3。模块化，对象和状态(I)

本章讨论与状态有关的编程问题。本节课讨论：

- 赋值和局部状态
- 基于状态变化的程序设计
- 引入赋值的得与失
  - 函数式和命令式程序设计
  - 命令式程序设计的缺陷
- 求值的环境模型
  - 环境模型中的求值规则
  - 建立过程对象，过程应用，局部状态，内部定义

系统设计和组织的策略

- 前面讨论的主要问题是：
  - 如何组合基本过程和基本数据
  - 如何构造各种复合对象（组合过程/数据）
  - 抽象在控制和处理程序复杂性中的重要作用
- 有效设计大型系统，还需要一些组织系统的原则
  - 只有一集高效算法，不足以构造出良好的大型系统
  - 系统的功能分解，结构组织和管理与算法一样重要（或更甚之）
- 为了系统化地完成设计，特别需要一些模块化策略
  - 模块化就是把复杂系统分解为一些边界清晰、易于独立理解的部分
  - 每个部分的内部成分之间关系较密切，内聚力强；不同部分具有良好的功能分离，相互之间的交互清晰、容易认识和处理
  - 良好模块化分解出的部分可以分别设计，分别开发和维护

设计和组织策略

- 假设构造一个系统的目标是希望模拟一个真实世界的系统
  - 一种有效策略：根据被模拟系统设计程序的结构
  - 针对实际物理系统中的每个对象，构造一个对应的程序对象
  - 针对实际系统里的每种活动，在计算系统里实现一种对应操作
  - 让所开发的系统的活动比较直接地反映被模拟系统的活动
- 采用这种设计系统策略，有一个重要问题必须考虑
  - 真实世界的系统是变化的（相应的，人的认识也不断深入）
  - 这些变化在人工系统里的反映，通常是需要在系统里增加新对象或新操作，或者需要修改已有对象和操作的行为
  - 为了有效完成模拟，我们希望构造出的模拟系统在遇到变化时
    - 在修改时只需要局部做，不需要大范围改变程序，
    - 在扩充时只需简单加入对象或操作，局部修改/加入相关操作

设计和组织策略

- 本章将讨论两种系统的组织策略：
  - 把系统看成是由一批相互作用的对象组成
    - 真实系统中的对象随着时间的进展不断变化
    - 模拟它们的系统对象也吸引相应地变化
  - 把系统看作一种信号处理系统

    关注流过系统的信息流
- 基于对象和基于流的设计途径都对语言提出了新要求
  - 基于对象，需要有在存在期间保持其标识但本身又能变化的对象

    这是一种新的计算模型，带来许多本质性变化，包括有关计算的基本观点，基本操作，抽象的计算模型及其实现
  - 基于流的技术要求一种延时求值技术

    常规程序工作者也正在考虑基于流的计算的描述问题

对象：状态和变化

- 对象的观点是对世界的一种看法：
  - 世界由一批事物(对象)组成，每个对象有其状态和行为方式
  - 对象的状态随时间不断变化，其行为受到历史的影响
- 例：一个银行账户有状态
  - 对“能取出100元吗”的回答依赖于该账户此前存钱和取钱的历史
  - 对同一个问题，不同时刻的回答可能不同
- 为模拟真实世界的对象，程序里需要用状态变量表示计算对象的状态
  - 选择什么样的状态变量（如，记录余额还是记录全部交易历史），要根据对实际对象行为的认识和所做模拟的（预期）目标
  - 状态变量的信息应足以确定对象的后续行为
- 系统里的不同对象相互有联系，可能通过交互影响彼此的状态

  有些对象联系更紧密，可能形成分组，或构成大系统里的子系统

模拟真实世界

- 基于对象状态的观点是组织系统的计算模型（程序）的有力手段

  实际上，这也就是常规的程序设计采用的计算模型
- 基于状态的观点倡导的系统模块化方式：
  - 把系统分解为一组计算对象，用它们模拟真实系统中对象的行为
  - 用计算系统里所有计算对象的全体模拟真实系统的整体行为
- 现实世界里真实对象随着时间而改变状态，要模拟它们
  - 程序里就需要有随着运行不断改变状态的对象
  - 为此需要改变程序对象状态的操作

    主要是赋值操作
  - 赋值，就是修改对象的状态变量

    常规语言中的变量，OO 里的对象等等，都是对象

局部状态变量

- 现在考虑模拟一个银行账户：假定过程 **withdraw** 是从该账户提取现金的操作。可能出现下面操作序列：

```
(withdraw 25)
75
(withdraw 60)
15
(withdraw 25)
"Insufficient funds"
(withdraw 10)
5
```

