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
- 后面有两个基于状态模拟的大型实例
  - 数字电路模拟
  - 约束传播语言

回顾：有局部状态的对象

- 扩充的创建银行账户的过程，账户可以提款和存款：

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown req -- MAKE-ACCOUNT" m))))
  dispatch)
```

- 返回一个具有局部状态的对象（是一个过程）

  以相应消息作为输入，该对象将返回过程 **withdraw** 或 **deposit**

C语言：带局部状态的对象？

- 在 C 语言里，可以定义有局部状态的过程（函数）吗？
- 考虑利用函数的局部变量
- 定义一个简单的计数器过程：

```
typedef enum ACCmd {reset, inc, dec} ACCmd;

int counter(ACCmd command) {
  static int count = 0;
  switch (command) {
    case reset: count = 0; break;
    case inc: count++; break;
    case dec: count--; break;
  }
  return count;
}
```

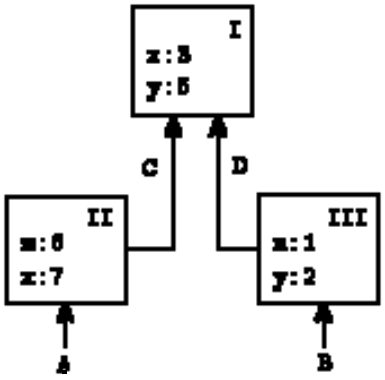
- 只能定义单一的包含简单状态的对象，定义包含复杂状态的对象或者对象生成器需要更复杂的结构和使用规则

环境和求值

- 一般的组合表达式都包含变量
  - 求值表达式的过程中需要使用变量的值
  - 变量的值需要记录在某个地方
  - 这种记录变量约束的结构称为“环境”
- 环境确定了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
- 代换模型：将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
  - 现在变量已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

求值的环境模型

- 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

- 实际上，前面的代换模型也需要环境的支持
  - 由 **define** 引进的变量，**define** 的值需要保存在环境里
  - 基本过程和用户定义过程的定义都需要保存在环境里，使用时通过检索环境得到相应的定义

环境模型下的求值

- 为描述解释器的意义，我们假定有一个全局环境
  - 它只包含一个全局框架，其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
  - 在基于新模型的求值过程中，过程定义，调用和退出导致的环境变化是求值过程中最重要的，最需要关注的事项
- 首先，对 **lambda** 表达式的求值将得到一个过程对象。过程对象是一个对 (*c*, *e*)，其中 *c* 是过程的代码，*e* 是环境指针：
  - 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
