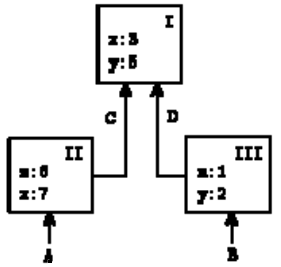
3。模块化，对象和状态(2)

这本节课讨论：

- 求值的环境模型
  - 环境模型中的求值规则
  - 过程应用
  - 局部状态
  - 内部定义
- 基于状态的模拟
  - 两个简单实例：队列和表格
  - 数字电路模拟

求值的环境模型

- 代换模型：复合过程作用于一组参数时，先求值实参，而后用这些值代换过程体里的形参，再求值代换后的过程体
- 有了赋值后，代换模型就失效了。因为变量不再是代表值的简单名字，而表示某种“存储位置”，其中保存的值可随计算进展而改变
- 要处理赋值，求值模型必须反映存储的概念。下面的新模型称为环境模型。有几个概念：
- 环境：框架（frame）的链接序列
  - 框架是可空的表格，每项表示一个变量的约束。在一个框架里每个变量至多有一个约束
  - 每个框架有一个指向其外围框架的指针，全局框架位于最上层，它没有外围框架
- 一个变量在一个环境里的值，就是它在该环境里的第一个有约束的框架里的那个约束值



例：x在环境A中的值，在环境 B中的值

环境模型和求值规则

- 环境确定表达式求值的上下文。没有环境，表达式求值就没有意义

  (+ 1 1) 的求值也需要上下文为 + 提供意义
- 为描述解释器的意义，我们假定有一个全局环境，其中只有一个全局框架，框架中包含着所有基本过程名的意义约束
- 在新求值模型里，组合表达式的基本求值规则仍是：
  - 求出组合式的各子表达式的值
  - 将运算符表达式的值作用于运算对象表达式的值
- 一个过程对象是一个对 (*c*, *e*)，其中 *c* 是过程代码，*e* 是环境指针。求值 lambda 表达式得到的是过程对象：
  - 其代码就是 lambda 表达式的体
  - 其环境指针指向求值该 lambda 表达式时的环境
- 下面用例子说明有关情况，包括：环境在求值中的作用，求值过程中框架的创建，过程对象的创建等

环境模型下的求值

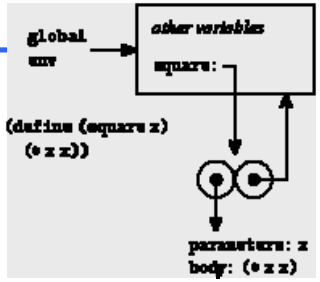
- 过程对象的建立和约束。在全局环境中求值

  (define (square x) (* x x))

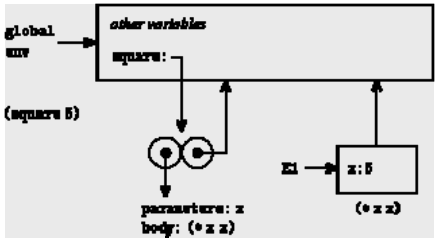
  实际上就是求值：

  (define square (lambda (x) (* x x)))

  图：在全局环境里增加了 square 的约束，它约束于新建的过程对象



- 过程应用。在全局环境里求值表达式 (square 5)：
  - 过程应用表达式创建新环境 E1，其中 x （形参）约束到 5（实参的值）
  - 在环境 E1 中求值过程体 (* x x) 得到结果 25



环境模型下的求值规则

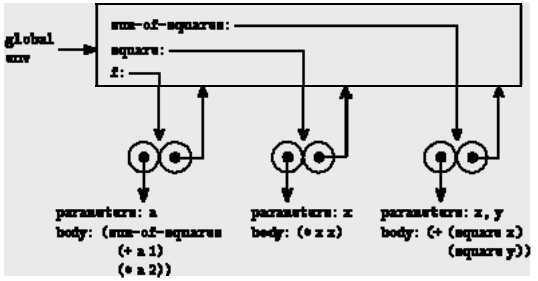
- 环境模型下的两条基本求值规则：
  - 一个过程对象应用于一组实参的过程：先构造一个新框架，该框架以过程对象的框架作为外围框架，框架里是过程的形参与对应实参值的约束。而后在这个新环境中求值过程体
  - 在环境 E 里求值一个 lambda 表达式：建立一个过程对象，其代码是该 lambda 表达式的体，其环境指针指向 E
- 用 define 定义一个符号：在当前框架里建立一个约束，将被定义符号约束到给定值（如果当前框架已有这个符号，则改变其约束。【另请参考书上的注释】）
- (set! <*变量*> <*value*>) 的作用：
  - 在当前环境中找到<*变量*>的约束（可能不在当前框架里）
  - 将该变量的约束值修改为由 <*value*> 计算出的值
  - 如果环境中没有<*变量*>的约束，则报告错误
- 新求值规则比代换模型复杂很多。但它是 Scheme 解释器工作方式的真正模型，可以基于这个模型实现 Scheme 解释器（第4章）