- 上面操作序列中两次调用 **(withdraw 25)** 取 **25** 元，得到的结果不同

  与前面计算数学函数的过程相比，这个过程的性质完全不同

  withdraw 的行为与时间有关，依赖于某些随时间改变状态的变量

  以前的操作历史会影响后面操作的行为

变量和赋值

- 考虑银行账户和 **withdraw** 的实现
- 用变量 **balance** 表示账户余额，**withdraw** 定义为依赖它的过程：

```
(define balance 100)

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

  begin 是特殊形式，它逐个求值参数，返回最后一个参数的值

- 在写 **(set! balance ...)** 的地方不能用 **(define balance ...)**

  define 在其所在的定义域里创建变量及其约束

  在这里将建立 **withdraw** 里局部的 **balance**，不能实现需要的功能

  更具体的语义细节后面介绍

变量和赋值

- **set!** 表示赋值

  (set! balance (- balance amount))

  使 **balance** 重新约束到由表达式 **(- balance amount)** 计算出的值

- 赋值操作 **set!**

  set! 是赋值运算符，一般形式是

  (set! <*name*> <*new-value*>)

  set! 找到最接近其使用处的名字为 **<name>** 的已经有定义的变量，修改它的值约束

说明

- 一般形式的 **lambda** 表达式和 **define** 形式

  其“体”部分可以写多个表达式，其求值方式与 **begin** 表达式一样

- **lambda** 表达式的参数表之后可以写多个表达式，其一般形式是

  (lambda (x …) <exp1> … <expn>)

  语义：顺序求值 **<expi>**，以最后一个表达式的值作为值

- **define** 定义过程的一般形式与此类似：

  (define (f x …) <exp1> … <expn>)

  调用时逐个求值过程体中的表达式，以最后一个表达式的值为值

- 如果表达式不改变变量状态，写多个表达式就完全没必要

  有了 **set!**（包括有了 **define**），前面表达式的求值有可能影响后面的表达式，在过程体里写一系列的表达式就有用了

  下面有很多这方面的例子

局部状态变量

- 前面定义可行但不妥：
  - **balance** 是全局的，任何过程都能访问和修改
  - 不安全
- 应把它改为 **withdraw** 里的局部变量：

```
(define new-withdraw
      (let ((balance 100))
        (lambda (amount)
          (if (>= balance amount)
              (begin (set! balance (- balance amount))  balance)
              "Insufficient funds"))))
```

let 创建一个包含局部变量 **balance** 的环境，并将它初始化为**100**

new-withdraw 的功能和前面的 **withdraw** 一样

但 **balance** 是局部的，任何其他操作都不能触动它

局部状态变量

- **set!** 和局部变量的结合形成一种通用编程技术
  - 下面将一直用这种技术构造有局部状态的计算对象
  - 这一技术带来一个新问题：代换模型对这种程序失效了，需要新的计算模型（后面介绍）
- 下面考虑 **new-withdraw** 的一些问题和变形
- 把 **new-withdraw** 修改为一种创建“提款处理器”的过程：

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

形参是局部变量，定义 **make-withdraw** 可以不用 **let**

局部状态变量

- 调用 **make-withdraw** 得到一个新的提款处理器，如：

```
(define acc1 (make-withdraw 100))
```

- 使用实例:

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))
(W1 50)
50
(W2 70)
30
(W2 40)
"Insufficient funds"
(W1 40)
10
```

建立的两个提款处理器相互无关

局部状态变量

- 可以扩充为创建银行账户的过程，使账户不仅可提款，还可存款：

```
(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)
```

这个过程返回一个带局部环境的对象（过程）

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

局部状态变量

- 使用实例：

```
(define acc (make-account 100))
((acc 'withdraw) 50)
50
((acc 'withdraw) 60)
"Insufficient funds"
((acc 'deposit) 40)
90
((acc 'withdraw) 60)
30
```

- 可创建任意多个独立的账户对象：

```
(define acc2 (make-account 200))
((acc2 'withdraw) 50)
150
```


赋值：得与失

- 赋值给程序的理解带来了新问题

  另一方面，把系统看成一组有内部状态的对象，也是实现模块化设计的强有力技术

- 下面看一个例子
- 实例：设计随机数生成过程 **rand**，希望对它反复调用能生成一系列整数，这些数具有均匀分布的统计性质

  假定已有一个过程 **rand-update**

  对一个数调用它将得到下一个（随机）数

  x_2 = (rand-update x_1)

  x_3 = (rand-update x_2)