- 下面通过几个例子说明求值过程中的一些基本情况，包括：求值过程中框架的创建；过程对象的创建；等等

环境模型下的求值：建立过程对象和约束

- 在全局环境中求值

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

  实际上就是求值：

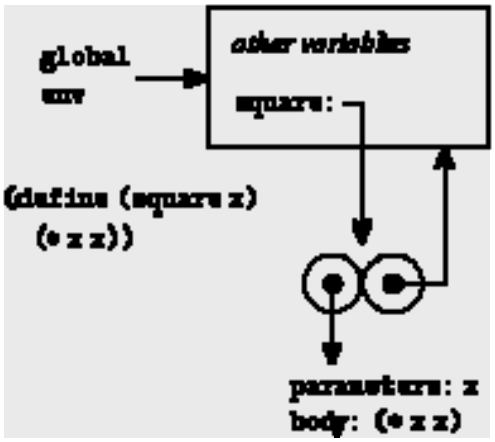
  (define square (lambda (x) (* x x)))

- 求值效果是在全局环境增加了 **square** 的约束，它约束于新建的过程对象
- 右图：在原有其他变量约束之外，新建立了 **square** 的约束

  square 约束于一个过程对象

  其体部分包括参数和过程体代码

  环境指针指向全局环境，也就是这个 **lambda** 表达式的求值时所在的环境

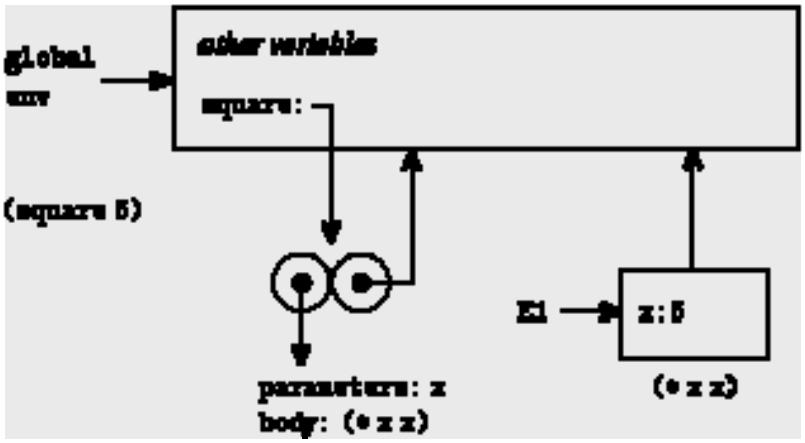


环境模型下的求值：过程应用

- 在全局环境里求值组合表达式时，
  - 首先根据过程对象的参数表和实参建立新框架，以全局环境框架作为外围框架，建立以新框架为当前框架的新环境
  - 在新环境里求值过程体

求值 **(square 5)**：

- 过程应用表达式的求值先创建新环境 **E1**，建立一个新框架作为当前框架，其中 **x**（形参）约束到 **5**（实参的值）
- 在 **E1** 中求值过程体 **(* x x)** 得到结果 **25**



环境模型下的求值规则

环境模型下的求值有两条基本规则：

- 将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，框架里存入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**
- 注意：
  - 上面规则中说的环境未必是全局环境，可以是任何的环境
  - 记录过程对象中的环境，建立新环境的规则都是一样的

define 和 set!

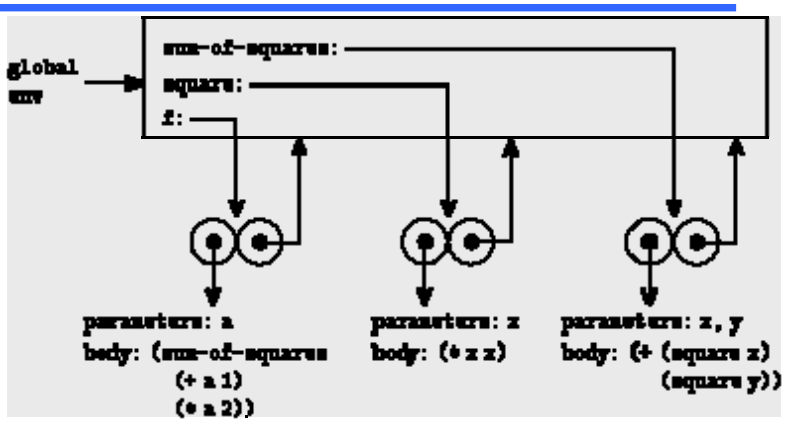
- 现在可以清晰地说明 **define** 和 **set!** 的差异了
- 用 **define** 的作用是在环境的当前框架里定义一个符号：
  - 在当前框架里建立一个约束，将被定义符号约束到给定值
  - 如果当前框架已有这个符号，则改变其约束（注意书上的注释）
- **(set! <*变量*> <*value*>)** 的作用：
  - 在当前环境里查找 **<*变量*>** 的约束。如果在当前框架里找到，就确定这一约束；否则到其外围框架里查找。这一查找过程可以沿着外围环境关系前进许多步
  - 将找到的约束中该变量的约束值修改为由 **<*value*>** 计算出的值
  - 如果环境中没有**<*变量*>**的约束（查找过程达到了全局框架仍然没找到），就报告变量无定义错误
- 新求值规则比代换模型复杂很多。但它表现了 **Scheme** 解释器工作方式，可以基于这个模型实现 **Scheme** 解释器（第4章）

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)))
```

