函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

http://blog.sina.com.cn/u/3266490431

第八讲

Racket中可修改的表和序对

●Racket中，pair，list 是不可以修改的。mpair，mlist是可修改的。要使用mpair,mlist，需要在程序开头加上:

```
(require scheme/mpair)
```

例如:

```
(mlist 1 2 3)
=> (mcons 1 (mcons 2 (mcons 3 '())))

(mcons 1 2)
=> (mcons 1 2)
```

Racket中可修改的表和序对

●Racket中，关于可修改列表(mlist)的函数：

mcar	**取mlist的表头元素(或取mpair序对的前部)**
mcdr	**取mlist的表尾(或取mpair序对的后部)**
set-mcar!	**修改mlist的表头(或修改mpair序对的前部)**
set-mcdr!	**修改mlist的表尾(或修改mpair序对的后部)**
mcons	**添加元素到mlist的头部(或构造mpair序对)**
mlist->list	**mlist转换为list (不对列表元素做进一步转换)**
list->mlist	**list转换为mlist (不对列表元素做进一步转换)**

上面函数不能用于不可修改列表(list)。car,cdr 也不能用于可修改列表

mlist?	**判断是否是mlist**
mpair?	**判断是否是mpair**

●Racket中没有 set-car! set-cdr!

Racket中可修改的序对

```
(require scheme/mpair)

(define pr (mcons 1 2))
(mcar pr) => 1
(car pr)  => error
(mcdr pr) => 2
(pair? pr) => #f
(mpair? pr) => #t
(set-mcar! pr 4)
(set-mcdr! pr 5)
pr => (mcons 4 5)
```

mlist彻底转换成list

```
(define (mymlist->list mlst)
  (if (null? mlst)
      '()
      (if (mpair? mlst)
          (let ((first (mcar mlst)))
            (if (or (mpair? first)  (pair? first))
                (cons (mymlist->list first)
                  (mymlist->list (mcdr mlst)))
                (cons first (mymlist->list (mcdr mlst)))))
          (let ((first (car mlst)))
            (if (or (mpair? first)  (pair? first))
                (cons (mymlist->list first)
                  (mymlist->list (cdr mlst)))
                (cons first (mymlist->list (cdr mlst))))))))
```

list彻底转换成mlist

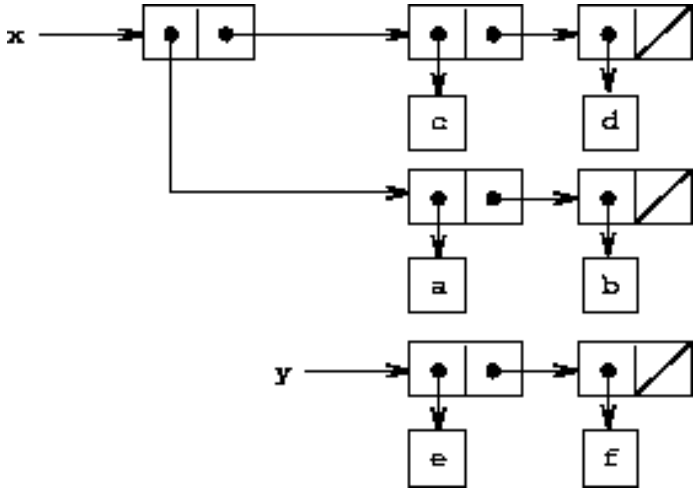
```
(define (mylist->mlist lst)
  (if (null? lst)
      '()
      (if (pair? lst)
          (let ((first (car lst)))
            (if (or (mpair? first)  (pair? first))
                (mcons (mylist->mlist first)
                  (mylist->mlist (cdr lst)))
                (mcons first (mylist->mlist (cdr lst)))))
          (let ((first (mcar lst)))
            (if (or (mpair? first)  (pair? first))
                (mcons (mylist->mlist first)
                  (mylist->mlist (mcdr lst)))
                (mcons first (mylist->mlist (mcdr lst))))))))
```

Racket中可修改的表

```
(define x (mlist (mlist 'a 'b) 'c 'd))
(define y (mlist 'e 'f))
```

或:

```
(define x (mylist->mlist '((a b) c d)))
(define y (mylist->mlist '(e f)))
```

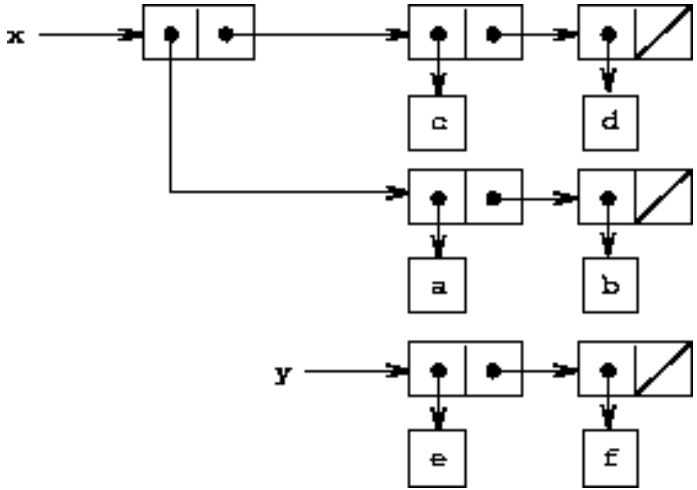


Racket中可修改的表

```
(define x (mlist (mlist 'a 'b) 'c 'd))
(define y (mlist 'e 'f))
```

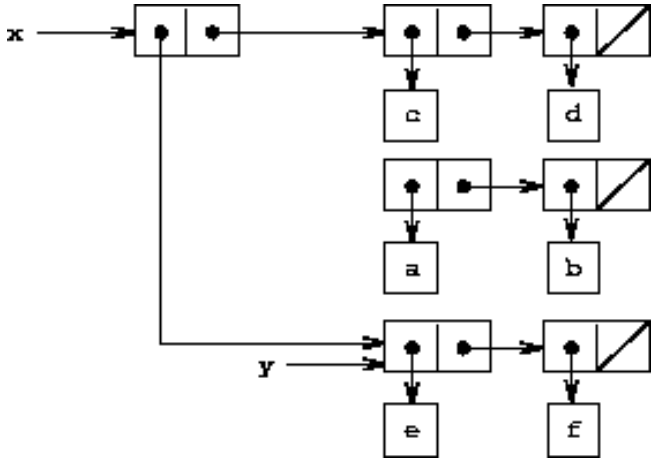
或:

```
(define x (mylist->mlist '((a b) c d)))
(define y (mylist->mlist '(e f)))
```



(set-mcar! x y) →

将x的表头元素改成y

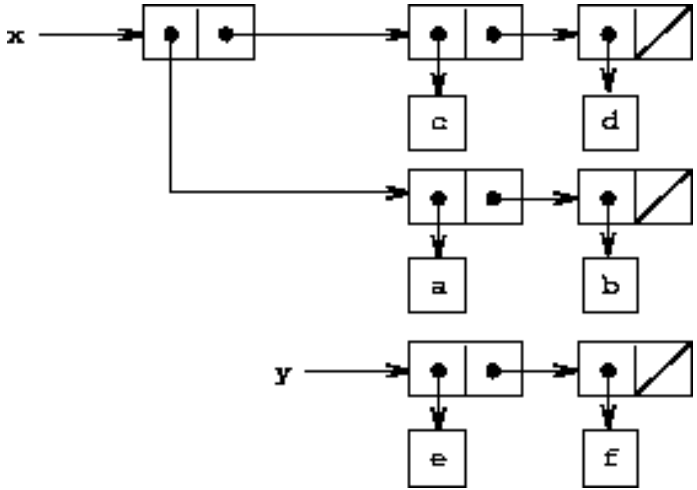


Racket中可修改的表

```
(define x (mlist (mlist 'a 'b) 'c 'd))
(define y (mlist 'e 'f))
```

