3。模块化，对象和状态(2)

这本节课讨论：

- 基于变动数据和状态的模拟
  - 修改表结构
  - 共享和相等
  - 赋值与表结构的变动（修改）
  - 队列和表格
- 数字电路模拟
  - 概念，模拟技术
  - 实现

用变动数据做模拟

- 本节课介绍用具有局部状态的对象做模拟的技术和问题
- 根据前面考虑，建立数据抽象时
  - 应定义一套构造函数和选择函数
  - 用它们封装数据抽象的具体实现
- 下面要基于对象（其状态可能变化）的观点构造系统
  - 为模拟这种系统，需要有能随计算而变化状态的复合数据对象
  - 需要增加一类修改状态的操作

    称为改变函数（mutator）
  - 例如，模拟银行账户时，其表示数据结构应支持余额设置操作：

    (set-balance! <*account*> <*new-value*>)

用变动数据模拟

- 下面还是用序对作为构造复合对象的通用粘合机制
  - **Scheme** 的基本构造机制
  - 请复习一下
- 要用可变对象做模拟，出现了一个新问题：
  - 需要构造状态可以变化的对象

    还是这个对象，但其内容变了

    现在要考虑基于序对构造出来的复杂的有结构对象
  - 修改对象状态，就需要有修改序对内容的操作

    这种操作改变原来的数据结构
- 序对改变操作是 **set-car!** 和 **set-cdr!**，各有两个参数
  - 修改作为其第一个参数的序对的 **car** 或 **cdr** 部分
  - 第二个参数作为修改中使用的值

表结构的变动

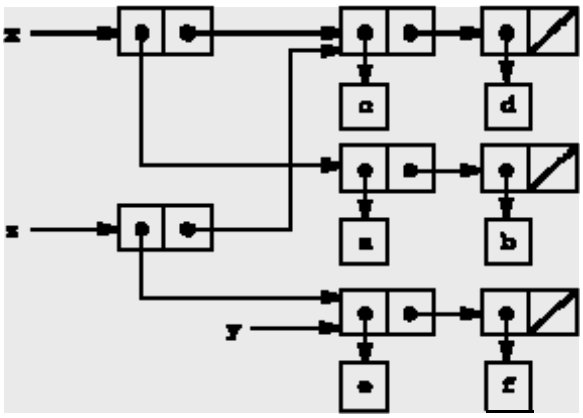
- 例：

  (set-xar! x y) 把 **x** 所指序对的 **car** 修改为 **y**（的值）

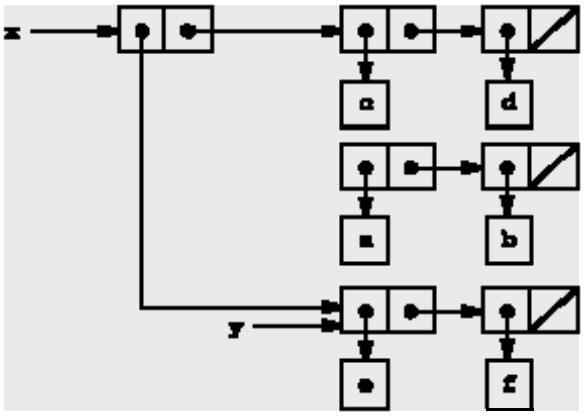
  (set-cdr! x z) 把 **x** 所指序对的 **cdr** 修改为 **z**（的值）