  反复做可得到一个（具有随机性的）整数序列

随机数生成

- 可定义一个带局部状态的过程 **rand**，实现一个随机数生成器：

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

rand-init 取某个整数（或具有整数值的变量）

- 也可以直接使用函数 **rand-update** 生成随机数

使用形式 x_2 = (rand-update x_1)

但这种方式使用起来很麻烦

  - 需要用新变量接受结果
  - 每次使用都需要注意送给它的参数
  - 如果用错，生成的整数序列的随机性就没保证了

使用或不使用赋值：得与失

- 下面考虑引入赋值的得与失
- 先看一个对比实验，看看使用赋值或不使用赋值的情况

比较程序的清晰和模块化性质

实例：利用随机数功能，实现蒙特卡罗模拟

- 蒙特卡罗方法：
  - 用大量随机数做试验
  - 统计试验的结果得到相应的结论
- 具体试验：
  - 两个整数之间无公因子的概率是 $6/\pi^2$
  - 下面用蒙特卡罗方法验证这一结果

基于状态和赋值的解

- 采用状态变量和赋值（使用 rand）的定义

```
(define (cesaro-test) (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0) (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1) (+ trials-passed 1)))
          (else (iter (- trials-remaining 1) trials-passed))))
  (iter trials 0))

(define (estimate-pi trials)   ; trials: 试验次数
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))
```

- 定义中的 monte-carlo 是核心过程
  - 系统的结构很清楚
  - 很容易看清，这个实现确实反映了我们的想法

不使用赋值的解

- 不用赋值而直接用 rand-update，也可以写出程序：

```
(define (estimate-pi trials)
  (sqrt (/ 6 (random-gcd-test trials random-init))))

(define (random-gcd-test trials initial-x)
  (define (iter trials-remaining trials-passed x)
    (let ((x1 (rand-update x)))
      (let ((x2 (rand-update x1)))
        (cond ((= trials-remaining 0)
               (/ trials-passed trials))
              ((= (gcd x1 x2) 1)
               (iter (- trials-remaining 1) (+ trials-passed 1) x2))
              (else
               (iter (- trials-remaining 1)  trials-passed  x2))))))
  (iter trials 0 initial-x))
```

方法可行。但可以看到，只用函数式过程，random-gcd-test 必须显式操作随机数 x1 和 x2，迭代时把 x2 作为新输入

蒙特卡罗模拟的分析

- 做这个试验需要两个随机数，可能有试验用到三个或更多随机数

  函数式写法必须时时注意维护这些随机数，越多越难维护

- 比较两种实现：
  - 不用赋值的代码里没有很好地体现出蒙特卡罗方法本身（没有这个概念），相关行为与其他操作交织在一起
  - 在用 **rand** 的版本里蒙特卡罗方法是独立过程，随机数的使用细节被屏蔽在 **rand** 过程的内部
- 看到的一些现象：
  - 在复杂计算中，从其中一部分观察，其他部分都像在随着时间不断变化，它们通常都隐藏了一些变化的细节（内部状态）
  - 如果想基于这种认识这样分解系统，最直接的方式就是用计算对象模拟系统随时间变化的行为，用局部变量模拟部分的内部状态，用赋值模拟状态变化

简单总结

- 本例讨论了处理状态变化的两种方式：
  - 通过显式计算

    其中通过额外的参数传递随时间变化的状态
  - 采用局部状态变量和赋值

    自然地利用变化的状态
- 后一方式可能得到更好模块化的系统

  但也带来许多麻烦（下面讨论）
- 在没有赋值的时候
  - 以同样参数调用同一过程总得到同样结果
  - 这种过程就像是在计算数学的函数
  - 无赋值的编程称为函数式编程

引进赋值的代价

- 有了赋值（set!），语义就不能用简单代换模型解释了。而且
  - 描述这种语言程序里的对象和赋值的理论框架都不可能是简单的
  - 不可能做出具有很漂亮的数学性质的简单模型
- 看下面两个过程：

```
(define (make-decrementer
          balance)
  (lambda (amount)
    (- balance amount)))
(define D (make-decrementer 25))
(D 20)
5
(D 10)
15
(D 10)
15
```

```
(define (make-simp-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
(define W (make-simp-withdraw 25))
(W 20)
5
(W 10)
-5
(W 10)
-15
```


引进赋值的代价

- 代换模型能解释 make-decrementer
  - 但它无法解释 make-simplified-withdraw
  - 不能解释为什么两个(W 10) 调用会得到不同结果
- 代换模型里名字实际上只是值的代号
  - 可以通过代换消除掉其中的名字
  - 代换模型就是反复用值代换名字，最后得到结果（过程体的代换也是用值代换名字的一种形式）
- 有赋值后变量就不是代表值的简单名字