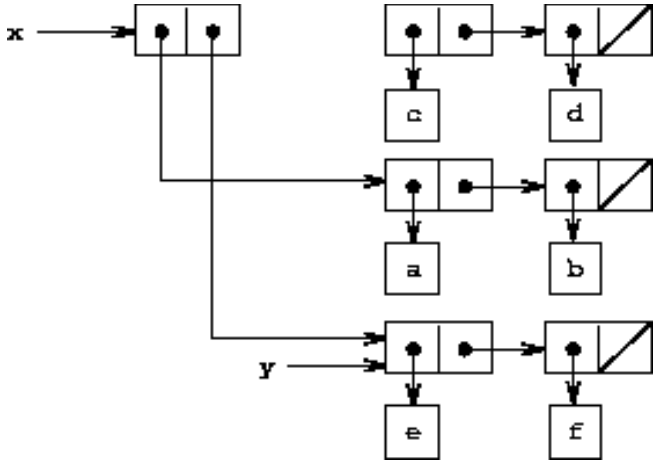
或:

```
(define x (mylist->mlist '((a b) c d)))
(define y (mylist->mlist '(e f)))
```



(set-mcdr! x y) →

将x的表尾改成y

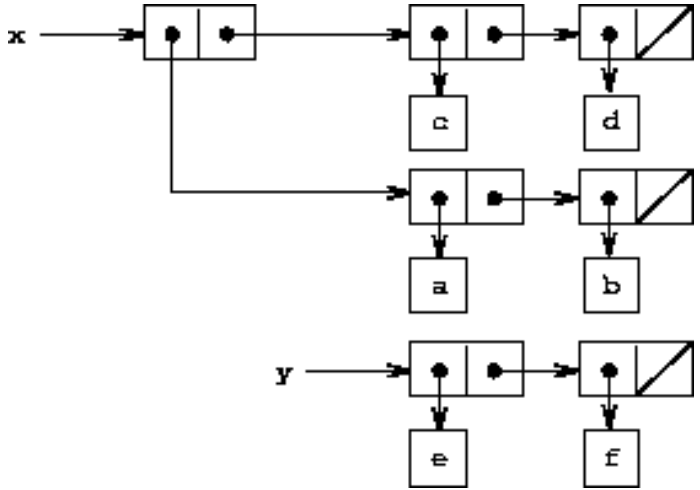


Racket中可修改的表

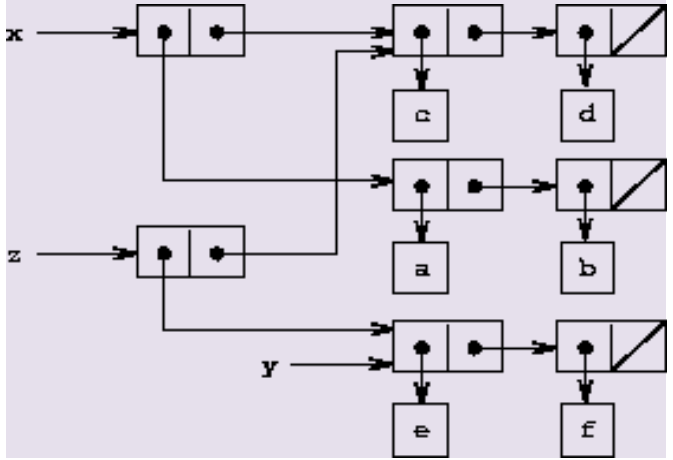
```
(define x (mlist (mlist 'a 'b) 'c 'd))
(define y (mlist 'e 'f))
```

或:

```
(define x (mylist->mlist '((a b) c d)))
(define y (mylist->mlist '(e f)))
```

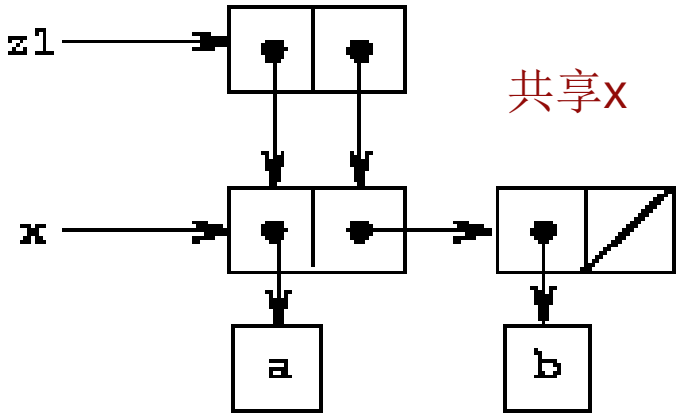


(define z (mcons y (mcdr x)))

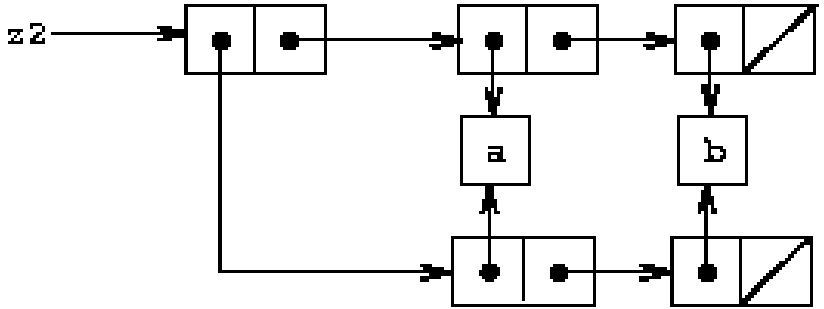


共享

```
(define x (mlist 'a 'b))
(define z1 (mcons x x))
```



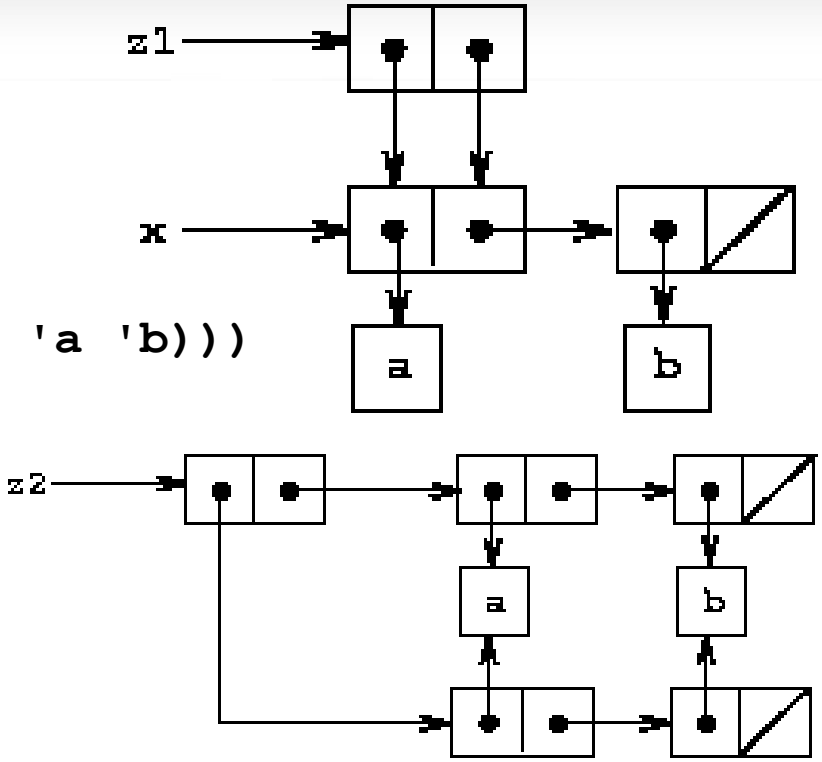

```
(define z2 (cons (list 'a 'b)
                 (list 'a 'b)))
```