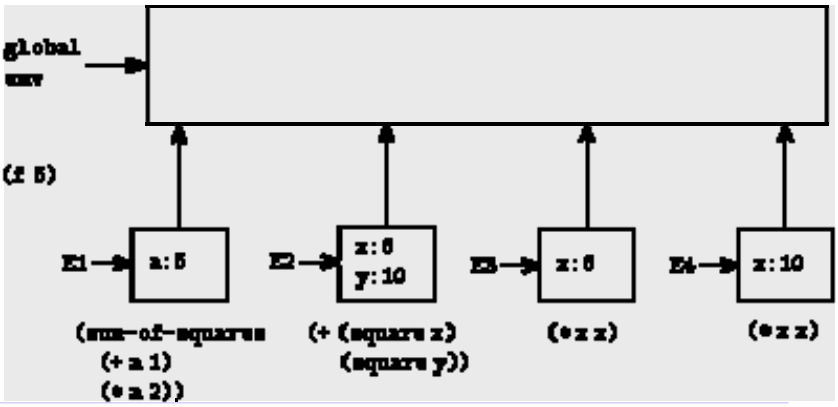
三个定义建立起的环境见图



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

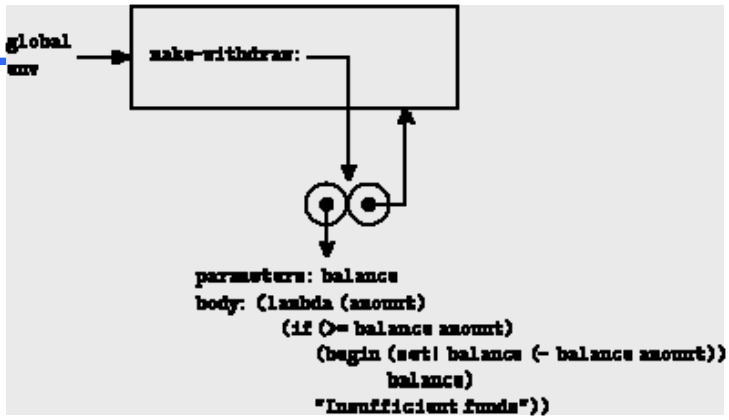


框架和局部状态

<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")))
```

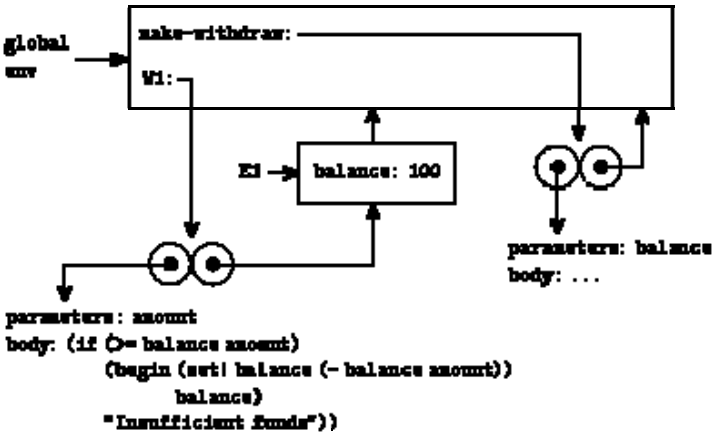


调用