假设 **x** 的值为 **((a b) c d)**，**y** 的值为 **(e f)**

做 **(define z (cons y (cdr x))** 得到

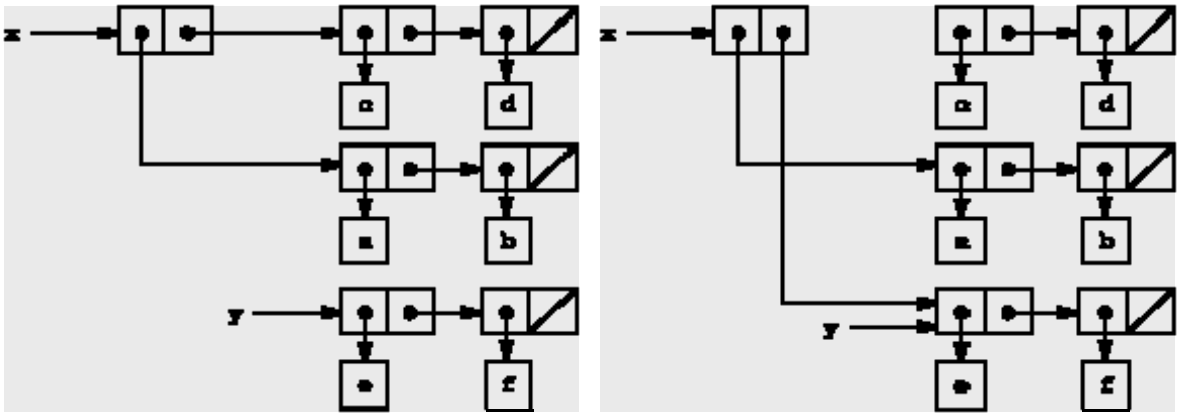


(set-car! x y) 得到：



表结构的变动

在 x 的值为 ((a b) c d)，y 值为 (e f) 的情况下执行 (set-cdr! x y)：



总结和比较：

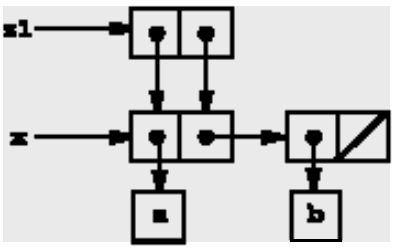
- **set-car!** 和 **set-cdr!** 修改已有的表结构（是破坏性操作）
- **cons** 通过建立新序对的方式构建表结构（没有破坏性）
- 可以用建立新序对的操作 **get-new-pair** 和两个破坏性操作 **set-car!** 和 **set-cdr!** 实现 **cons**

共享和相等

- 赋值引起“同一个”和“变动”问题。如果不同数据对象共享某些序对，问题就可能暴露。例：

  (define x (list 'a 'b))
  (define z1 (cons x x))

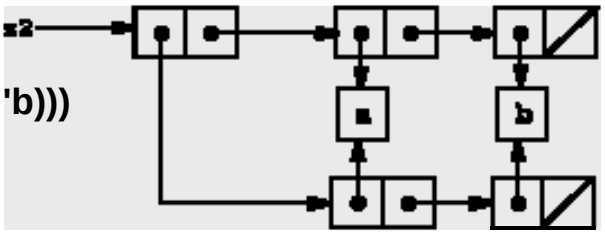
  得到的状态如右图



- 下面表达式产生另一个结构

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

  在 **Scheme** 里，符号具有唯一性



- **z1** 和 **z2** 貌似“一样”。用 **car/cdr/cons** 无法发现其中是否有共享。但是如果修改表结构，就会暴露共享的情况

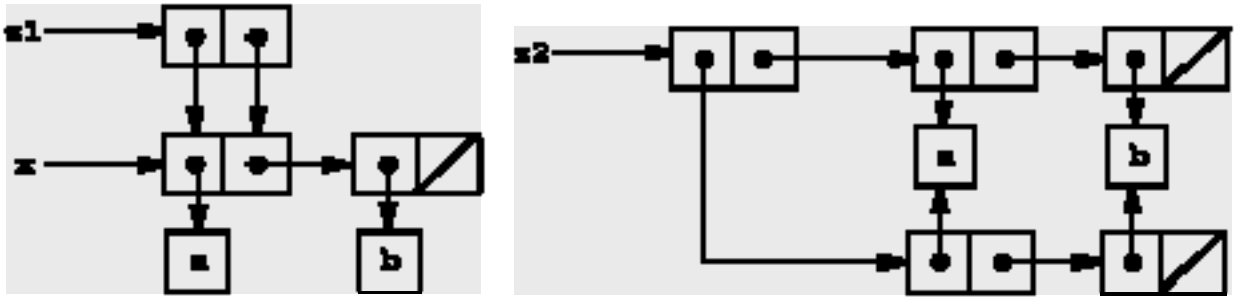
(set-car! (car z1) 'wow)	**(set-car! (car z2) 'wow)**
z1	**z2**
((wow b) wow b)	***((wow b) a b)***

共享和相等

- 前面说 **eq?** 检查两个符号是否相同

  实际上，**eq?** 检查两个表达式的值是否为同一个实体

- 例如，**(eq? x y)** 检查 **x** 和 **y** 的值是否同一个对象（引用同一对象）
  - 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
  - 对前一页的两个情况（下图），**(eq? (car z1) (cdr z1))** 得真，而 **(eq? (car z2) (cdr z2))** 得假