z1和z2似乎一样。用 mcar,mcdr ,equal?无法看出结构中是否有共享

共享

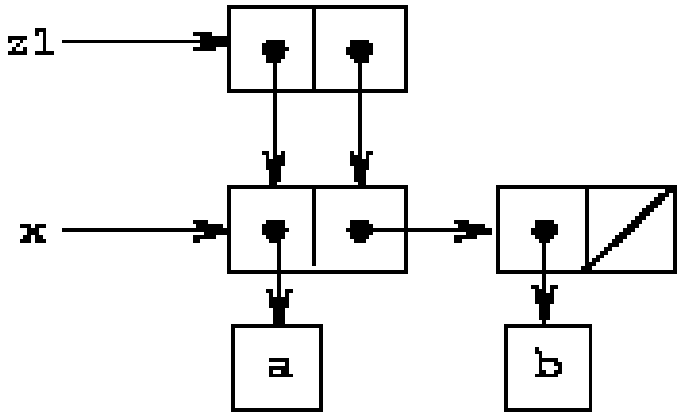
```
(define x (mlist 'a 'b))
(define z1 (mcons x x))
(define z2 (cons (list 'a 'b)(list 'a 'b)))
(set-mcar! (mcar z1) 'wow)
(set-mcar! (mcar z2) 'wow)
(mymlist->list z1)
=> '((wow b) wow b)
(mymlist->list z2)
=> '((wow b) a b)
```



可以看出，z1的mcar和mcdr共享同一个对象，z2不然。

共享

```
(define x (mlist 'a 'b))
(define z1 (mcons x x))
(define s (mcons 'c1 z1))

(mymlist->list s) => '(c1 (a b) a b)
(set-mcar! x 'c2)
(mymlist->list z1)
(mymlist->list s)
```



共享

```
(define x (mlist 'a 'b))
(define z1 (mcons x x))
(define s (mcons 'c1 z1))

(mymlist->list s) => '(c1 (a b) a b)
(set-mcar! x 'c2)
(mymlist->list z1) => '((c2 b) c2 b)
(mymlist->list s)
```

共享

```
(define x (mlist 'a 'b))
(define z1 (mcons x x))
(define s (mcons 'c1 z1))

(mymlist->list s) => '(c1 (a b) a b)
(set-mcar! x 'c2)
(mymlist->list z1) => '((c2 b) c2 b)
(mymlist->list s) => '(c1 (c2 b) c2 b)

;mcons和cons都是只创建一个序对，不会复制列表
```

相等

- eq?和equal?以及 = 的区别

= 只能用于判断数字是否相等

equal? 可以判断两个对象内容是否相等(两个指针指向的地方的内容是否相同)

eq? 判断两个对象是否是同一个(两个指针是否指向同一个对象)

由于符号的唯一性，(eq? 'a 'a) 得真

cons 总建立新序对，(eq? (cons 'a 'b) (cons 'a 'b)) 总是假

相等

```
(define x (list 'a 'b))
(define z1 (cons x x))
(define z2 (cons (list 'a 'b) (list 'a 'b)))
(define z3 (cons x x))
(define z4 z1)

(equal? (car z1) (car z2))
(eq? (car z1) (cdr z1))
(eq? (car z1) (car z2))
(equal? z1 z2)
(eq? z1 z2)
(eq? z1 z3)
(eq? z1 z4)
```

相等

```
(define x (list 'a 'b))
(define z1 (cons x x))
(define z2 (cons (list 'a 'b) (list 'a 'b)))
(define z3 (cons x x))
(define z4 z1)

(equal? (car z1) (car z2))   #t
(eq? (car z1) (cdr z1))
(eq? (car z1) (car z2))
(equal? z1 z2)
(eq? z1 z2)
(eq? z1 z3)
(eq? z1 z4)
```

相等

```
(define x (list 'a 'b))
(define z1 (cons x x))
(define z2 (cons (list 'a 'b) (list 'a 'b)))
(define z3 (cons x x))
(define z4 z1)

(equal? (car z1) (car z2))   #t
(eq? (car z1) (cdr z1))   #t
(eq? (car z1) (car z2))
(equal? z1 z2)
(eq? z1 z2)
(eq? z1 z3)
(eq? z1 z4)
```

相等

```
(define x (list 'a 'b))
(define z1 (cons x x))
(define z2 (cons (list 'a 'b) (list 'a 'b)))
(define z3 (cons x x))
(define z4 z1)

(equal? (car z1) (car z2))   #t
(eq? (car z1) (cdr z1))   #t
(eq? (car z1) (car z2))   #f
(equal? z1 z2)
(eq? z1 z2)
(eq? z1 z3)
(eq? z1 z4)
```

相等

```
(define x (list 'a 'b))
(define z1 (cons x x))
(define z2 (cons (list 'a 'b) (list 'a 'b)))
(define z3 (cons x x))
(define z4 z1)

(equal? (car z1) (car z2))   #t
(eq? (car z1) (cdr z1))   #t
(eq? (car z1) (car z2))   #f
(equal? z1 z2)   #t
(eq? z1 z2)
(eq? z1 z3)
(eq? z1 z4)
```

相等

```
(define x (list 'a 'b))
(define z1 (cons x x))
(define z2 (cons (list 'a 'b) (list 'a 'b)))
(define z3 (cons x x))
(define z4 z1)

(equal? (car z1) (car z2))   #t
(eq? (car z1) (cdr z1))   #t
(eq? (car z1) (car z2))   #f
(equal? z1 z2)   #t
(eq? z1 z2)    #f
(eq? z1 z3)
(eq? z1 z4)
```

相等

```
(define x (list 'a 'b))
(define z1 (cons x x))
(define z2 (cons (list 'a 'b) (list 'a 'b)))
(define z3 (cons x x))
(define z4 z1)

(equal? (car z1) (car z2))   #t
(eq? (car z1) (cdr z1))   #t
(eq? (car z1) (car z2))   #f
(equal? z1 z2)   #t
(eq? z1 z2)    #f
(eq? z1 z3)    #f
(eq? z1 z4)
```

相等

```
(define x (list 'a 'b))
(define z1 (cons x x))
(define z2 (cons (list 'a 'b) (list 'a 'b)))
(define z3 (cons x x))
(define z4 z1)

(equal? (car z1) (car z2))   #t
(eq? (car z1) (cdr z1))   #t
(eq? (car z1) (car z2))   #f
(equal? z1 z2)   #t
(eq? z1 z2)    #f
(eq? z1 z3)    #f
(eq? z1 z4)    #t
```

对 mlist 以上结果依然相同。

可改变序对的闭包实现

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "Undefined operation -- CONS" m))))
  dispatch)
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
(define (set-car! z new-value) ((z 'set-car!) new-value)
  z)
(define (set-cdr! z new-value) ((z 'set-cdr!) new-value)
  z)
```

队列的实现

●构造一个队列，并可对其进行以下操作：