环境模型：简单过程的应用

假设有定义

```
(define (square x) (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(define (f a)
  (sum-of-squares (+ a 1)
                  (* a 2)))
```

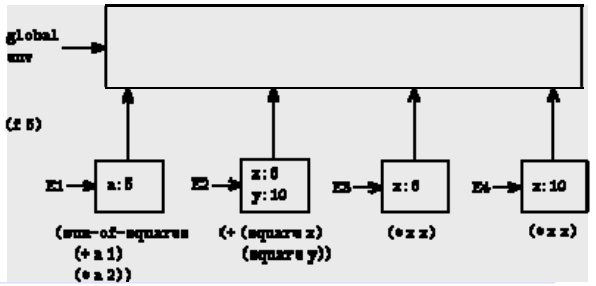
三个定义建立起的环境见图



对 (f 5) 的求值

求值时新建一个环境，其中有一个新约束

- 每个调用创建一个新框架，同一函数的不同调用的框架相互无关
- 这里没有特别关注返回值的传递问题

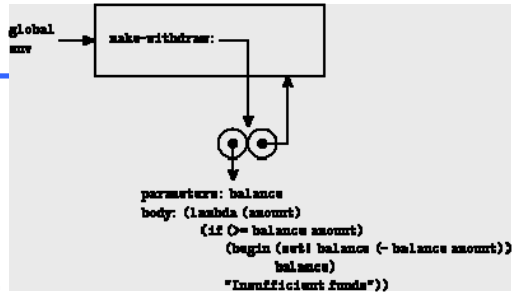


框架和局部状态

<u>有局部状态的对象在计算中的情况</u>

提款处理器代码：

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance
                     (- balance amount))
               balance)
        "Insufficient funds")))
```

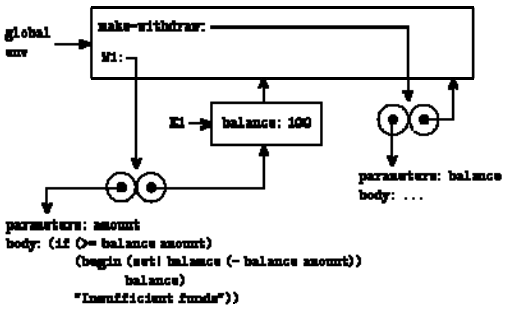


调用

```
(define W1 (make-withdraw 100))
```

新建环境 E1，在其中求值过程体

求值过程里 lambda 表达式将建立新过程对象（左），其环境指针指向 E1，W1 约束于这个过程对象



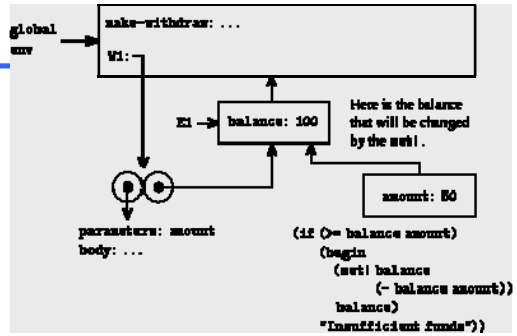
框架和局部状态

考虑过程调用：