  应该看作是保存值的位置，其值可以改变
- 赋值不仅破坏了简单的计算模型，其意义还更深远

  在计算模型里引入状态和变化，许多基本概念都出了问题
- 第一个问题：什么是同一个（sameness）？

引进赋值的代价

- 用同样参数两次调用 **make-decrementer**，得到的是同一个东西吗？

  (define D1 (make-decrementer 25))

  (define D2 (make-decrementer 25))

- 虽然 **D1** 和 **D2** 名字不同，但它们永远表现出同样行为

  在任何计算中任意相互替代，不会观察到任何差异

  在任何程序上下文里，把使用 **D1** 的地方换上 **D2**（或相反），不会看到程序行为有任何改变

- 调用 **make-simplified-withdraw** 两次，得到：

  (define W1 (make-simplified-withdraw 25))

  (define W2 (make-simplified-withdraw 25))

- **W1** 和 **W2** 有独立行为，在程序里不能任意相互替代

例：

(W1 20)
5
(W1 20)
-15
(W2 20)
5

同一和变化

- 如果一种语言支持“同样的东西可以相互替换”，而且这种替换不会改变表达式的值（程序的意义），称这种语言具有引用透明性
  - 纯函数式语言具有引用透明性
- 失去引用透明性，“同一个”的概念会变得很复杂
  - 现实生活也是如此，在现实中弄清什么是“同一个”也很困难
  - “你不可能两次趟过同一条河”
  - “你还是你吗？”
- 赋值打破了语言的引用透明性
  - “同样东西”的概念不再简单，不能通过描述形式直接确定
  - 要确定一个替换会不会改变表达式的意义变得很困难
  - 对程序进行推理也变得很困难
- 下面用实例说明这一问题对于编程的影响

同一和变化

- 假定 **Paul** 和 **Peter** 有银行账户，其中有 **100** 块钱。下面是这一事实的两种模拟。第一种模拟：

```
(define peter-acc (make-account 100))
(define paul-acc (make-account 100))
```

另一种模拟：

```
(define peter-acc (make-account 100))
(define paul-acc peter-acc)
```

- 两种情况下，开始时 **Paul** 和 **Peter** 都看到自己账户里有 **100** 元
  - 但随后的提款活动却会让他们发现两种情况是不同的
  - 问题：他们使用的是否“同一个”账户？
- 构造计算模型时，很容易把这两种情况弄错
  - 特别是两个账号共享时，使用改变状态的操作（赋值），操作一个账户将影响另一账户，这种影响没有在程序里明确表示

同一和变化

- 程序里两个不同描述实际指同一个东西时，称为别名（**aliasing**）
  - 如果不同数据结构之间出现了共享，从一条途径出发修改就可能产生“修改”了另一数据结构的“副作用”
  - 如果程序员对这件事不清楚，就可能由于疏忽造成程序错误（很常见，称为副作用错误）
- 另一情况：
  - 如果 **Paul** 和 **Peter** 只能检查账户而不能取款（是只读账户，其他人也不能改这两个账户），是否还应认为这两种模拟不同呢？
  - 如前面的有理数对象，一旦建立后，其内容永远也不会变

    所谓的“不变对象”（另如 **Java** 的字符串或 **Integer** 对象）
  - 如：网页，访问其原本或者副本，没差别（假定不修改）
  - 在一些情况下，究竟是实际上是引用了两个内容相同的对象，还是正好引用到同一对象，并没有实质性差别

命令式编程的缺陷

- 基于赋值的程序设计称为命令式程序设计

  命令式编程是常规软件开发中使用最广泛的编程范式

- 采用命令式程序设计的语言需要用更复杂的计算模型解释

  很容易出现一些在函数式程序设计中不会出的错误，即与操作的时间有关的错误（因为操作的效果依赖于历史）

- 重新看前面的迭代式求阶乘过程（函数式程序）：

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product) (+ counter 1))))
  (iter 1 1))
```

- 用命令式技术写这个程序，可以通过赋值直接修改 product 和 counter 的值，不需要通过参数传递

命令式编程的缺陷

- 按命令式方式写出的程序：

```
(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                 (set! counter (+ counter 1))
                 (iter))))
    (iter)))
```

- 上面过程里有两个赋值，调换其顺序后程序还正确吗？

```
(set! counter (+ counter 1))

(set! product (* counter product))
```

- 显然不是！在命令式程序设计里，赋值顺序非常重要，而函数式程序设计没有这种问题。后面还会看到命令式程序设计的更多问题

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)
```