```
(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



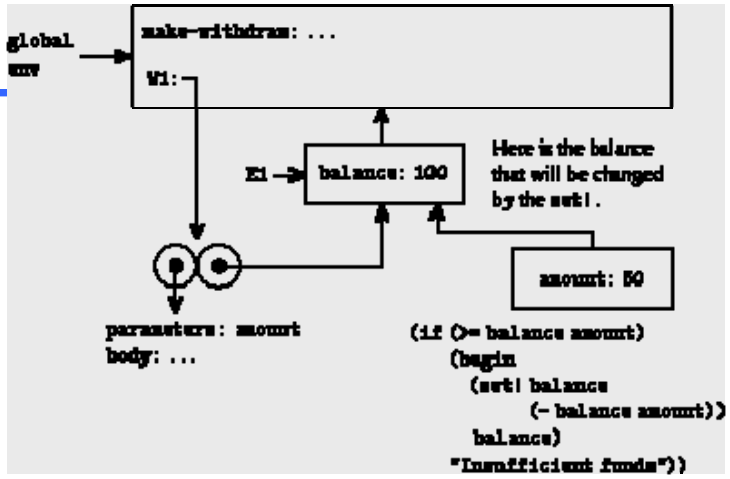
框架和局部状态

考虑过程调用：

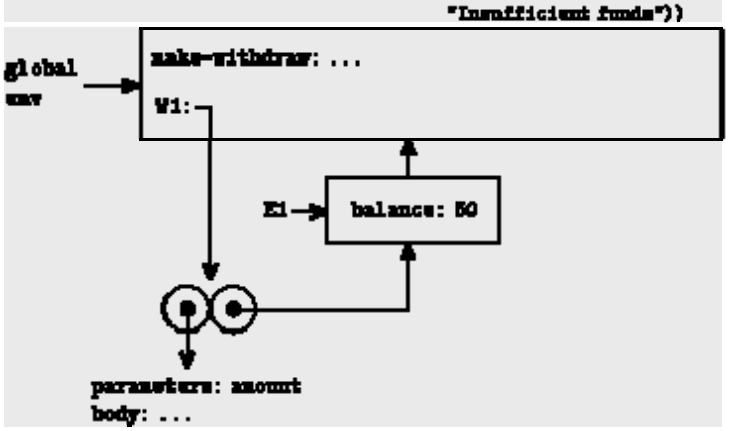
(w1 50)

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**



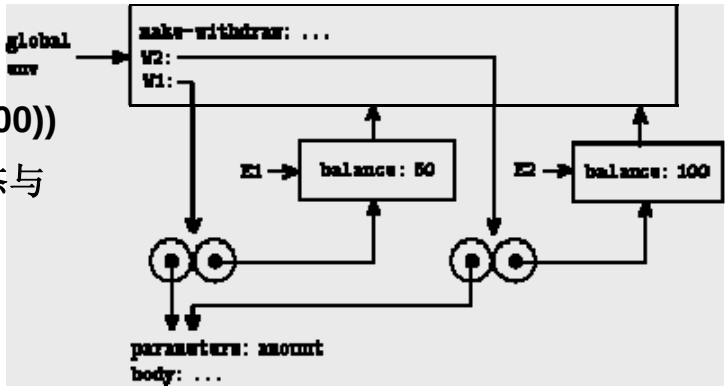
- 再调用 **W1** 将建立新框架，与上面建立的框架无关，但其外围框架仍是 **E1**
- 过程求值中将再次找到包含 **balance** 的框架 **E1** 并修改 **balance** 的约束值

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



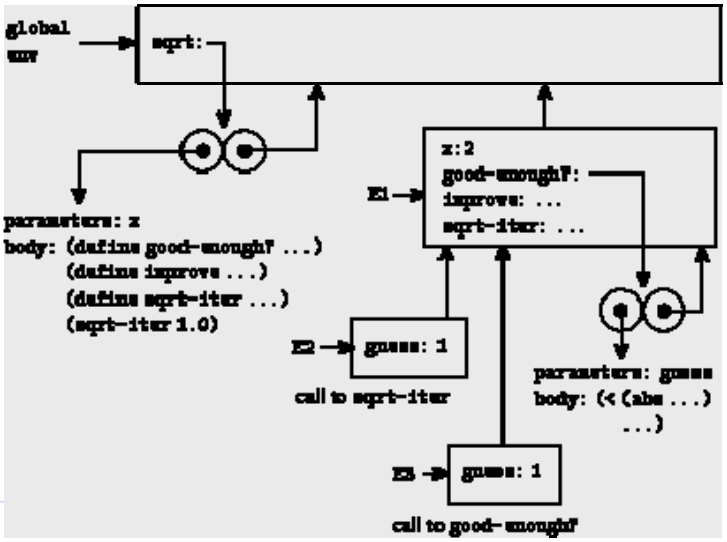
- 两个提款处理器是两个过程对象，各自独立变化
- 这里两个提款处理器（过程对象）的代码完全相同
  - 两者是共享同一份代码还是各有一份代码，是系统的实现细节
  - 具体实现方式并不影响程序的语义
- 聪明的编译器可能让它们共享代码，以提高内存利用的效率

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

- 现在可以看清楚与内部过程定义有关的一些情况
- 在建立过程对象时
  - 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
  - 代码的处理见前面说明，不同过程对象之间是否共享代码是系统的实现细节，不影响语义
- 应注意过程对象里的环境指针的作用，它决定所建立的新环境的结构，即，决定新的当前框架的外围框架是哪个框架