```
(W1 50)
```

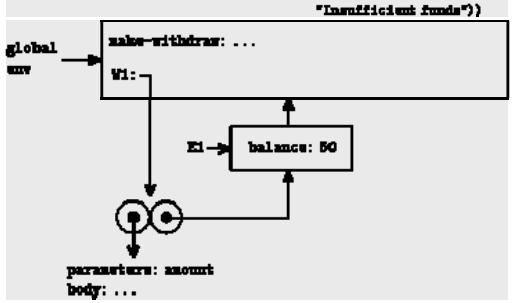
调用建立起一个新环境（右）并在其中求值

从环境的不同框架中，可以找到各变量的值



过程 set! 表达式的求值改变环境中 balance 的约束，使其值变为 50

- 再调用 W1 将建立新框架，与上面建立的框架无关，但其外围框架仍是 E1
- 过程求值中将再次找到包含 balance 的框架 E1 并修改 balance 的约束值

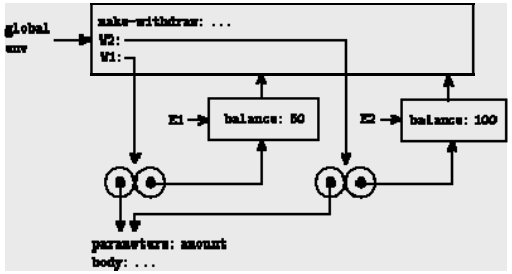


框架和局部状态

- 建立另一个提款处理器

(define W2 (make-withdraw 100))

- 新提款处理器 W2 的局部状态与 W1 的局部状态无关



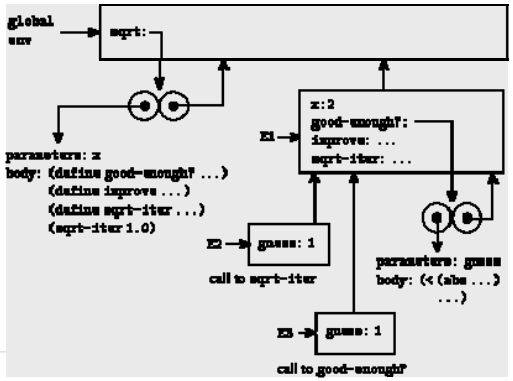
- 两个提款处理器是两个过程对象，各自独立变化
- 两个提款处理器（过程对象）的代码相同。它们是共享同一份代码还是各有一份代码，是系统实现的细节问题。这里的具体方式不影响语义
- 聪明的编译器可能让它们共享代码，以提高内存利用的效率

内部定义

- 考虑带有内部定义的过程：

```
(define (sqrt x)
  (define (good-enough? guess) (< (abs (- (square guess) x)) 0.001))
  (define (improve guess) (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess) guess (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

- 求值 (sqrt 2) 时建立框架 E1，其中包含形参 x 的约束和各个内部过程的约束
- 每个内部过程名约束到一个过程对象（包括代码和一个环境指针）。内部过程的环境指针都指向 E1
- 首次调用 good-enough? 时的现场情况如右图



内部定义

- 建立过程对象时
  - 内部过程的名字与相应过程对象的约束在一个局部框架里，与其他框架里的同名对象（变量或过程）无关
  - 内部过程对象的环境指针指向外围过程调用时的环境，因此内部过程里可以直接使用其外围过程的局部变量（形式参数等）
- 每次调用有内部过程定义的过程时，将新建一个框架
  - 包括重新建立其中的各内部过程对象
  - 不同过程对象是否共享代码是系统的实现细节，不影响语义

用变动数据做模拟

- 下面考虑如何用有局部状态的对象做模拟
- 前面提出，建立数据抽象时数据结构基于其构造函数和选择函数描述
- 现在考虑由对象（其状态不断变化）构成的系统。为模拟这种系统，复合数据对象的状态也应该能随计算进程变化，需要修改状态的操作。这种操作称为改变函数（mutator）
- 例如，为模拟银行账户，表示它的数据结构应支持余额设置操作：

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

- 我们用序对作为构造复合对象的通用粘合机制

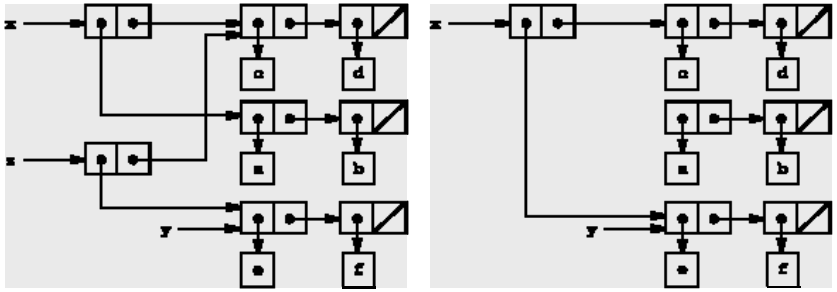
  要用于构造状态可变的对象，就需要有修改序对内容的操作

- 序对的改变操作是 set-car! 和 set-cdr!
  - 各有两个参数，它们的作用是将第一个参数（序对）的 car 或 cdr 修改为以第二个参数为值