结构共享

- 下面会看到，构造存在共享的结构的功能
  - 可以大大扩充用序对表示的数据结构的范围
  - 特别是：能表示任意复杂的数据对象，它们在存在期间标识不变，但其内部状态却在不断变化
  - 可以用这种结构模拟真实世界中的复杂且不断变化的对象
- 注意：如果存在结构共享
  - 对一部分数据结构的修改可能改变其他数据结构
  - 如果这种改变不是有意的，很可能是程序里的错误
  - 使用改变操作 **set-car!** 和 **set-cdr!** 时要特别小心
    - 必须清楚当时的数据共享情况
    - 否则就可能造成很难确认和排除的程序错误

改变结构就是赋值

- 前面介绍了用过程表示序对的技术：

  下面是一个略微修改的版本

  这里没有用 0 和 1 作为消息，而是用 car 和 cdr

```
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Undefined operation -- CONS" m))))
  dispatch)

(define (car z) (z 'car))

(define (cdr z) (z 'cdr))
```

- 前面说过，可以在这种结构的基础上实现整个 Scheme 系统

改变结构就是赋值

很容易修改这个框架，使它能支持变动操作：

```
(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)
```

理论保证：如果要在一个语言里支持变动，为其引进一个赋值就足够了

set-car!/set-cdr! 都可通过赋值实现

变动数据结构实例：队列

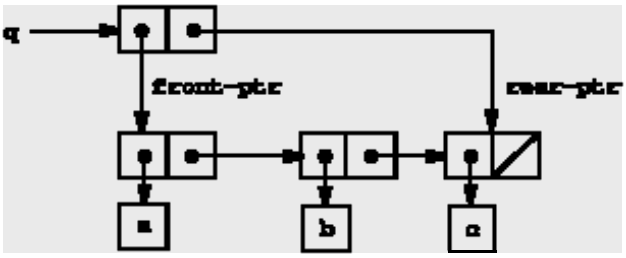
- 利用 **set-car!** 和 **set-cdr!** 能构造出一些基于 **car/cdr/cons** 不能实现的数据结构。其特点是
  - 在执行中，它们一直保持为同一个数据结构（标识不变）
  - 但是其内部的状态可以通过操作改变
- 下面考虑构造一个队列

  一些操作实例：

```
(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
```


队列

- 基本操作（三组）：
  - 创建：**(make-queue)**
  - 选择：**(empty-queue <q>)** 和 **(front-queue <q>)**
  - 改变：**(insert-queue <q> <item>)** 和 **(delete-queue <q>)**
- 下面实现采用的队列表示：



- 先定义几个辅助过程（为了清晰）：

```
(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))
```

队列

- 前端指针空时认为队列空

```
(define (empty-queue? queue)
  (null? (front-ptr queue)))
```

- 空队列是前后端指针均为空的序对：

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

- 选取表头元素就是取出前端指针所指元素的 car:

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

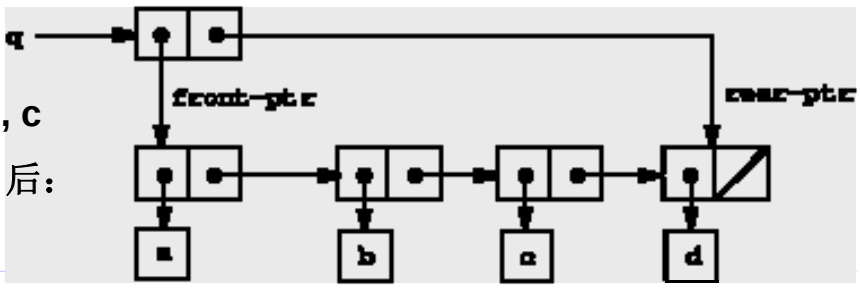

队列

- 向队列加入元素时，需要创建新序对，并将其连接在最后：

```
(define (insert-queue! queue item)
 (let ((new-pair (cons item '())))
   (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)
          (set-rear-ptr! queue new-pair)
          queue))))
```