- 这个过程返回一个具有局部状态的对象（也是一个过程）

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

C语言：带局部状态的过程？

- 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静态变量。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;
}
```

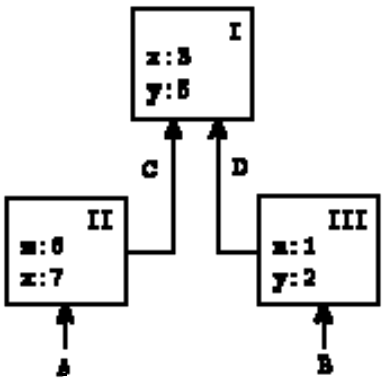
- 只能定义单一的包含简单状态的对象，定义包含复杂状态的对象或者对象生成器需要更复杂的结构和使用规则
- 更一般情况，通常需要通过存储管理和数据结构技术

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用到变量的值
  - 为能找到变量的值，需要在某个地方记录它
  - 记录变量约束值的结构称为“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
  - 即使求值 **(+ 1 1)**，也需要环境为 **+** 提供意义
- 代换模型（回忆）：
  - 整数和实数的值直接取得，变量代换为约束值
  - 组合式求值将过程作用于一组参数，先求值所有参数，而后
    - 对基本过程，直接得到它作用于实参的结果
    - 对复合过程，用实参代换过程体的形参后求值得到的过程体

环境和求值

- 有了赋值后，代换模型就失效了
  - 变量已不再是代表值的简单名字，而表示某种“存储位置”
  - 有关位置保存着值，而保存的值可以随计算进展而改变
  - 可行的求值模型必须能反映变量值的变化
- 处理赋值的求值模型需要有存储的概念。下面的新模型称为环境模型。几个概念：
- 环境：框架（**frame**）的链接结构
  - 框架：可空的表格，项表示变量约束。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（外围框架指针），全局框架在最上层，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其约束的框架里的约束值



例：x在环境A中的值，在环境 B中的值

求值的环境模型

- 实际上，前面代换模型也需要环境的支持
  - 由 **define** 定义的变量，其定义值需要保存到环境里
  - 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
- 为描述解释器的意义，我们假定有一个全局环境
  - 它只有一个全局框架，其中包含所有基本过程名的意义约束
  - 没执行任何程序之前，系统的当前环境就是全局环境
  - 随着程序的执行，当前环境不断变化
    - 可能在已有框架（包括全局框架）中增加新约束
    - 可能修改某个（某些）已有框架里的约束
    - 可能增加新的框架作为当前框架，原来的当前框架可能变成新框架的外围框架

环境模型下的求值

- 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
  - 赋值 **set!** 的执行可能改变当前环境里已有的约束，**define** 的执行可能导致环境中增加新的约束
  - 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最重要最需要关注的事项