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要能随着计算进程变化
  - 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*>)
- 要用序对作为构造复合对象的通用粘合机制，出现了新问题：
  - 需要构造的是状态可变的对象
  - 因此需要有修改序对内容的操作

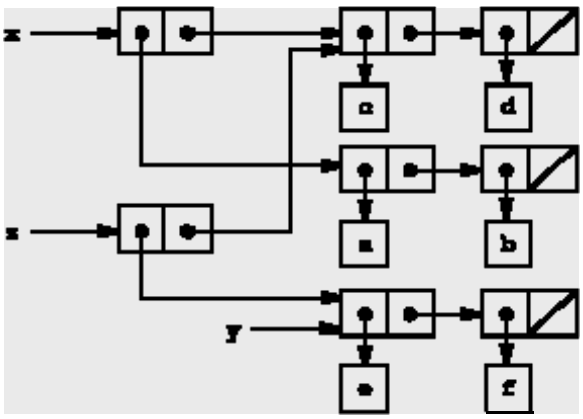
表结构的变动

- 序对的改变操作是 **set-car!** 和 **set-cdr!**

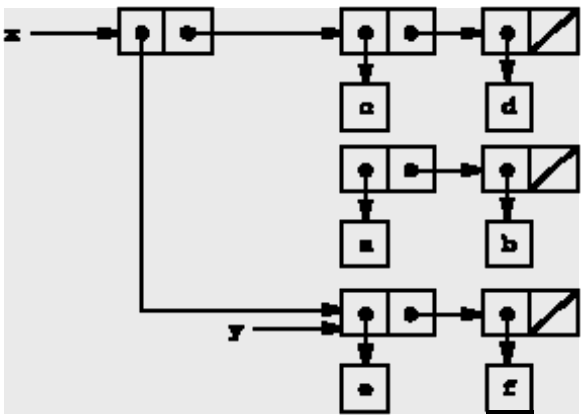
  各有两个参数，作用是修改作为其第一个参数的序对的 **car** 或者 **cdr** 修，改为以第二个参数为值

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

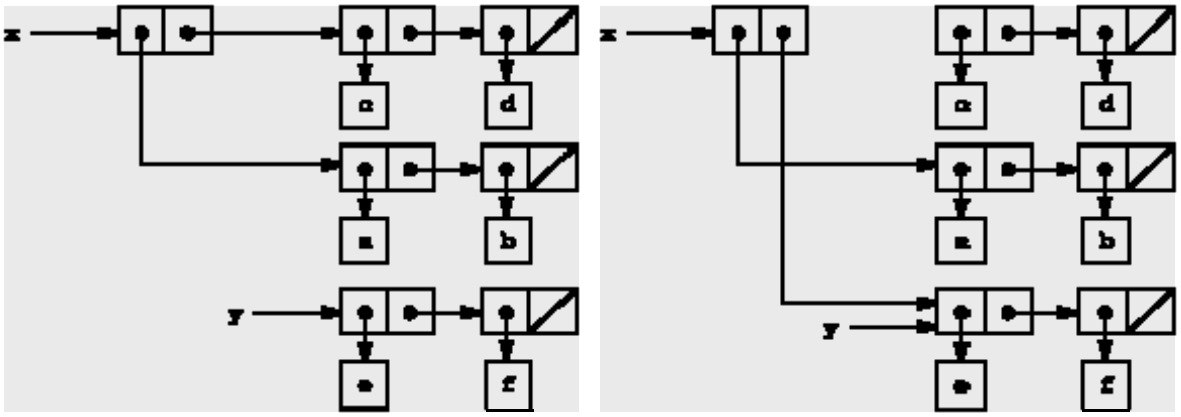


(set-car! x y) 得到：



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)：



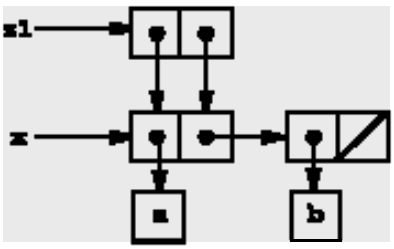
- **set-car!** 和 **set-cdr!** 修改已有的表结构（是破坏性操作）
- **cons** 通过建立新序对的方式构建表结构（没有破坏性）
- 可以用建立新序对的操作 **get-new-pair** 和两个破坏性操作 **set-car!** 和 **set-cdr!** 实现 **cons**