设队列 q 有元素 a, b, c

(insert-queue! q 'd) 后：

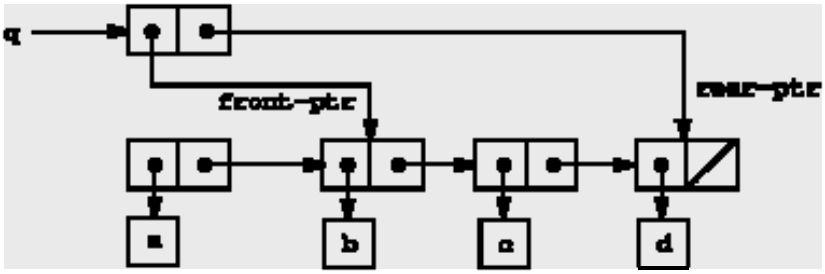


队列

- 删除元素时修改队列前端指针：

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

(delete-queue! q)
之后



Scheme 系统输出过程不理解队列结构，需要定义输出队列的过程

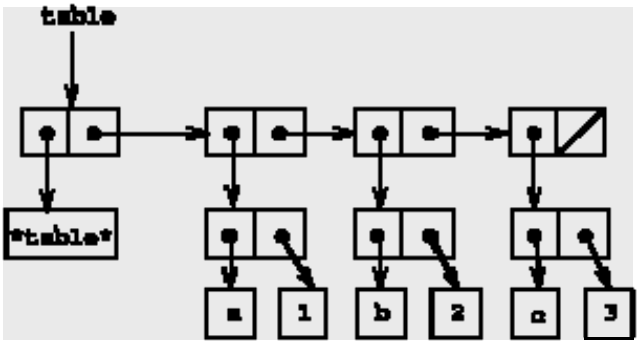
error 函数也不能正确输出有关队列的信息

表格

- 数据导向的编程中用两维表格保存各种操作的信息
- 现在考虑自己实现表格
  - 将表格实现为一种变动数据结构
  - 先考虑一维表格的构造
- 用序对表示关键码/值关联，特殊符号 *table* 作为表格头标志
- 表格：

  a: 1
  b: 2
  c: 3

  的结构如图

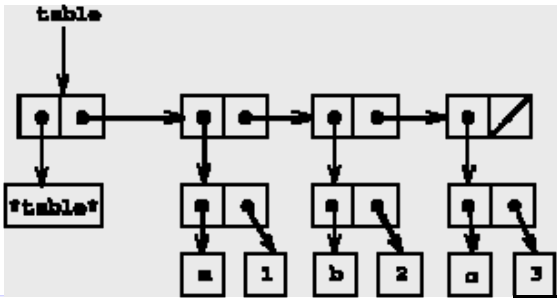


表格

- 表格查找过程 lookup 返回给定关键码所关联的值：

```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record) # 取得所需的值
        false)))

(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records))  (car records))
        (else (assoc key (cdr records)))))
```



表格：一维表格

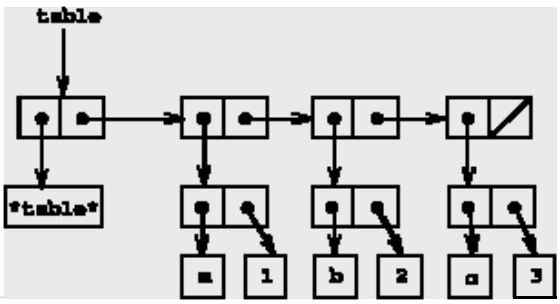
- 要为特定关键码关联一个新值，先找到该关键码所在的序对，然后修改其关联值。找不到时加一个表示该关联的序对

```
(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table          # 新关联项加在最前面
                  (cons (cons key value) (cdr table)))))
  'ok)
```