```
(define q (make-queue))
(insert-queue! q 'a)        a
(insert-queue! q 'b)        a b
(delete-queue! q)           b
(insert-queue! q 'c)        b c
(insert-queue! q 'd)        b c d
(delete-queue! q)           c d
```

●基本操作：

创建：(make-queue)

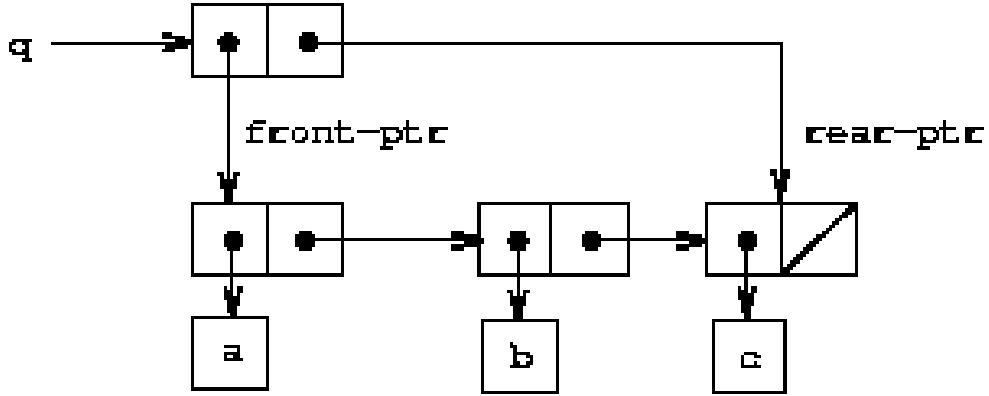
选择：(empty-queue <q>) 和(front-queue <q>)

修改：(insert-queue <q> <item>) 和(delete-queue <q>)

下面程序在racket中使用时，各种表或序对操作须改成相应的mlist或mpair操作

队列的实现

●队列是一个序对，前部为队列头指针，指向一个列表。后部尾为队列尾指针，指向前述列表中的最后一个box。队列头指针为空指针时，队列为空。




```
(define (front-ptr queue) (car queue))
(define (rear-ptr queue) (cdr queue))
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))
(define (empty-queue? queue) (null? (front-ptr queue)))
```

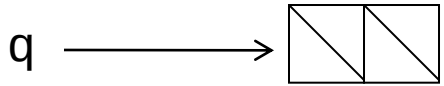
队列的实现

●创建空队列:

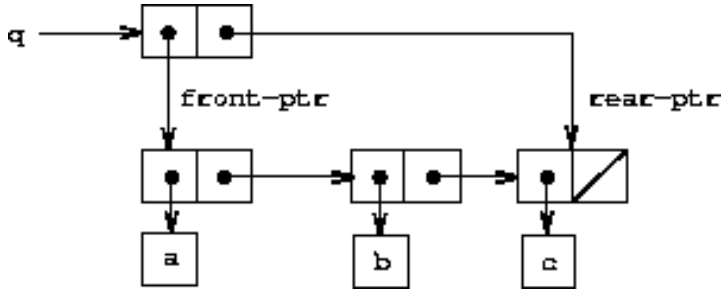
```
(define (make-queue) (cons '() '()))
```

例如:

```
(define q (make-queue))
```



●取队头元素:

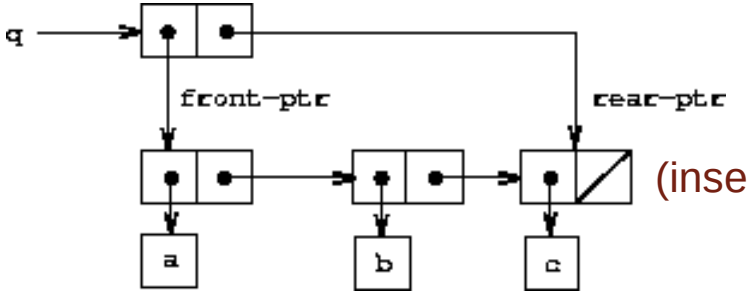



```
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))
```

队列的实现

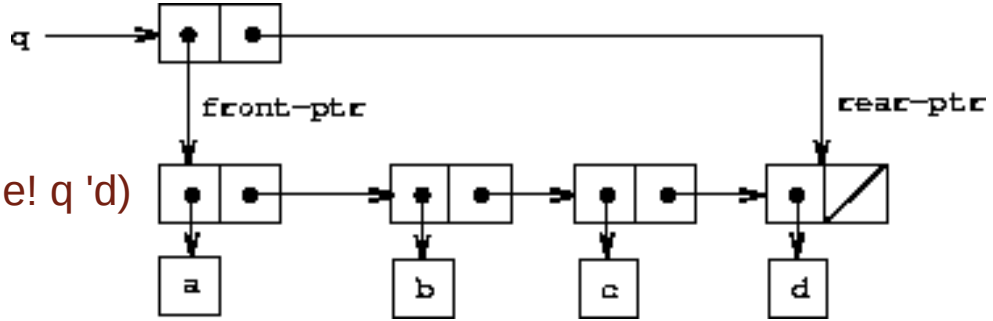
●在队尾插入元素:

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))  ;创建队尾的box
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)  ;(rear-ptr queue)是队尾box
           (set-rear-ptr! queue new-pair)
           queue))))
```



(insert-queue! q 'd)

→

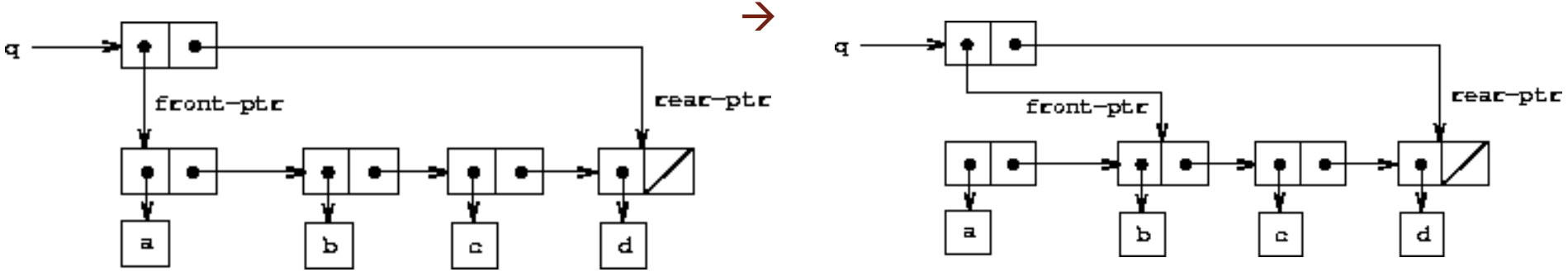


队列的实现

- 在队头删除元素：

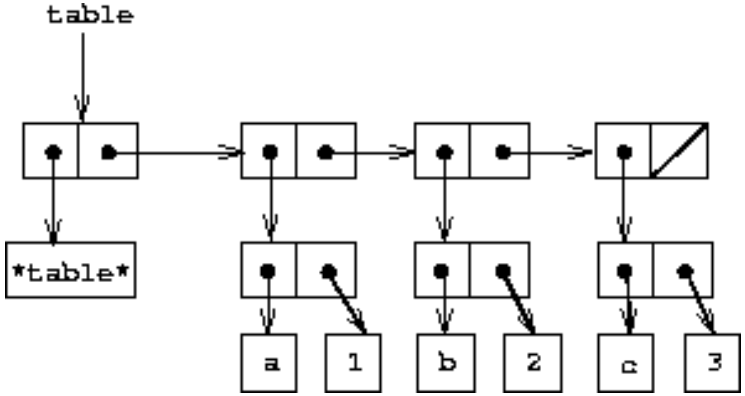
```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))
```

