2 x1) (intersection-sorted-set set1 (cdr set2)))))))
;;; 并集 O(n+m)
(define (union-sorted-set set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else
         (let ((x1 (car set1)) (x2 (car set2)))
           (cond ((= x1 x2) (cons x1 (union-sorted-set (cdr set1) (cdr set2))))
                 ((< x1 x2) (cons x1 (union-sorted-set (cdr set1) set2)))
                 ((> x1 x2) (cons x2 (union-sorted-set set1 (cdr set2)))))))))
```

可以发现，由于集合进行了排序，所以在进行交集和并集操作时，集合1中的每个元素无需在集合2中进行完整遍历，只需要两个集合交替依次比较各个元素就好，这两个操作的增长阶都降低为`O(n)`。

2.2.3 集合作为二叉树

更近一步，我们可以将集合表示为如下的二叉树的形式



其中每个节点为一个数据，而该节点左子树中的数据都比该节点小，右子树中的数据都比该节点大。当我们尝试检索某个值`x`时，如果当前节点比`x`小，则我们只需要在右子树中进行检索，如果当前节点比`x`大，则我们只需要在左子树进行检索。当二叉树是平衡时，每一步的检索都能时问题减小一般，使得检索的增长阶为`O(logn)`。

我们可以将二叉树表示为表的形式：`(数据项 左子树 右子树)`，则构造函数和选择函数为

```
 ;;; 构造函数
(define (make-tree entry left right)
  (list entry left right))
;;; 选择函数
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
;;; 基本函数
;;;; 判断元素 O(logn)
(define (element-of-tree-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set)) (element-of-tree-set? x (left-branch set)))
        ((> x (entry set)) (element-of-tree-set? x (right-branch set)))))
;;;; 添加元素 O(logn)
(define (adjoin-tree-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((> x (entry set)) (make-tree (entry set)
                                      (left-branch set)
                                      (adjoin-tree-set x (right-branch set))))
        ((< x (entry set)) (make-tree (entry set)
                                      (adjoin-tree-set x (left-branch set))
                                      (right-branch set)))))
;;;; 树转化为表
(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
       (cons (entry tree) (tree->list-1 (right-branch tree))))))
```

```
(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list (right-branch tree) result-list)))))
  (copy-to-list tree '()))
;;;; 表转化为树
(define (list->tree elements)
  (car (partial-tree elements (length elements))))
(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
                (non-left-elts (cdr left-result))
                (right-size (- n (+ left-size 1))))
            (let ((this-entry (car non-left-elts))
                  (right-result (partial-tree (cdr non-left-elts) right-size)))
              (let ((right-tree (car right-result))
                    (remaining-elts (cdr right-result)))
                (cons (make-tree this-entry left-tree right-tree) remaining-
elts))))))))
;;;; 交集
(define (intersection-tree-set set1 set2)
  (cond ((null? set1) '())
        ((null? set2) '())
        (else (list->tree (intersection-sorted-set (tree->list-1 set1) (tree->list-1
set2))))))
;;;; 并集
 (define (union-set-bal set1 set2)
   (cond ((null? set1) set2)
         ((null? set2) set1)
         (else (list->tree (union-sorted-set (tree->list-1 set1) (tree->list-1
set2))))))
```

这里的交集和并集的实现都是首先将数通过`tree->list-1`过程将其转化为有序表，再使用上一小节中的交集和并集过程来实现，然后再通过`list->tree`过程将有序表结果转化为树，所以这里的增长阶为`O(n)`。

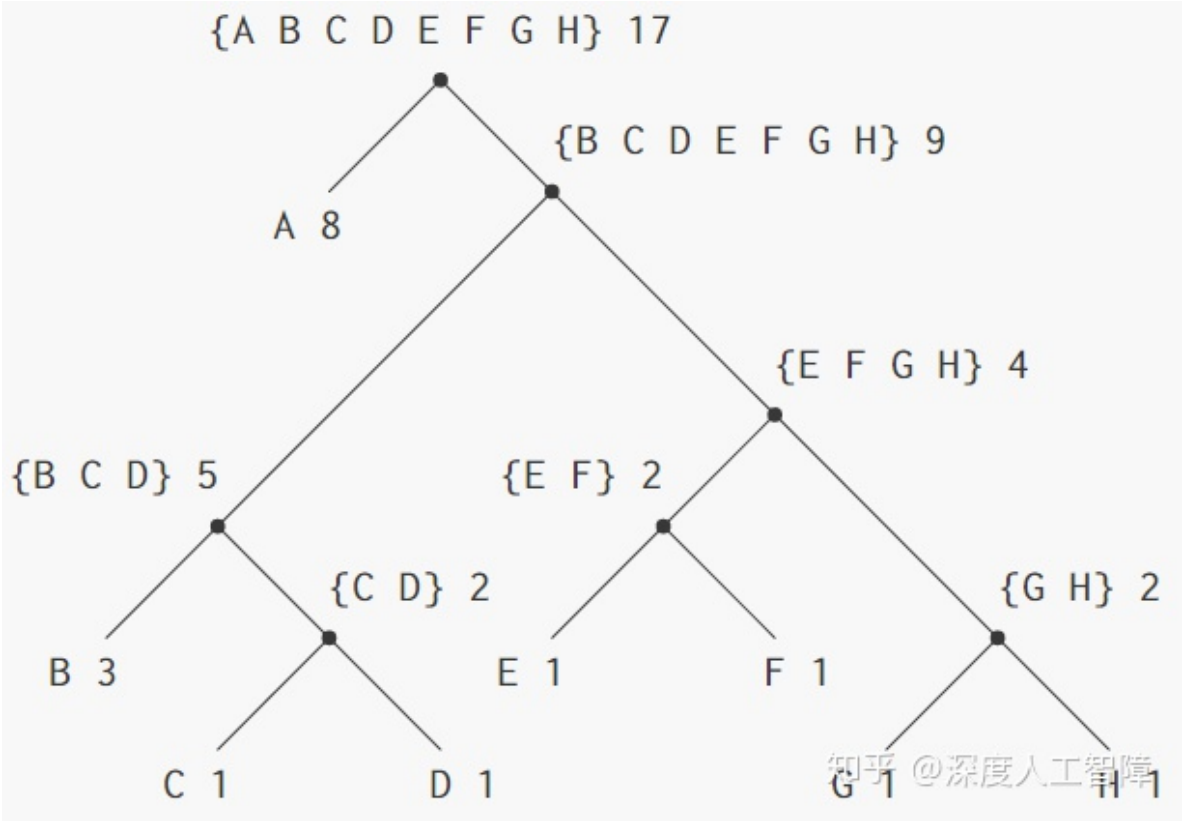
需要注意的是，二叉树只有在平衡时的增长阶才是`O(logn)`的，当我们按序插入元素时，会使得二叉树退化为有序表，此时的增长阶就退化为了`O(n)`。我们可以在执行若干步`adjoin-tree-set`过程就执行一次平衡过程，也可以使用类似B树和红黑树这类数据结构。

2.2.4 集合总结

可以发现我们采用不同的数据对象来表示集合会有不同的性能，而集合其实与信息检索任务密切相关，数据库可以表示为一个记录的集合，而数据管理系统会将大量时间用于访问和修改这些记录。

2.3 霍夫曼树

这里探索霍夫曼编码，一个霍夫曼编码可以表示为一颗二叉树，其中树叶是被编码的符号和对应的权重，非树叶节点代表一个集合，其中包含这个节点之下的所有树叶上的符号和树叶权重之和。



首先根据以上定义，创建树叶和树的构造函数和选择函数。其中每个树叶用一个表来表示，包含`('leaf symbol weight)`