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))

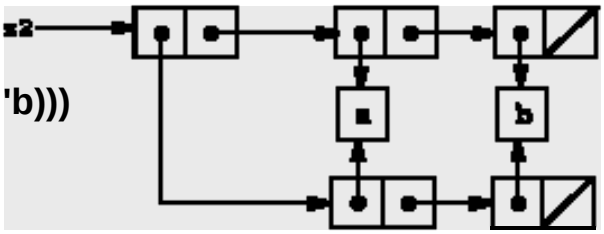
  得到的状态如右图



- 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

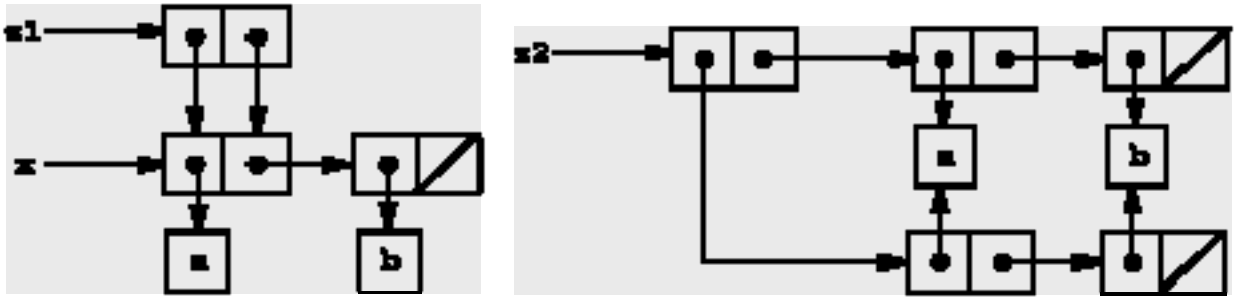


- **z1** 和 **z2** 貌似表示“同样”的表。只做 **car/cdr/cons** 不能察觉其中是否存在共享。如果能修改表结构，就会暴露共享的情况

(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

- 前面介绍 **eq?** 可用于检查两个符号是否相同，实际上它是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
  - 对前一页建立的两个情况（下图），**(eq? (car z1) (cdr z1))** 得真，而 **(eq? (car z2) (cdr z2))** 得假



结构共享

- 下面将看到，有了结构共享
  - 序对能表示的数据结构的范围将得到很大的扩充
  - 特别是：能够表示任意复杂的数据对象，在其存在期间，其标识不变，而其内部的状态不断变化
  - 这种结构可用于模拟真实世界中复杂的不断变化的对象
- 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

- 前面介绍了用过程表示序对的技术：

```
(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))
```

- 可以在此基础上实现 Scheme 系统

改变也就是赋值

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：要在一个语言里支持变动，只需为其引进一个赋值就足够了

set-car!/set-cdr! 都可通过赋值实现

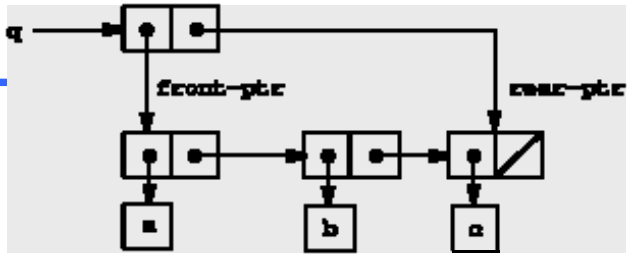
队列

- 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
- 下面考虑构造一个队列。下面是一些操作实例：

```
(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
```