表结构的变动

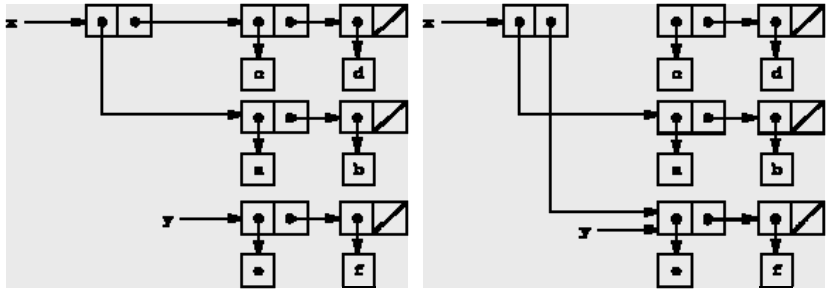
假设 x 的值为 ((a b) c d)，y 的值为 (e f)

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)：



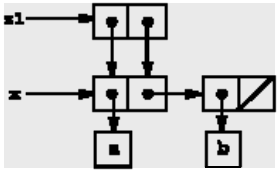
- set-car! 和 set-cdr! 修改已有的表结构（是破坏性操作）
- cons 通过建立新序对的方式构建表结构（没有破坏性）
- 可以用建立新序对的操作 get-new-pair 和两个破坏性操作 set-car! 和 set-cdr! 实现 cons

共享和相等

- 赋值引起“同一个”和“变动”问题。当不同数据对象共享某些序对时，问题会暴露出来。例：

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

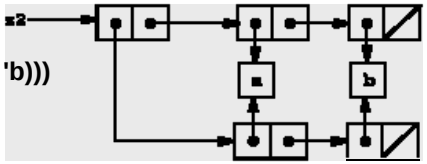
  得到的状态如右图



- 下面表达式产生另一个结构

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

  Scheme 里的符号总是共享的



- z1 和 z2 貌似表示“同样”的表。只做 car/cdr/cons 不能察觉其中是否存在共享。如果能修改表结构，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

- eq? 检查两个表达式的值是否为共享同一个实体。例如 (eq? x y) 检查 x 和 y 的值是不是同一个对象（引用同一个对象）
  - 由于 Scheme 里符号的唯一性，(eq? 'a 'a) 得真
  - cons 总建立新序对，(eq? (cons 'a 'b) (cons 'a 'b)) 得假
  - 对于前一页建立的两个情况，(eq? (car z1) (cdr z1)) 得真，而 (eq? (car z2) (cdr z2)) 得假
- 下面将看到
  - 结构共享能极大地扩充序对能表示的数据结构的范围
  - 存在共享时，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而为，那就很可能造成错误
  - 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，否则可能导致严重程序错误

改变也就是赋值

前面介绍过用过程表示序对：

```
(define (co
  (define (d
    (cond ((e
          ((eq?
          (else
   dispatch)
  (define (ca
  (define (cd
```

有了变动操作，这一框架仍然可以用：

```
(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：要在语言里支持变动，只需引进一个赋值就够了

set-car!/set-cdr! 都可通过赋值实现

队列

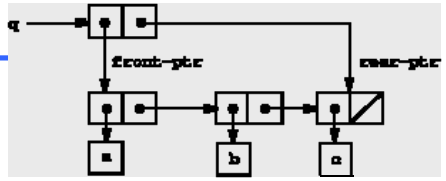
- 用 set-car! 和 set-cdr! 能构造出一些基于 car/cdr/cons 不能实现的数据结构。特点是同一个数据结构，可以随着操作改变其内部
- 考虑构造一个队列。其操作实例：