- 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：
  - 代码是 **lambda** 表达式的体和参数
  - 环境指针指向求值该 **lambda** 表达式时应该使用的环境
- 下面通过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

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)))

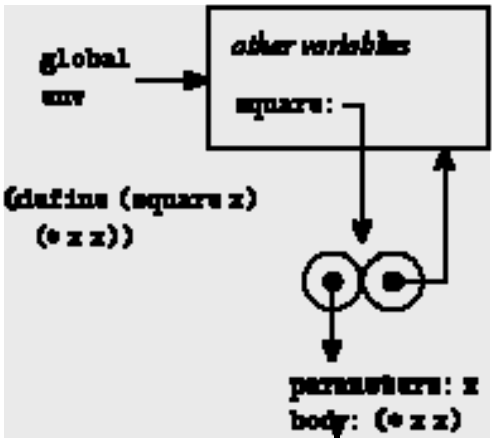
- 求值效果是在全局环境增加了 **square** 的约束，它约束于新建的过程对象

- 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

  其代码部分包括参数和过程体

  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



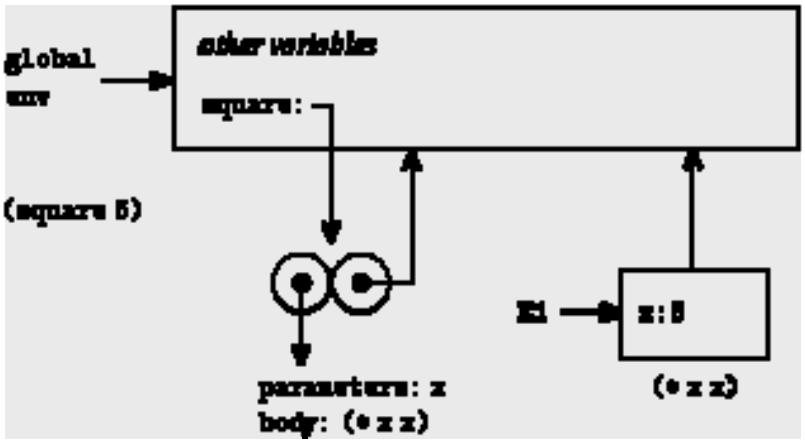
- 方框表示框架，其中是一些变量-值约束
- 两圆圈表示 **lambda** 表达式求值建立的过程对象

过程应用

- 在全局环境中求值组合表达式时，第一个参数的值是一个过程对象
  1. 建立新环境
     - 用过程对象的参数表和实参创建一个新框架
     - 建立以新框架为当前框架，以环境指针为外围框架的新环境
  2. 在新环境里求值过程体

求值 **(square 5)**：

- 求值时先创建新环境 **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

- 上述规则是一般的：
  - 规则中的外围环境通过环境指针确定（未必是全局框架）
  - 建立过程对象时确定了环境，建立新环境的规则都一样

求值的两条基本规则：

- 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** 表达式的体和参数
  - 其环境指针指向 **E**
- 将一个过程对象应用于一组实参的过程：
  1. 构造一个新框架，其中存入过程的形参与对应实参的约束，该框架以过程对象的环境指针作为外围框架，形成一个新环境
  2. 在这个新环境中求值过程体

define 和 set!

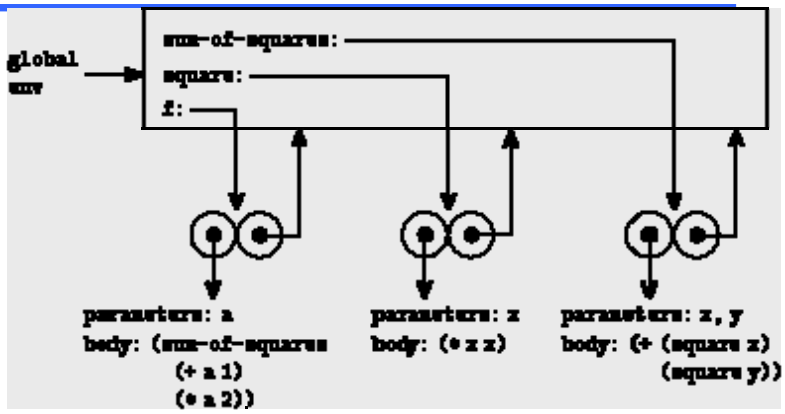
- 现在详细说明 **define** 和 **set!** 的不同
- **define** 的作用是在当前环境的当前框架里定义一个符号：
  - 在当前框架里建立一个约束，将被定义符号约束到给定值
  - 如果当前框架已有这个符号，则改变其约束（注意书上的注释）
- **(set! <*变量*> <*value*>)** 的作用：
  - 在当前环境里查找 **<*变量*>** 的约束。如果当前框架里有，约束就确定了；否则到外围框架去找。查找可以沿外围环境指针前进多步
  - 把找到的约束中变量的约束值修改为由 **<*value*>** 算出的值
  - 如果环境中没有**<*变量*>** 的约束（查找到达全局框架但仍未找到），就报告变量无定义错误
- 新求值规则比代换模型复杂很多。它表现了 **Scheme** 解释器工作方式，可以根据它实现 **Scheme** 解释器（第**4**章）
- 下面看几个例子

简单过程的应用

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)))
```