- 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>)**

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))))
```

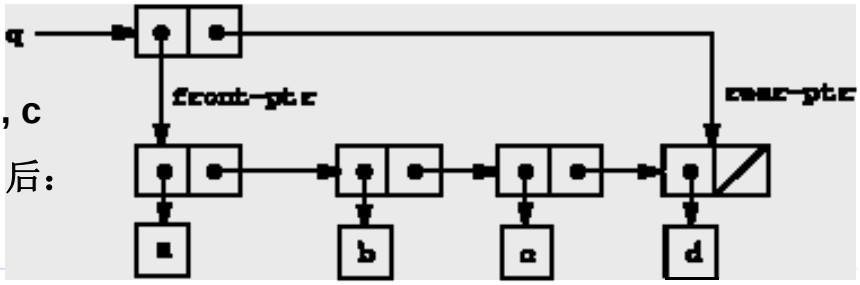
队列

- 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：

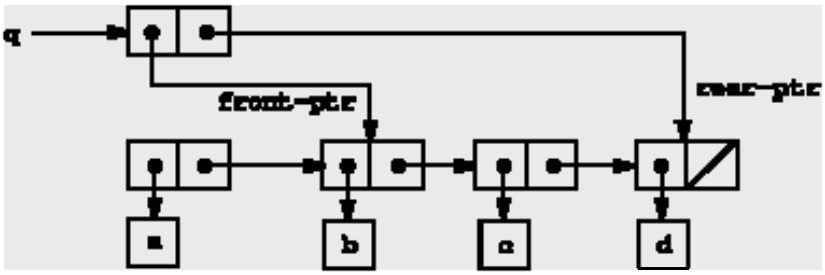


队列

- 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)))
```

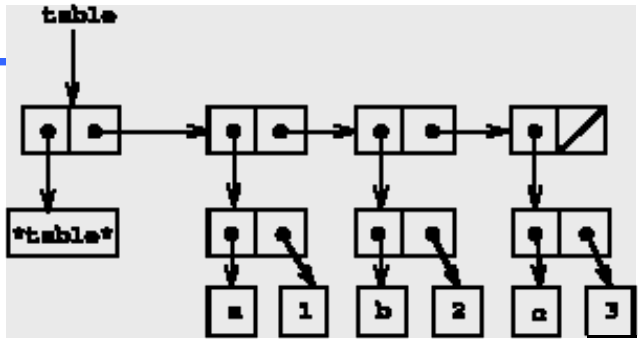
(delete-queue! q)
之后



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*))
```

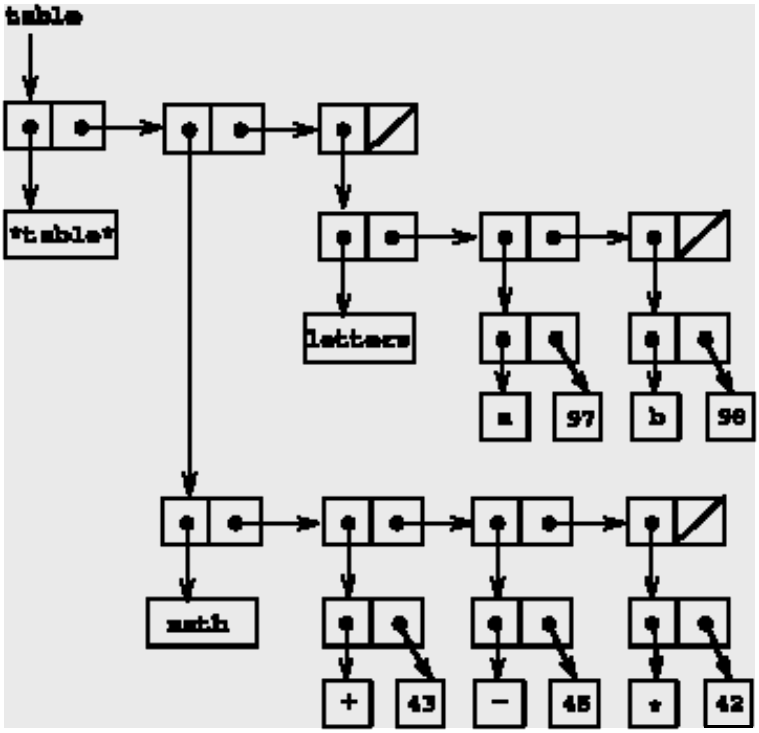
表格：两维表格

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

它们将直接使用 **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!))
```

- 这个表格就可以支持前一章讨论的“数据驱动的程序设计”，那里需要的就是一个记录名字类型和操作关联的表格，

  两维表格正好用于建立操作名和类型信息对的索引

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
  - 对 **lambda** 表达式的求值建立新的过程对象
  - 调用过程时需要创建新框架
- 变动操作
  - **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