```
(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
```

- 基本操作：
  - 创建：(make-queue)
  - 选择：(empty-queue <q>) 和 (front-queue <q>)
  - 改变：(insert-queue <q> <item>) 和 (delete-queue <q>)

队列

- 采用如右图的队列表示



- 先定义几个辅助过程（为清晰）：

```
(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))
```

- 前端指针空时认为队列空；空队列是前后端指针均为空的序对：

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

(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))))
```

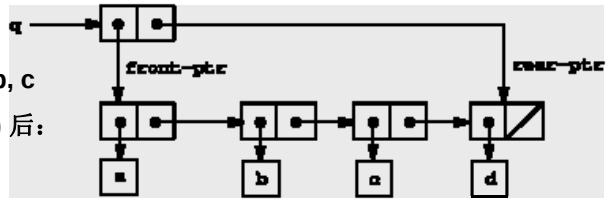

队列

- 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：

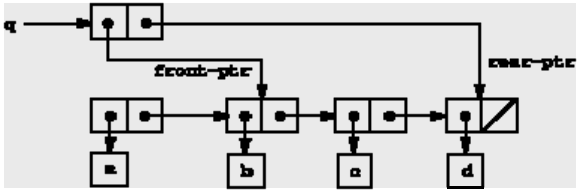


队列

- 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

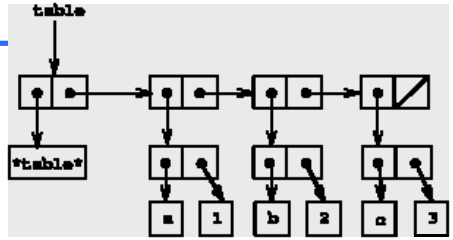


Scheme 系统输出功能不理解队列结构，需要自己定义输出队列的过程

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操作的信息。现在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)
        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

- 为特定关键码关联新值时，需要先找到该关键码所在的序对，而后修改其关联值。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
                  (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*))
```

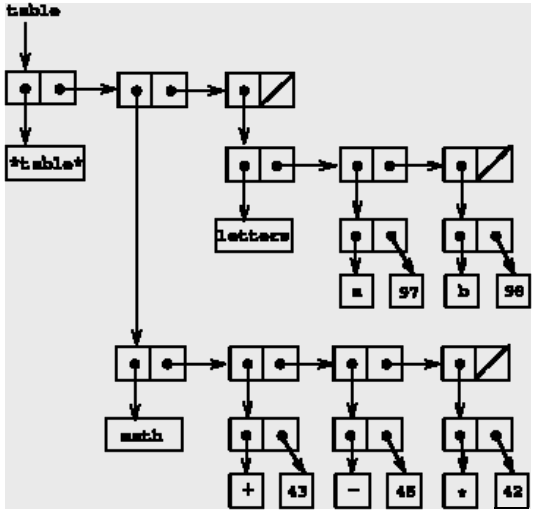

表格：两维表格

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

- 查找时，用关键码逐层查找

```
(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)))
```

- 插入关键码时逐层查找，可能需要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“表格生成器”生成表格对象，其中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 lookup/insert! 不需要 table 参数，直接用 local-table 关联的表格

- 创建一个操作表格（创建其他表格也一样）：

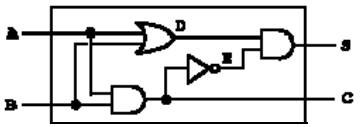
```
(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))
```


数字电路模拟器

- 下面介绍一个大型实例：是数字电路模拟器
- 复杂的数字电路应用于许多领域，其正确设计也非常重要。数字电路由一些简单元件构成，通过复杂连接形成复杂的行为。为理解正在设计的电路，需要做模拟
- 数字电路模拟是事件驱动的模拟的代表。这是重要的计算机应用领域，基本想法：系统活动中发生一些事件，事件又会引发其他事件
- 电路的计算模型由一些对象构成，对象对应于各种基本电路元件
  - 连线在对象间传递数字信号（信号只能是 0 或 1）
  - 功能块有若干输入端口和输出端口，从输入信号计算输出
  - 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。各种逻辑门都有若干单位时间的延迟

数字电路模拟器：连线

- 连接基本功能块，可得到更复杂的功能块。如半加器，有输入 A 和 B，输出 S（和）和 C（进位）。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



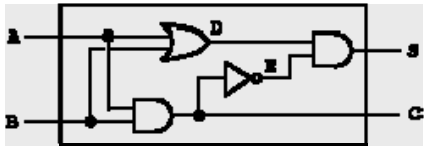
- 下面构造一个模拟数字电路的程序。其中
  - 连线用计算对象表示，它们能保持信号
  - 功能模块用过程模拟，它们产生正确的信号关系
- 构造连线的基本操作是 make-wire。例如构造 6 条连线：

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

数字电路模拟器：组合和抽象

- 将反门、与门和或门连接起来，可以构成半加器：

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

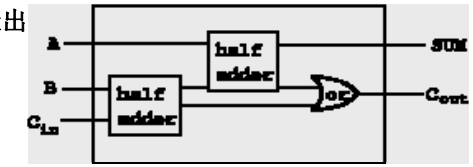


- 更好的方式是把构造操作定义为过程，取 4 个连线参数：