两种情况都需要修改已有的表格

- 创建新表格就是构造一个空表格

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



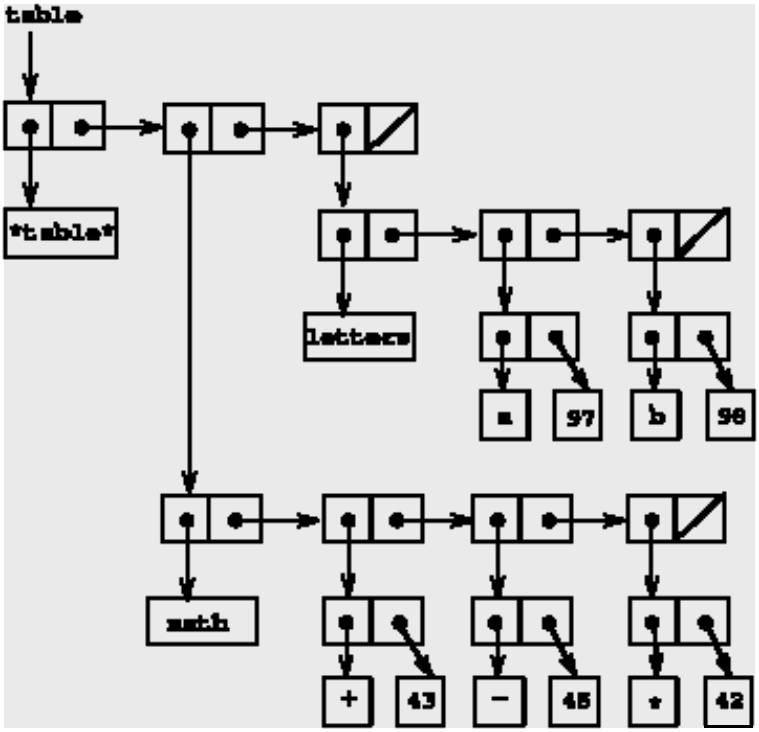
表格：两维表格

考虑两维索引的表格

- 两维表格是以第一个关键码为关键码，以一维表格为关联值的表格
- 右图表示的表格

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

其中有两个子表格



表格：两维表格

- 两维表格查找：用两个关键码逐层查找

```
(define (lookup key-1 key-2 table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record (cdr record)  false))
        false)))
```

表格：两维表格

- 插入关键码也要逐层查找，可能建立新的子表格或表格项：

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr 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! table
                  (cons (list key-1 (cons key-2 value))
                        (cdr table)))))
  'ok)
```


表格：表格生成器

- 表格操作都以一个表格为参数，允许同时有许多表格
- 下面“操作表格生成器”生成存放操作的表格对象（实际上可以存放任何东西），lookup-proc/insert-proc “说明”其用途

  用一个表格数据结构作为所生成对象的局部数据