(delete-queue! q)



一维表格的实现

- 要实现一个能按关键字查询的一维表格
  - 表格用列表来实现，列表中除头元素外，每个元素都是一个序对，其car是关键字，cdr是值
  - 列表的头一个元素固定设为'*table*，以便在列表中增删元素



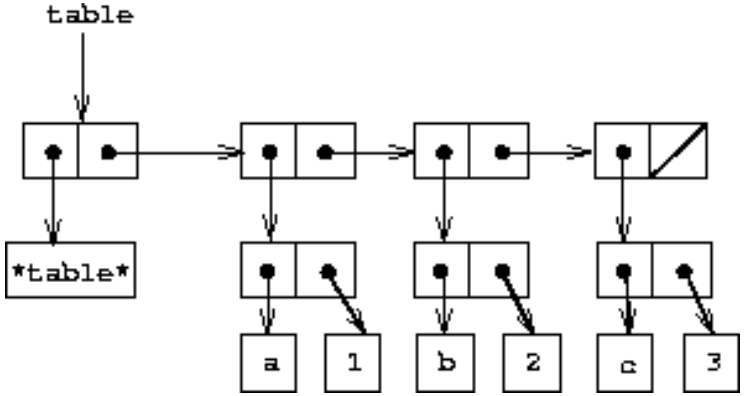
(*table* (a . 1) (b . 2) (c . 3))

下面程序在**racket**中使用时，各种表或序对操作须改成相应的**mlist**或**mpair**操作

一维表格的实现

●表格的查询:

```
(define (lookup key table);查找关键字为key的记录并返回其值。找不到则返回false
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        false)))
(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))
```



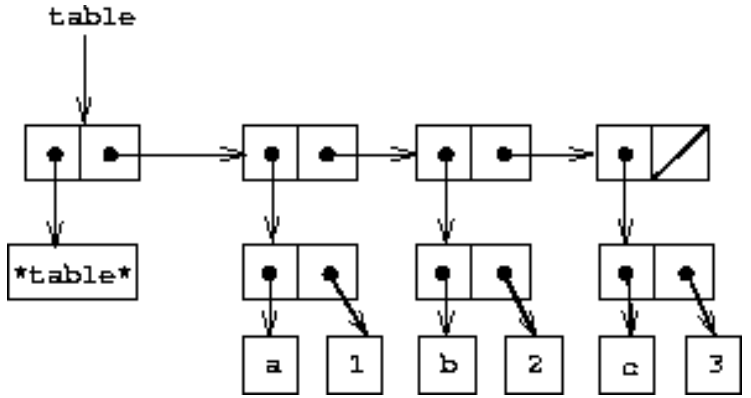
一维表格的实现

●表格的插入:

插入序对(key . value) 若已经存在关键字为key的元素，则修改其值为value

```
(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value) (cdr table)))))
  'ok)
```

被插入的元素放在表格的最前面(*table*之后)



一维表格的实现

- 创建新表:

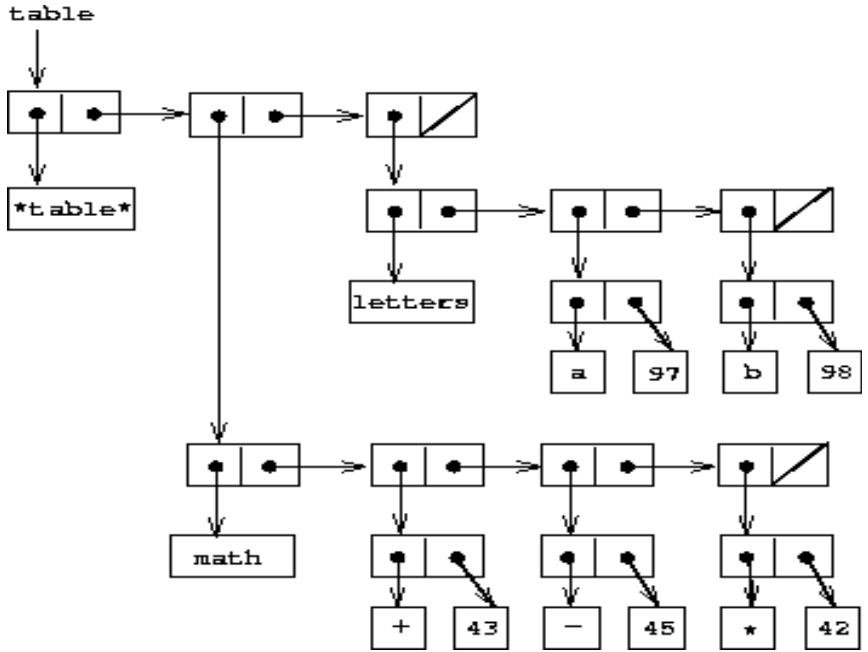
```
(define (make-table)
  (list '*table*))
```

二维表格的实现

●二维表格可以根据两种关键字检索记录。可以看作是一个列表，除头元素外每个元素都是一个序对，其car是一张一维表格，cdr是剩余的二维表格。

('*table* ('math (+ . 43) (- . 45) (* . 42)) ('letters ('a . 97) ('b . 98))

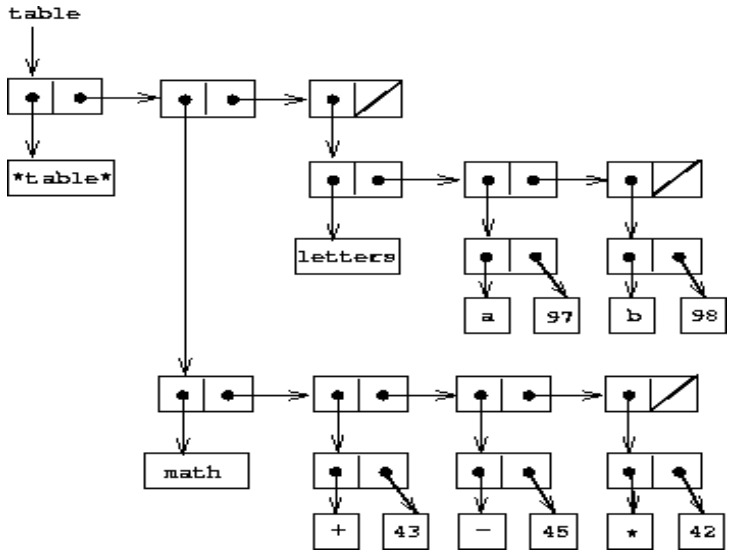
math:
  +: 43
  -: 45
  *: 42
letters:
  a: 97
  b: 98



二维表格的实现

●二维表格的查找（使用两个关键字）：

```
(define (lookup key-1 key-2 table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable ;subtable是个一维表格，表头是 key-1
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (cdr record)
              false))
        false)))
```



二维表格的实现

●二维表格的插入（使用两个关键字）：

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable ;subtable是一维表格
        (let ((record (assoc key-2 (cdr subtable))))
          (if record ;record序对
              (set-cdr! record value)
              (set-cdr! subtable ;插在最前面
                        (cons (cons key-2 value);构造新序对
                              (cdr subtable)))))
        (set-cdr! table
                  (cons (list key-1
                              (cons key-2 value))
                        (cdr table)))))
  'ok)
```

二维表格的实现

- 创建新表(与一维表格相同)：

```
(define (make-table)
  (list '*table*))
```

二维表格的闭包实现

希望能用以下方式使用表格：

```
(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))
```

解决办法：

```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2)
      (let ((subtable (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record (assoc key-2 (cdr subtable))))
              (if record
                  (cdr record)
                  false))
            false)))
```

二维表格的闭包实现