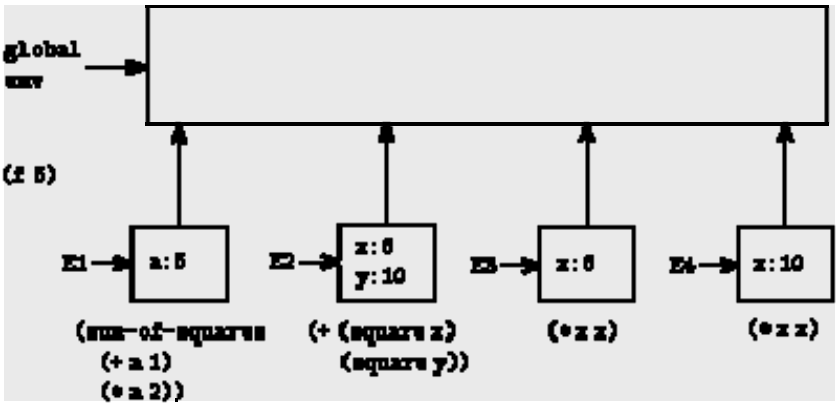
三个定义建立起的环境见图



对 (f 5) 的求值

求值时新建一个环境，其中有一个新约束

- 每个调用建立新框架，同一函数的不同调用各建立框架，相互无关
- 这里没有特别关注返回值的传递问题



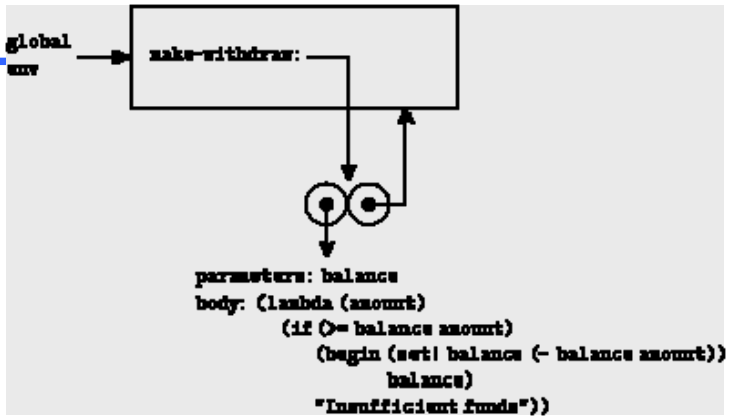
框架和局部状态

有局部状态的对象计算的情况

提款处理器代码：

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

求值该定义使 make-withdrew 约束于创建的过程对象

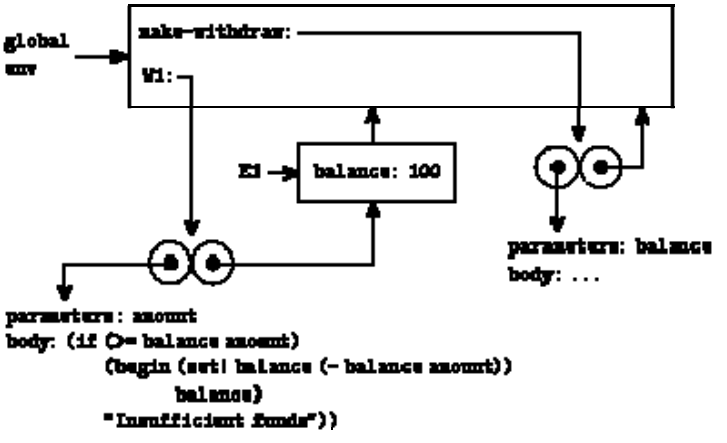


调用

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

建立环境 E1，在其中求值过程体

求值 lambda 表达式建立一个过程对象（左），其环境指针指向 E1，W1 约束于这个过程对象



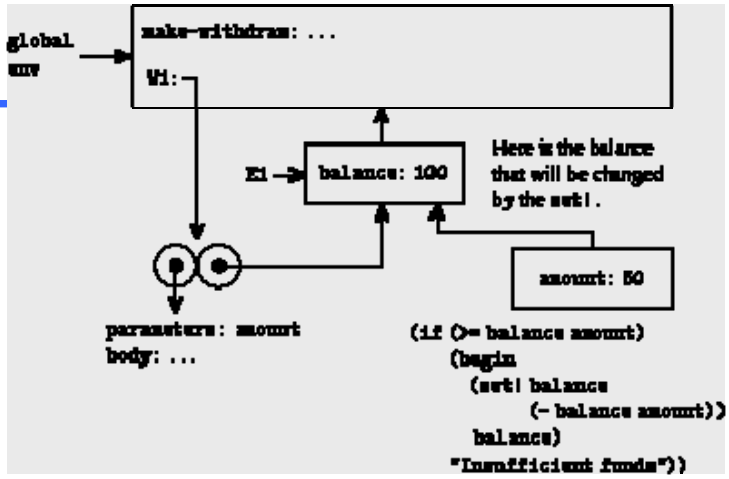
框架和局部状态

现在求值组合式（过程调用）：

(w1 50)