```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2) ... ... )
    (define (insert! key-1 key-2 value) ... ... 'ok)
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unknown operation -- TABLE" m))))
    dispatch))
```

  内部过程 lookup/insert! 不需要 table 参数，它们都基于作用域规则直接使用局部变量 local-table 关联的表格

表格：表格生成器

- 创建一个操作表格（创建其他表格也一样）：

  (define operation-table (make-table))

  (define get (operation-table 'lookup-proc))

  (define put (operation-table 'insert-proc!))

- 这样实现的表格可以支持前一章讨论的“数据驱动的程序设计”

  回忆一下：在“数据驱动的程序设计” 里，需要用一个表格

  记录各对类型名与操作名关联的实际操作

  上面定义的两维表格正好能用于

  建立操作名和类型信息对到实际过程的的索引

- 这里使用的技术可以由于建立其他类似结构

  下面会看到表格数据结构的很多实际应用

实例：数字电路模拟器

- 本章的第一个大型实例：构造一个数字电路模拟器

  完成一种基于状态的系统模拟

- 复杂的数字电路被应用在许多领域，其正确设计非常重要
  - 一个数字电路包含一些简单元件
  - 通过复杂连接形成复杂的行为
  - 为理解正在设计的电路，应该在实际实现电路之前先做模拟
  - 人们开发了许多电路辅助设计系统，有商品的和开源的系统（该领域称为电子设计自动化，**Electronic Design Automation**）

- 数字电路模拟器代表了一类计算机应用：事件驱动的模拟系统
  - 事件驱动的模拟是一个重要的计算机应用领域，在社会生产生活的许多设计领域发挥着重要作用
  - 通过模拟，可以帮助设计师认识被模拟系统的性质，理解所做设计的特点，认识设计缺陷，为改进设计提供线索

事件驱动的模拟

- 事件驱动的模拟的基本想法：
  - 被模拟的系统通常由一组对象组成，对象处在不断的活动中，整个系统的活动就是这些对象活动的整体效果
  - 系统的活动中将会发生一些事件
    - 由于一些对象遇到的一些状况而引发
    - 事件的发生可能有顺序（例如，由发生的时间确定）
  - 一个事件的发生又可能引发其他事件
    - 事件可能影响一些对象，导致它们的状态变化
    - 对象的状态变化（或达到的状态）可能触发新事件
  - 事件的发生和产生的影响导致了系统状态的不断变化
- 许多系统的活动可以用上面这套方法模拟

  做这种模拟需要用计算机，为此需要为系统建立计算机化的模型

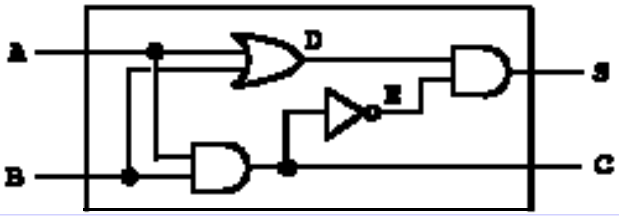
数字电路模拟

- 实际（数字）电子线路由一些基本功能块和它们之间的连线组成
  - 连线在基本功能块之间传递信号

    数字电路传递的信号只有 **0** 和 **1** 两种
  - 一个功能块有若干输入端口和一个输出端口，其功能就是能按某种方式从输入信号计算出输出信号
  - 功能块产生输出信号有一定延迟
- 基本功能块：反门 **(inverter)**，与门 **(and-gate)**，或门 **(or-gate)** 等
  - 每个功能块有确定的输入端和输出端
  - 不同功能块具有不同的输出端信号与输入端信号的关系
  - 产生输出都有若干单位时间的延迟

数字电路模拟器：连线

- 电路的计算模型由一些计算对象构成
  - 模拟实际电路里的各种基本电路元件，模拟其功能/延迟等
  - 需要某种机制模拟连线，模拟对象之间的数字信号传递
- 连接基本功能块，可得到更复杂的（组合）功能块。如半加器，有输入 A 和 B，输出 S（和）和 C（进位）
  - 电子线路见下
  - A、B 之一为 1 时 S 为 1；A 和 B 均为 1 时 C 为 1
  - 由于延迟，得到输出的时间可能不同
- 模拟器还应能模拟电路的组合（和抽象），组合功能块可以继续组合



连线

- 现在考虑构造一个模拟数字电路的程序。其中
  - 连线用计算对象表示，它们能保持信号
  - 功能模块用过程模拟，它们产生正确的信号关系
- 在构造模拟数字电路的系统里
  - 连线是一种数据抽象
  - 连线的构造操作是 make-wire
- 例如构造 6 条连线：

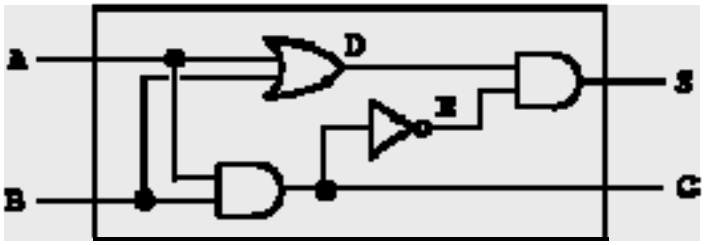
```
(define a (make-wire))   (define b (make-wire))
(define c (make-wire))   (define d (make-wire))
(define e (make-wire))   (define s (make-wire))
```

- 各种基本门电路也是数据抽象
  - 假定已经定义了相应的构造函数（后面有实际定义）

数字电路模拟器：组合和抽象

- 连接几个反门、与门和或门，可以做出一个半加器：

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



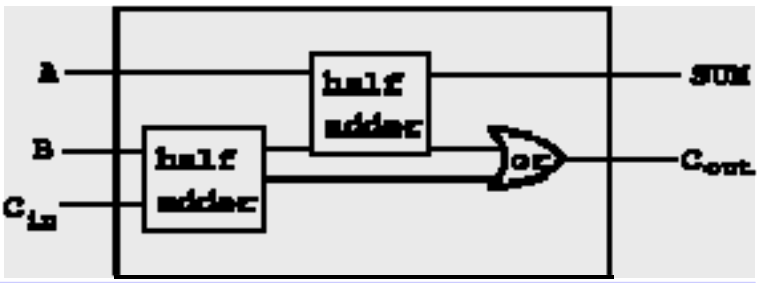
- 更好的是把半加器的构造操作定义为过程，它取 4 个连线参数：

```
(define (half-adder a b s c) ; 建立过程抽象（实现功能块抽象）
  (let ((d (make-wire)) (e (make-wire))) ; 内部连线
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```


数字电路模拟器

- 用两个半加器和一个或门可以构造出一个全加器
- 全加器过程的定义：

```
(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))
```



数字电路模拟器：语言结构

- 以构造好的功能块作为部件，可以继续构造更复杂的电路
- 这些实际上建立了一种特殊语言，支持构造结构任意复杂的数字电路。语言的元素：
  - 基本功能块是基本元素。实现特定效果，使连线的信号能按某种特定方式影响其他连线
  - 用连线连接不同功能块，是语言的组合机制
  - 将复杂连接方式定义为过程，是这里的抽象机制
- 连线基本操作
  - **(get-signal <*wire*>)** 返回 **<*wire*>** 上的当前信号值
  - **(set-signal! <*wire*> <*new value*>)** 将 **<wire>** 的信号值设为新值
  - **(add-action! <*wire*> <*procedure of no arguments*>)** 要求在 **<wire>** 的信号值改变时执行指定过程（过程注册）
  - **after-delay** 要求在给定时延之后执行指定过程（两个参数）