```
(define (insert! key-1 key-2 value)
  (let ((subtable (assoc key-1 (cdr local-table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value) (cdr subtable)))))
        (set-cdr! local-table
                  (cons (list key-1
                              (cons key-2 value))
                        (cdr local-table)))))
  'ok)
(define (dispatch m)
  (cond ((eq? m 'lookup-proc) lookup)
        ((eq? m 'insert-proc!) insert!)
        (else (error "Unknown operation -- TABLE" m))))
dispatch))
```

Racket中表格的实现

要使上面的各种表格程序在Racket中运行，只需要在最前面：

```
(require scheme/mpair)
(define list mlist)
(define cdr mcdr)
(define car mcar)
(define set-car! set-mcar!)
(define set-cdr! set-mcdr!)
(define cons mcons)
(define assoc massoc)   ;使用scheme的基本函数massoc。自定义的assoc可以去掉
```

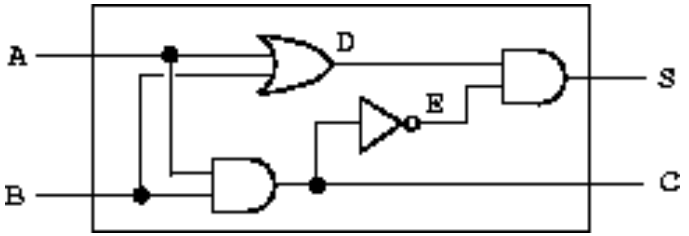
数字电路的模拟

数字电路的模拟 ---概念和原理

●线路 wire

由 make-wire 创建，例如:

```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
```



初始信号值是0

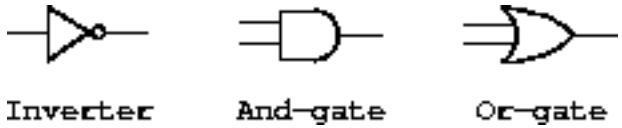
- 线路具有一个内部状态，即该线路上的信号值，可为0或1
- 线路上的信号值被改变后，该线路会激发一系列操作，这些操作的最终作用是改变其他相关线路上的信号值。可以为线路添加它所能激发的操作。因此线路能激发什么样的操作，和线路被用作什么样的电路的输入有关。把线路连进某个电路，就会为该线路添加一些信号值改变时会激发的操作。

数字电路的模拟 ---概念和原理

- 门 gate

➢ 有与门，或门，非门三种。门由线路组成，规定了线路之间的关系。

```
(and-gate a b c)
(inverter c e)
(or-gate a b d)
```



➢ (or-gate a b c)表示有一个或门，输入是a和b，输出是c。(or-gate a b c)的执行结果，是使得将来如果a或b上的信号发生了改变，则经过时间 or-gate-delay后，c上的信号也会发生改变。

```
(define inverter-delay 2) ;非门的延迟是 2
(define and-gate-delay 3) ;and gate 的延迟是3
(define or-gate-delay 5) ;or gate的延迟是 5
```

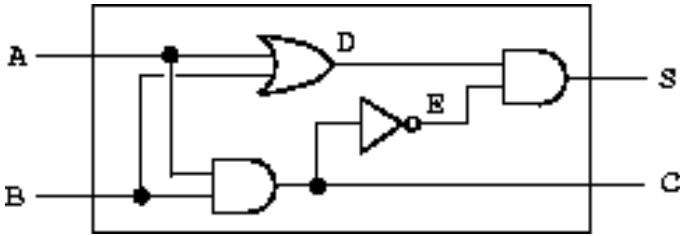
数字电路的模拟 ---概念和原理

- 组合电路

例子：半加器

```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok-half-adder))
```

半加器电路：



A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

规定了 a b c d e s 各条线路之间的关系，即如果a,b信号发生改变，则会引起s和c信号改变。

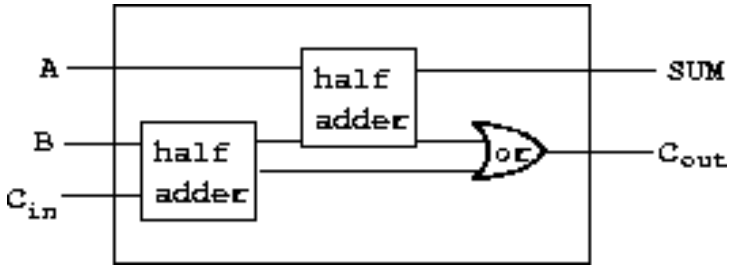
数字电路的模拟 ---概念和原理

- 组合电路

例子：全加器

```
(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)
    'ok))
```

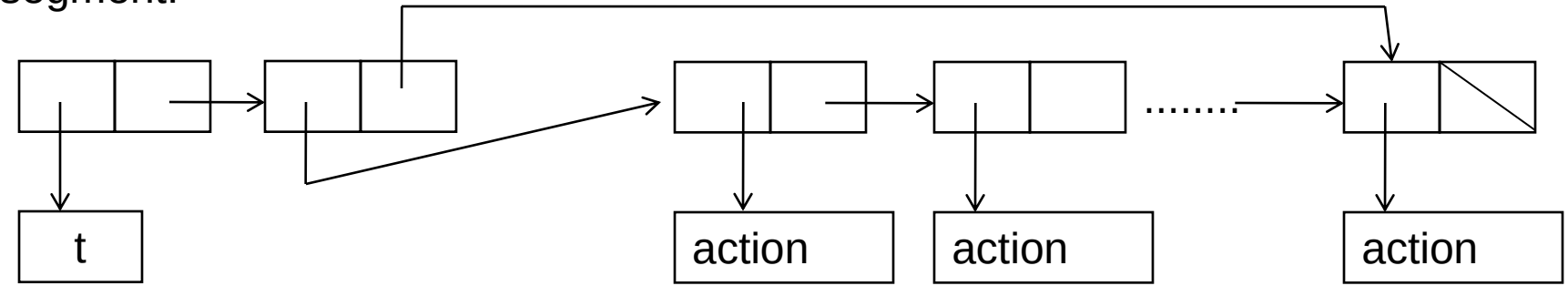
全加器电路：



规定了 a b c-in sum c-out 各条线路之间的关系，即如果a,b, c-in信号发生改变，则会引起sum和c-out信号改变。

数字电路的模拟---概念和原理

- segment:



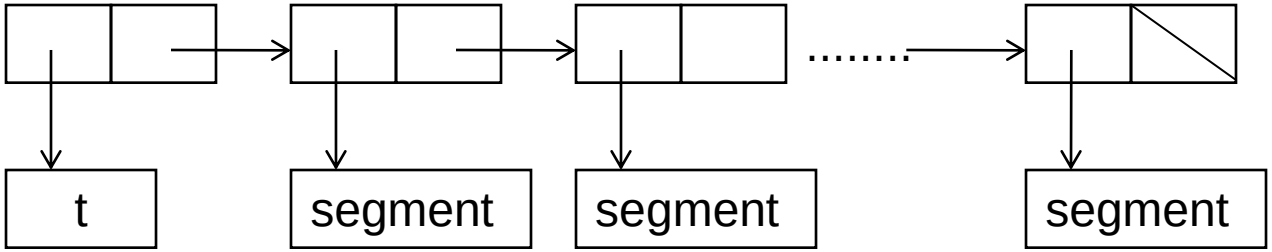
一个segment表示在时间t应该发生的一系列动作(t用一个数代表，动作action存于队列)。每个action都形如:

```
(lambda ()  (set-signal! output new-value)))))
```

即将某个线路output上的信号值设置为 new-value。不同action中的output和new-value都可以不同

数字电路的模拟---概念和原理

- agenda:



t是整个系统的“当前时间”，用一个数表示。
agenda中记录了时间t以后应该依次发生的各种动作。

整个电路程序里只有一个aganda，如下定义：

```
(define (make-agenda) (list 0))   ;开始agenda中没有要发生的事情，起始时间是0
(define the-agenda (make-agenda))
```

数字电路的模拟--- 概念和原理

●**驱动函数propagate**

```
(define (propagate)  ;依次执行 the-agenda里的各个动作，执行完一个动作，就将该动作从the-agenda里删除
  (if (empty-agenda? the-agenda)
      'done-agenda-is-empty
      (let ((first-item (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

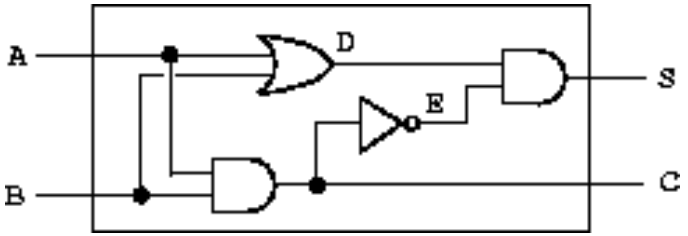
➢一旦有某条线路W上的信号值被改变，线路W上被激发的操作就会往 the-agenda中添加动作（动作就是改变另外一些线路上的信号）。动作的发生时间由W信号改变的时间再加上一个延迟得到。

➢整个程序对电路的模拟，就是修改某些线路上的信号值，然后调用propagate，使得整个电路上某些线路上的信号值发生变化。

数字电路的模拟 ---概念和原理

●例,半加器电路的模拟

```
(define a (make-wire))
(define b (make-wire))
(define s (make-wire))
(define c (make-wire))
```




```
(half-adder a b s c)
(set-signal! a 1) ;会触发a上的add-action操作，对the-agenda添加动作
(propagate) ;依次执行the-agenda上的动作
(get-signal s)  =>1
(get-signal c)  =>0

(set-signal! b 1)
(propagate)
(get-signal s) =>0
(get-signal c) =>1
```

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

数字电路的模拟 --- call-each

```
(define (call-each procedures) ;依次执行线路上的操作列表procedures中的操作，本过程仅在make-wire中被使用
  (if (null? procedures)
      'done-of-call-each-process
      (begin
        ((car procedures)) ;执行表头的操作
        (call-each (cdr procedures)))))
```

此处操作列表中的每个“操作”，形式都是:

```
(after-delay inverter-delay
                    (lambda ()
                      (set-signal! output new-value)))
```

after-delay作用是往the-agenda中添加一个指定触发时间的“动作”，该动作即为改变某条线路上的信号（触发时间是系统当前时间加上延迟时间)。

数字电路的模拟 --- 线路(wire)的实现

```
(define (make-wire)
  (let ((signal-value 0) ;本条线上的信号值(0或1)
        (action-procedures '())) ;本条线上信号发生变化时，需要触发的操作列表
    (define (set-my-signal! new-value)
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 (call-each action-procedures));依次执行操作列表中的操作
           'done-of-set-my-signal))
    (define (accept-action-procedure! proc) ;在操作列表头部加一个操作
      (set! action-procedures (cons proc action-procedures))
      (proc));必须要执行一次proc，因为 wire这条线的初始值就可能影响到其他wire的值，执行一次才能使得其他wire的值可能被更新
    (define (dispatch m)
      (cond ((eq? m 'get-signal) signal-value)
            ((eq? m 'set-signal!) set-my-signal!)
            ((eq? m 'add-action!) accept-action-procedure!)
            (else (error "unknown operation -- wire" m))))
    dispatch))
```

数字电路的模拟 --- 线路(wire)的实现

```
(define (get-signal wire)
  (wire 'get-signal))

(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))

(define (add-action! wire action-procedure)
;往线路上加一个操作，刚加上时和线上信号改变时，该操作会被执行
  ((wire 'add-action!) action-procedure))

;action-procedure 是下面这个形式：
;(after-delay inverter-delay
;                    (lambda ()
;                      (set-signal! output new-value)))
```

数字电路的模拟 --- 与或非门

```
(define (logical-not s) ;逻辑非运算
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "invalid signal" s))))

(define (logical-and a1 a2) ;逻辑与运算
  (cond ((or (= a1 0) (= a2 0)) 0)
        ((and (= a1 1) (= a2 1)) 1)
        (else (error "invalid signal" a1 " " a2))))

(define (logical-or a1 a2) ;逻辑或运算
  (cond ((or (= a1 1) (= a2 1)) 1)
        ((and (= a1 0) (= a2 0)) 0)
        (else (error "invalid signal" a1 " " a2))))
```

数字电路的模拟 --- 与或非门

```
(define (and-gate a1 a2 output) ;与门
  (define (and-action-procedure)
    (let ((new-value
           (logical-and (get-signal a1) (get-signal a2))))
      (after-delay and-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 and-action-procedure);将操作加入操作列表
  (add-action! a2 and-action-procedure)
  'ok-and-gate)
```

各种gate会对其上的输入线路添加操作，这些操作都是调用after-delay往the-agenda中添加修改其他线路信号的动作。在线路上的信号变化时，线路上的操作就会被执行。

数字电路的模拟 --- 与或非门

```
(define (or-gate a1 a2 output)  ;或门
  (define (or-action-procedure)
    (let ((new-value
           (logical-or (get-signal a1) (get-signal a2))))
      (after-delay or-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 or-action-procedure)
  (add-action! a2 or-action-procedure)
  'ok-orgate)
```

数字电路的模拟 --- 与或非门

```
(define (inverter input output) ;非门
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay inverter-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! input invert-input)
  'ok)
```

数字电路的模拟 --- after-delay

```
(define (after-delay delay action)
   (add-to-agenda! (+ delay (current-time the-agenda))
                   action
                   the-agenda))
```

此处的action形式实际上都是:

```
(lambda ()  (set-signal! output new-value)))))
```

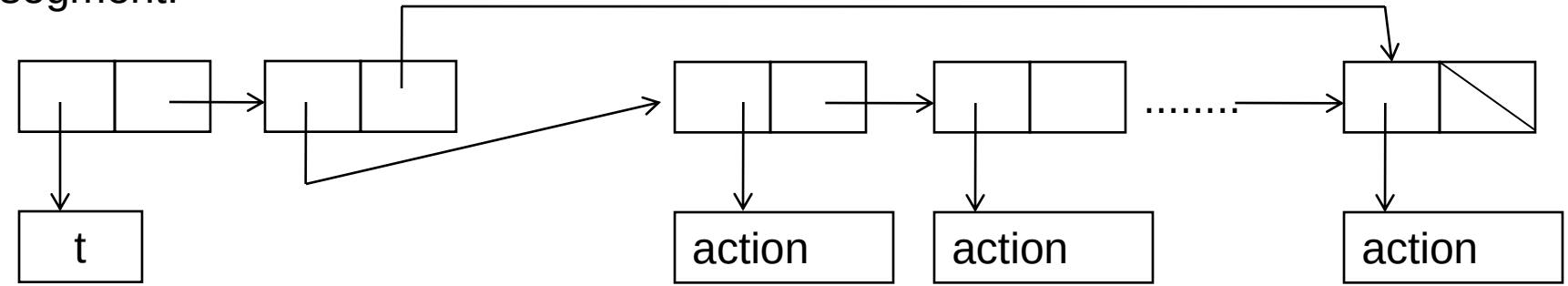
只不过 output和 new-value有所不同

(+ delay (current-time the-agenda) 是action的触发时间

delay就是 or-gate-delay, and-get-delay, inverse-delay 三者之一

数字电路的模拟---概念和原理

- segment:



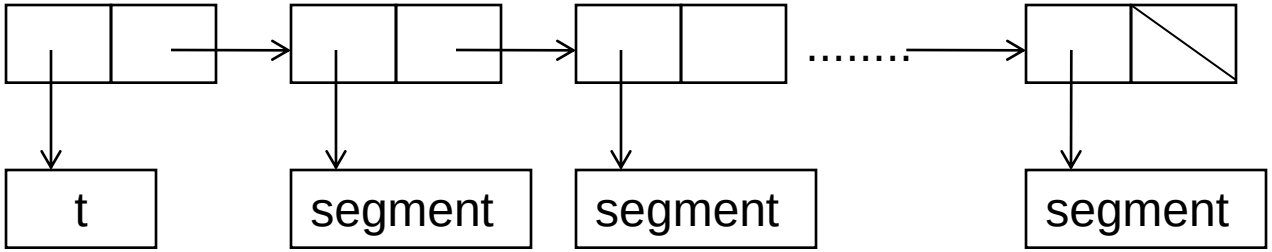
一个segment表示在时间t应该发生的一系列动作(t用一个数代表，动作存于队列)。每个action都形如:

```
(lambda ()  (set-signal! output new-value)))))
```

即将某个线路output上的信号值设置为 new-value。不同action中的output和new-value都可以不同

数字电路的模拟---概念和原理

- agenda:



t是系统的“当前时间”，用一个数表示。
agenda中记录了时间t以后应该依次发生的各种动作。

整个电路程序里只有一个aganda，如下定义：

```
(define (make-agenda) (list 0))   ;开始agenda中没有要发生的事情，起始时间是0
(define the-agenda (make-agenda))
```

数字电路的模拟 --- segment和agenda

```
(define (make-time-segment time queue) (cons time queue))
;只在add-to-agenda中调用。segment由时间和动作队列组成

(define (segment-time s) (car s))
(define (segment-que s) (cdr s))

(define (make-agenda) (list 0))   ;系统的开始时间是0
(define (current-time agenda) (car agenda)) ; car 是当前时间，会不断更新
(define (set-current-time! agenda time)
  (set-car! agenda time))
(define (segments agenda) (cdr agenda)) ;agenda头部是时间，尾部是segment的列表
(define (set-segments! agenda segments)
  (set-cdr! agenda segments))
(define (first-segment agenda) (car (segments agenda)))
(define (rest-segments agenda) (cdr (segments agenda)))
(define (empty-agenda? agenda)
  (null? (segments agenda)))
```

数字电路的模拟 --- segment和agenda

```
;把发生在time时间的action加入到agenda
;每个action都形如:
;(lambda ()  (set-signal! output new-value)))))

(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments);segments是按时间排序的segment的列表
    (or (null? segments)
        (< time (segment-time (car segments)))));比现有的动作都早

  ;创建一个新的segment
  (define (make-new-time-segment time action)
    (let ((q (make-que)))
      (insert-que! q action)
      (make-time-segment time q)))
;(define (make-time-segment time queue) (cons time queue))
```

数字电路的模拟 --- segment和agenda

```
;把action加入到segments
  (define (add-to-segments! segments)
    ;执行的前提是time肯定不会比 (car segments)的时间更早
    (if (= (segment-time (car segments)) time)
    ;如果action的时间time和某个segment的时间一致，则将其加入到这个segment
        (insert-que! (segment-que (car segments))
                     action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
              (set-cdr! segments (cons (make-new-time-segment time
                                                              action)
                                       (cdr segments)))
                                  ;在segments的头部新插入一个segment
              (add-to-segments! rest))))))
```

数字电路的模拟 --- segment和agenda

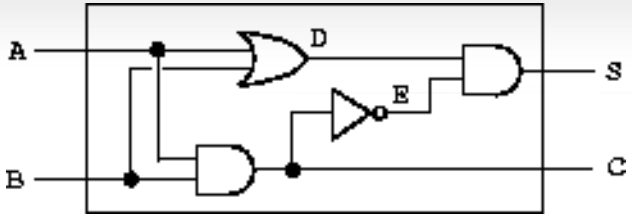
```
(define (remove-first-agenda-item! agenda);删除第一个segment中的第一个动作
  (let ((q (segment-que (first-segment agenda))))
    (delete-que! q)
    (if (empty-que? q)
        (set-segments! agenda (rest-segments agenda))
        'ok-remove-first-agenda-item!)))

(define (first-agenda-item agenda) ;取第一个segment中的第一个动作
  (if (empty-agenda? agenda)
      (error "Agenda is emptyP")
      (let ((first-seg (first-segment agenda)))
        (set-current-time! agenda (segment-time first-seg)) ;set the
time in agenda to the time of first item when accessing the first item
        (front-que (segment-que first-seg)))))

;the-agenda is the global variable
(define the-agenda (make-agenda))
```

数字电路的模拟 --- probe

```
(define (probe name wire) ;在wire添加观测函数
  (add-action! wire
               (lambda ()
                 (newline)
                 (display name)
                 (display " ")
                 (display (current-time the-agenda))
                 (display "  new-value = ")
                 (display (get-signal wire))
                 (newline))))
```

数字电路的模拟

```
(define a (make-wire))
(define b (make-wire))
(define s (make-wire))
(define c (make-wire))
(display "---step1") (newline)
(probe 's s)
(probe 'c c)
(display "---step2") (newline)
(half-adder a b s c)
(display "---step3") (newline)
(set-signal! a 1)
(display "---step4") (newline)
(propagate)
(get-signal s)
(get-signal c)
```

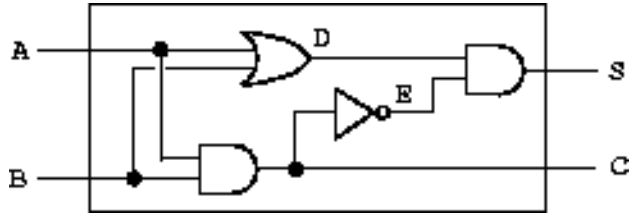



```
inverter-delay 2
and-gate-delay 3
or-gate-delay 5
```

```
---step1

s 0  new-value = 0

c 0  new-value = 0
---step2
'ok-half-adder
---step3
'done-of-call-each-process
---step4

s 8  new-value = 1
'done-agenda-is-empty
1
0
```

数字电路的模拟

```
(display "---step5") (newline)
(set-signal! b 1)
(display "---step6") (newline)
(propagate)
(display "---step7") (newline)
(get-signal s)
(get-signal c)
(display "---step8") (newline)
the-agenda
(display "---step9") (newline)
(define k (make-wire))
(display "---step10") (newline)
(probe 'k k)
```




```
inverter-delay 2
and-gate-delay 3
or-gate-delay 5
```

```
---step5
'done-of-call-each-process
---step6

c 11  new-value = 1

s 16  new-value = 0
'done-agenda-is-empty
---step7
0
1
---step8
(mcons 16 '())
---step9
---step10

k 16  new-value = 0
```

数字电路的模拟

要使上面程序在Racket中能运行，需要：

```
(require scheme/mpair)
(define car mcar)
(define cdr mcdr)
(define list mlist)
(define set-car! set-mcar!)
(define set-cdr! set-mcdr!)
(define cons mcons)
```