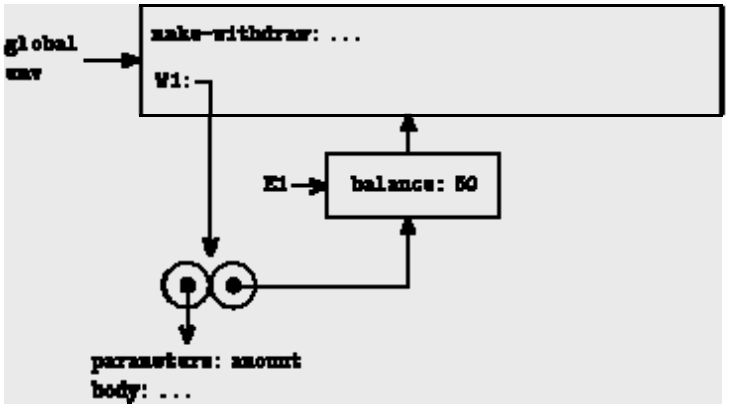
调用建立起新环境（右）并在其中求值

从环境框架里找变量的值



set! 表达式求值改变环境中 balance 约束，其值变为 50

- 再次调用 **W1** 将建立一个新框架，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

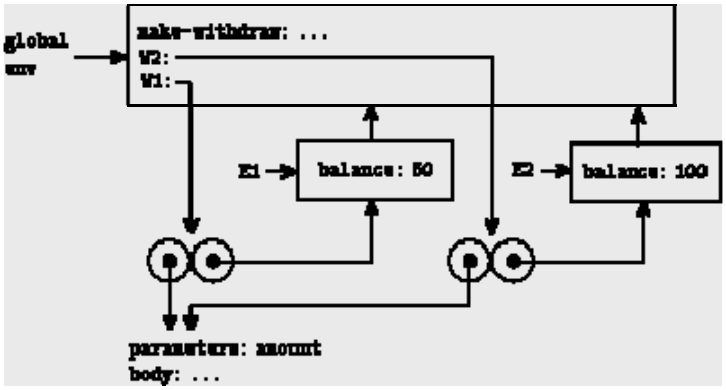


框架和局部状态

- 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



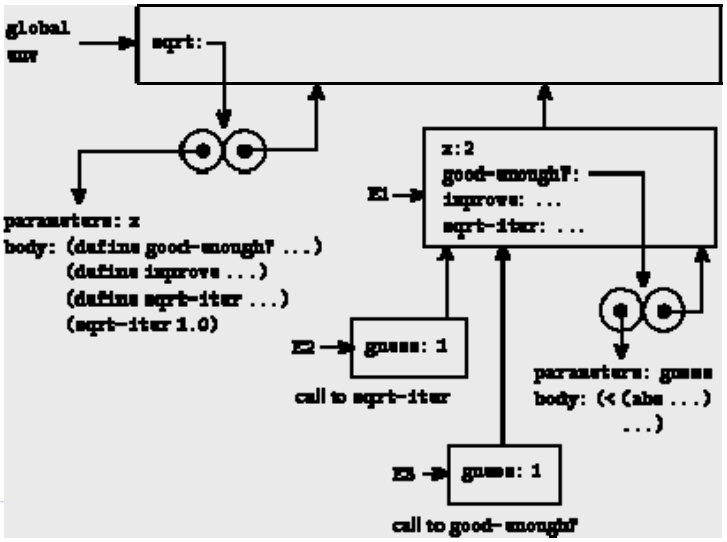
- 两个提款处理器是两个过程对象，各自独立变化
- 两个提款处理器（过程对象）的代码完全相同
  - 两者共享同一代码，还是各有一份代码，是系统的实现细节
  - 不同实现方式不影响程序语义，但可能影响资源消耗和效率
- 聪明的编译器可能让它们共享代码，以提高内存利用的效率

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) 建立框架 E1，其中有形参 x 的约束和内部过程约束（右图只表示了一个过程）
- 内部过程名约束到过程对象（代码和一个环境指针）。它们的环境指针都指向 E1
- 首次调用 good-enough? 时的现场情况如右图



内部定义

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
- 应注意过程对象的环境指针的作用，它决定所建新环境的结构，即，决定新的当前框架的外围框架是哪个框架

set! 和 define 的不同（总结）

- 在执行 **set!** 时
  1. 找到被赋值变量在当前环境中的约束
  2. 修改这个约束中的约束值
  - 查找变量的过程从当前框架开始
  - 如果在一个框架里没找到要找的变量，就到其外围框架里去找
  - 如果这一查找到达全局框架仍未找到变量的约束，报错
- 执行 **define** 时
  - 只考虑当前框架，不考虑当前环境中的其他框架
  - 在当前框架里为被定义变量建立约束
    - 如果当前框架里不存在这个变量，就建立新变量约束
    - 如果当前框架里已有该变量，就改变其约束（有些 **Scheme** 实现在这种情况下报错或发出警告）

总结

- 变动和赋值（**set!**），是模拟复杂系统的有力手段

  也导致计算的代换模型失效，需要更复杂的模型来解释计算过程
- 用有局部状态的过程实现具有局部状态变量的对象
- 引入状态和状态变化带来的效益和问题
  - 模块化手段，变化的封装
  - 同一和变化，引用透明性的丧失
  - 操作和时间
- 求值的环境模型
  - **lambda** 表达式的求值建立新过程对象
  - 调用过程时需要创建新框架
  - 注意 **set!** 和 **define** 的意义，局部定义等