```
(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))
```


数字电路模拟器：语言结构

- 用两个半加器和一个或门可构造出一个全加器。过程定义：




```
(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))
```

- 以构造出的功能块作为部件，可以继续构造更复杂的电路
- 这就建立了一种语言，可用于构造具有任意复杂结构的数字电路
  - 基本功能块是这个语言的基本元素
  - 将功能块用连线连接，是这个语言的组合机制
  - 将复杂连接方式定义为过程，是这里的抽象机制

数字电路模拟器：基本功能块

- 基本功能块实现特定效果，使一条连线的信号能影响其他连线。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> 上的信号值设置为新值
  - (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）
- 基本功能块基于这些操作定义。反门在给定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)    ; 在 input 连线对象登记一个无参过程
  'ok)
(define (logical-not s)
  (cond ((= s 0) 1)    ((= s 1) 0)
        (else (error "Invalid signal" s))))
```


数字电路模拟器：基本功能块

- 与门有两个输入，这两个输入得到信号时都要执行动作
- 过程 logical-and 与 logical-not 类似

```
(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 ... ...))
```

- 或门、异或门等可以类似定义
- inverter-delay 和 and-gate-delay 等是模拟中的常量，需要事先定义

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))
```

执行无参过程 proc

辅助过程 call-each 逐个调用过程表里的过程（都是无参过程）

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。调用 make-wire 返回新的连线对象
- 连线状态被连接在其上的功能块共享，一个部件的活动通过交互影响连线状态，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

- 为实现 after-delay 操作，这里的想法是维护一个待处理表，记录需要处理的事项。该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，要求在给定时间运行给定过程

  (current-time <*agenda*>) 返回当前时间

- 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 ) )
```


数字电路模拟器：模拟

- 模拟驱动过程 propagate 顺序执行待处理表中的项。处理中可能加入新项。只要待处理表不空，模拟就会继续进行

```
(define (propagate)
  (if (empty-agenda? the-agenda)
      'done
      (let ((first-item (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

- 实现一个“监视器”，把它连在连线上时，该连线的值改变时它就会输出信息

```
(define (probe name wire)
  (add-action! wire
               (lambda ()
                 (newline)
                 (display name)
                 (display ", Time = ")
                 (display (current-time the-agenda))
                 (display "  New-value = ")
                 (display (get-signal wire)))))
```

数字电路模拟器：模拟实例

- 初始化：

```
(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，其中两条安装监视器：

```
(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
(probe 'carry carry)
carry 0  New-value = 0
```

- 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，其中时间段按时间排序

  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

- 需要从待处理表删除第一项时，应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：

```
(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)))))
```

至此数字电路模拟系统的开发完成，请大家自己总结一下

总结

- 变动和赋值，是模拟复杂系统的有力手段
  - 导致计算的代换模型失效
  - 需要用复杂的环境模型来解释计算过程
  - 注意 lambda 表达式建立过程对象，调用过程时创建新框架
- set! 改变变量的约束，set-car! 和 set-cdr! 改变序对成分的约束
- 我们用有局部状态的过程实现具有局部状态变量的对象
- 注意 set! 和 define 的不同意义
- 基于状态改变建立的数据结构：队列和表格
- 最后的实例很有意思。其中：
  - 用有局部状态的对象反映系统状态，用消息传递实现对象间交互
  - 定义了好用的语言，支持所需对象的构造和组合。用过程作为组合手段。对象的构造具有闭包性质
  - 容易在 OO 语言里模拟，请考虑如何在 C 语言里实现