数字电路模拟器：基本功能块

- 基本功能块基于这些操作定义
- 反门在特定的时间延迟后将输入的逆送给输出

```
(define (inverter input output)
    (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)

(define (logical-not s)
    (cond ((= s 0) 1)
          ((= s 1) 0)
          (else (error "Invalid signal" s))))
```

在 **input** 连线对象上登记一个无参过程 **inverter-input**

数字电路模拟器：基本功能块

- 与门有两个输入，两个输入得到信号时都要执行某种动作
- 过程 logical-and 与 logical-not 类似，只是有两个参数

```
(define (and-gate a1 a2 output)
  (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)
  (add-action! a2 and-action-procedure)
  'ok)

(define (logical-and a1 a2) (cond ... ...))
```

- 常见的或门、异或门等都可以用类似技术定义
- inverter-delay 和 and-gate-delay 等是电路模拟中的常量，需要事先另外定义

数字电路模拟器：连线

- 连线是计算对象，有局部的信号值变量 signal-value 和记录一组过程的变量 action-procedures，连线信号值改变时执行这些过程：

```
(define (make-wire)
  (let ((signal-value 0) (action-procedures '())) ；初始值
    (define (set-my-signal! new-value)
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 (call-each action-procedures))
          'done))
    (define (accept-action-procedure! proc)
      (set! action-procedures (cons proc action-procedures))
      (proc))
    (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))
```

辅助过程 call-each 逐个调用过程表里的各个过程（都是无参过程）

立刻执行无参过程 proc一次

数字电路模拟器：连线

```
(define (call-each procedures)
    (if (null? procedures)
        'done
        (begin
          ((car procedures))
          (call-each (cdr procedures)))))
```

- 使用连线的几个过程：

```
(define (get-signal wire)
  (wire 'get-signal))
(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))
(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))
```

- 连线是典型的变动对象，保存可能变化的信号，用带局部状态的过程模拟。**make-wire** 过程构造出一个新连线对象
- 连线状态被连接其上的所有功能块共享，一个部件的活动通过交互影响与之相连的连线的状态，进而影响连接在这条线上的其他部件

数字电路模拟器：待处理表

- 考虑 **after-delay** 操作的实现

  要求在一段时间之后执行某个操作

  需要维护的信息：何时执行，执行什么动作

  想法：维护一个表，称为待处理表，记录将来需要处理的事项

- 待处理表是一个数据抽象，提供如下操作：

  (make-agenda) 返回新建的空待处理表

  (empty-agenda? <*agenda*>) 判断待处理表是否为空

  (first-agenda-item <*agenda*>) 返回待处理表中第一个项

  (remove-first-agenda-item! <*agenda*>) 删除待处理表里的第一项

  (add-to-agenda! <*time*> <*action*> <*agenda*>) 向待处理表中加入一项，其意义是要求在给定时间运行的过程

  (current-time <*agenda*>) 返回当前时间

待处理表

- 待处理表用 **the-agenda** 表示
- **after-delay** 向其中加入一个新项

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

- 待处理表的具体实现后面考虑

数字电路模拟器：模拟

- 模拟驱动过程 **propagate**：它顺序处理待处理表中的项
  - 表项记录的是要求在某特定时间做某个动作
  - 处理表中的项就是在“特定时间”完成“事件”要求的动作
  - 处理待处理表项的过程中可能生成新的事件请求，导致给待处理表加入新项（要求在将来的特定时间发生的事件）
  - 只要待处理表不空（还有事件等待发生），模拟就继续进行
- 过程定义

```
(define (propagate) ; 迭代式地取出最早的事件并执行之
(if (empty-agenda? the-agenda)
    'done
    (let ((first-item (first-agenda-item the-agenda)))
      (first-item)
      (remove-first-agenda-item! the-agenda)
      (propagate))))
```

监视器

- 为看到所实现的数字电路模拟系统的运行情况，需要实现一个为人服务的观察接口过程

  这种过程称为“监视器”（monitor）

- “监视器”过程的定义：

```
(define (probe name wire)
  (add-action! wire ; 同样是通过注册的方式登记
               (lambda ()
                 (newline)
                 (display name)
                 (display ", Time = ")
                 (display (current-time the-agenda))
                 (display "  New-value = ")
                 (display (get-signal wire)))))
```

  把它连（注册）在某一根连线 wire 上，一旦连线值改变，就会调用这个过程产生相应输出

数字电路模拟器：模拟实例

- 模拟实例：模拟一个半加器的运行情况
- 初始化：建立待处理表，定义一些常量

```
(define the-agenda (make-agenda))
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)
```