```
; 树叶
;; 构造函数
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))
;; 选择函数
(define (symbol-leaf leaf) (cadr leaf))
(define (weight-leaf leaf) (caddr leaf))
;; 基本函数
(define (leaf? object)
  (eq? (car object) 'leaf))
```

而树也用一个表来表示，包含`(left right symbols weight)`，可以通过第一个元素不是`'leaf`来判断不是树叶

```
; 树
;; 构造函数
(define (make-code-tree left right)
  (list left                                    ;左子树
        right                                   ;右子树
        (append (symbols left) (symbols right)) ; 符号集
        (+ (weight left) (weight right))))      ; 权重
;; 选择函数
(define (left-branch tree) (car tree))
(define (right-branch tree) (cadr tree))
(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (cadddr tree)))
```

当我们获得一个霍夫曼树时，可以将输入的编码进行解码，其中`0`转向左子树，`1`转向右子树，当遇到树叶时就通过树叶的符号确定当前解码的符号，对应的代码为

```
 ;; 解码
(define (decode bits tree)
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch (choose-branch (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch)))))
  (decode-1 bits tree))
;;; 通过编码bit确定下一分支
(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error "bad bit: CHOOSE-BRANCH" bit))))
```

编码过程则是依次遍历每个符号，在霍夫曼树中判断该符号到树叶节点的路径

```
;; 编码
(define (encode message tree)
  (if (null? message)
      '()
      (append
       (encode-symbol (car message) tree)
       (encode (cdr message) tree))))
;;; 判断元素 O(n)
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))
;;; 在树中获得对应符号的编码
(define (encode-symbol symbol tree)
  (if (leaf? tree)
      '() ;到树叶就一定是正确的符号
      (let ((left-symbols (symbols (left-branch tree)))
            (right-symbols (symbols (right-branch tree))))
        (cond ((element-of-set? symbol left-symbols)
               (cons '0 (encode-symbol symbol (left-branch tree))))
              ((element-of-set? symbol right-symbols)
               (cons '1 (encode-symbol symbol (right-branch tree))))
              (else (error "bad symbol: ENCODE-SYMBOL" symbol))))))
```

而想要生成霍夫曼树，首先需要将树叶和树放到一个集合中，根据权重来升序排序，则每次可以挑选前两个元素来构建一颗新的树。所以首先探索如何将新的元素（树叶或树）加入该有序集合中

```
;;; 加入元素
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set) (adjoin-set x (cdr set))))))
```

而对应的创建霍夫曼树的代码如下，会首先通过`make-leaf-set`过程来根据符号-权重对偶表来创建出有序集合，然后使用`successive-merge`根据有序表来创建霍夫曼树

```
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))
(define (make-leaf-set pairs)
  (if (null? pairs)
      '()
      (let ((pair (car pairs)))
        (adjoin-set
         (make-leaf (car pair)     ; symbol
                    (cadr pair))  ; frequency
         (make-leaf-set (cdr pairs))))))
```

其中，`successive-merge`的代码如下

```
 (define (successive-merge set)
  (if (= (length set) 1)
      (car set)
      (let ((first (car set))
            (second (cadr set))
            (rest (caddr set)))
        (successive-merge (adjoin-set (make-code-tree first second) rest)))))
```

总结：这一节主要介绍了符号的表示，然后创建了一个简单的求导系统（根据之前介绍的方法），最终介绍了不同方式表示集合对性能的影响，注意我们这里只修改了集合的构造函数和选择函数。