- 定义 4 条线路，在其中两条各安装一个监视器：

```
(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))
(probe 'sum sum)
sum 0  New-value = 0 ; 立即执行一次的效果
(probe 'carry carry)
carry 0  New-value = 0
```

模拟实例

- 把线路连接到半加器上：

```
(half-adder input-1 input-2 sum carry)
ok
```

- 将 input-1 的信号置为 1，而后运行这个模拟：

```
(set-signal! input-1 1)
done
(propagate)
sum 8  New-value = 1
done
```

- 这时将 input-2 上的信号置 1 并继续模拟：

```
(set-signal! input-2 1)
done
(propagate)
carry 11  New-value = 1
sum 16  New-value = 0
done
```

时刻 11 时 carry 变为 1

时刻 16 时 sum 变为 1

数字电路模拟器：待处理表的实现

- 待处理表的功能与队列类似，其中记录要运行的过程。元素是时间段，包含一个时间值和一个队列，队列里是在该时间要执行的过程：

```
(define (make-time-segment time queue)  (cons time queue))
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))
```

- 待处理表是时间段的一维表格，时间段按时间排序

  为了方便，在表头记录当前时间（初始为 0）

```
(define (make-agenda) (list 0))
(define (current-time agenda) (car agenda))
(define (set-current-time! agenda time) (set-car! agenda time))
(define (segments agenda) (cdr agenda))
(define (set-segments! agenda segments)
  (set-cdr! agenda segments))
(define (first-segment agenda) (car (segments agenda)))
(define (rest-segments agenda) (cdr (segments agenda)))
```

数字电路模拟器：待处理表的实现

- 将动作加入待处理表的过程：

```
(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments) (< time (segment-time (car segments)))))
  (define (make-new-time-segment time action)
    (let ((q (make-queue)))
      (insert-queue! q action)
      (make-time-segment time q)))
  (define (add-to-segments! segments)
    (if (= (segment-time (car segments)) time)
        (insert-queue! (segment-queue (car segments)) action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
              (set-cdr! segments
                (cons (make-new-time-segment time action) (cdr segments)))
              (add-to-segments! rest)))))
  (let ((segments (segments agenda)))
    (if (belongs-before? segments)
        (set-segments! agenda
          (cons (make-new-time-segment time action) segments))
        (add-to-segments! segments))))
```


数字电路模拟器：待处理表的实现

- 现在考虑待处理表数据抽象的实现，其操作主要是加入和删除
- 需要从待处理表删除第一项时，应该删除第一个时间段队列里的第一个过程。如果删除后队列为空，就删除这个时间段（队列）：

```
(define (remove-first-agenda-item! agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (delete-queue! q)
    (if (empty-queue? q)
        (set-segments! agenda (rest-segments agenda)))))
```

- 待处理表的第一项就是其第一个时间段队列里的第一个过程。提取项时应该更新待处理表的时间：

```
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "Agenda is empty -- FIRST-AGENDA-ITEM")
      (let ((first-seg (first-segment agenda)))
        (set-current-time! agenda (segment-time first-seg))
        (front-queue (segment-queue first-seg)))))
```

数字电路模拟器

- 数字电路模拟系统的开发完成。现在总结一下
- 系统类型：离散事件模拟系统
  - 事件，事件的发生，导致某些动作并可能产生新事件
  - 技术：用一个待处理事件表，维护和使用这个表
  - 这是很有一般性的问题和技术
- 具体系统：模拟电子线路的活动
  - 一组基本功能块，表示基本电路，用带有状态的对象模拟
  - 实际电路是更简单的电路的组合（闭包性质）
  - 定义连线对象，用于连接简单电路构成更复杂的电路。在模拟中，连线负责在部件之间传递信号
- 注意：电路模拟器接口形成了一个描述电路及其组合的语言：基本功能块作为基本元素，用连线组合，通过定义高级过程的方式建立抽象

总结

- 表结构的变动

  变动操作 **set-car!** 和 **set-cdr!**

  从理论上，为支持基于状态的模拟（编程），只需要一个赋值操作

  在 **Scheme** 语言里是 **set!** 特殊形式
- 基于状态改变建立的数据结构
  - 队列
  - 表格，一维和两维
- 数字电路模拟，是一个大实例
  - 构造一个描述数字电路的语言：基本块，作为组合机制的连线，组合电路，组合电路的抽象（定义过程）
  - 动作的注册技术
  - 通过待处理表，实现离散事件模拟系统的运行