函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

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

第七讲

用对象模拟真实世界

- 纯的函数式程序设计方法没有赋值语句，没有“状态”之说，变量只是数值的代号，其值不会更改，也不需要维持其存储空间。函数只要参数相同，执行结果就相同。带来的好处是函数没有副作用，易于调试，易于并行。

- 真实的世界是有“状态”的概念的。每个事物都处于变化之中，都有其“状态”。状态都可以改变。因此，允许赋值语句的程序设计语言，更容易模拟真实的世界。

scheme的赋值语句

```
(define balance 100)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance);begin是特殊形式，逐个求值参数，返回最后一个参数的值
      "Insufficient funds"))
```

(withdraw 25)
75
(withdraw 25)
50
(withdraw 60)
"Insufficient funds"
(withdraw 15)
35

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

scheme的多表达式过程

```
(define (f x)
    (display "in f:") (display x) (newline)
    (+ x 1)
    (* x 2))
```

f被调用时，依次对函数体中多个表达式求值，返回最后一个表达式的值

```
(f 5)
=>
in f:5
10
```

scheme的多表达式过程

等价写法：

```
(define f
  (lambda (x)
   (display "in f:") (display x) (newline)
   (+ x 1)
   (* x 2)))
```

(lambda (x …) <exp1> … <expn>)

语义：顺序求值**<expi>**，以最后一个表达式的值作为值

```
(f 5)
=>
in f:5
10
```

如果表达式都是没有副作用的，且不存在赋值语句，则完全没必要写多个表达式。因为除了最后一个表达式外，前面的表达式都没用。有了赋值语句，则前面表达式的执行情况有可能会影响到最后一个表达式的执行结果。

scheme的局部变量

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


(withdraw 40)
60
(withdraw 40)
60
(withdraw 40)
60
;每次执行withdraw都要执行 (define balance 100)，因其为函数体的一部分
```

scheme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 40)
60
(new-withdraw 40)
20
(new-withdraw 40)
"Insufficient funds"
;new-withdraw是一个lambda表达式，该表达式中，balance的值只初始化一次

scheme的局部变量

```
(define (make-withdraw balance);形参balance是局部变量
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

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

创建"账号对象"(闭包)

```
(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);接受消息
          ((eq? m 'deposit) deposit) ;接受消息
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

创建"银行账号对象"(闭包)

```
(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);接受消息
          ((eq? m 'deposit) deposit) ;接受消息
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

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

引进赋值的好处

- 避免函数传递大量参数，提高编程效率和运行效率

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数（随机的）

x2 = (rand-update x1)

x3 = (rand-update x2)

反复做可得到一个随机整数序列

实例：随机数生成器

●可定义一个带局部状态和赋值语句的过程**rand**，实现一个随机数生成器：

```
(define rand
  (let ((x random-init)) ;random-init取某个整数作为初始值
    (lambda ()
      (set! x (rand-update x))
      x)))
```

● 也可以直接使用函数**rand-update** 生成随机数序列

使用形式***x2 = (rand-update x1) x3 = (rand-update x2) .....***

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

实例：蒙特卡洛法求π

- 定理：两个随机生成的整数无公因子的概率是$6/\pi^2$

使用赋值的做法:

```
(define (estimate-pi trials);考察trial对整数
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))
(define (cesaro-test) ;随机生成两个整数，看公因数是否是1
   (= (gcd (rand) (rand)) 1))
(define (monte-carlo trials experiment);experiment描述条件
  (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))
```

实例：蒙特卡洛法求π

```
(define (estimate-pi trials) ;不使用赋值的纯函数式做法:
  (sqrt (/ 6 (random-gcd-test trials random-init))));random-init在外定义
(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))
```

实例：蒙特卡洛法求π

- 纯函数式实现方法的劣势：

1) 需要额外的参数
2) 测试的条件写死在代码里，不容易替换。生成随机数和使用随机数的过程没有分开处理，而是交织在一块，代码耦合程度高，模块化程度低。不好。
3) 需要更多随机数来做测试，则维护这些随机数麻烦。

引入赋值带来的坏处

- 无法做到以同样参数调用同一过程总得到同样结果(不利于程序测试，并行)
- 代换模型失效，程序失去引用透明性，导致难以用机器分析程序的行为。
- 多条赋值语句的执行顺序需要仔细考虑。

引入赋值带来的坏处

- 无法做到以同样参数调用同一过程总得到同样结果(不利于程序测试，并行)
  对比:

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

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))

(define D (make-decrementer 25))
(D 20)
5
(D 20)
5
```

引入赋值带来的坏处

- 代换模型失效，程序失去引用透明性，导致难以用机器分析程序的行为。

  对比：

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

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))

(define D (make-decrementer 25))
(D 20)
5
(D 20)
5
```

赋值使得代换模型失效

代换模型:

变量名只是值的代号，不管出现在哪里，都可以同时用同样的值替换。过程名也是用值(类似于过程的地址)替换。反复用值替换名字以及求值，最终求得表达式的结果。

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))

(define D (make-decrementer 25))
(D 20)
5
(D 20)
5
```

经过代换：

```
(lambda (amount)
   (- balance amount)))   =>

(lambda (amount)
   (- 25 amount)))   =>

(- 25  20)  =>  5
```

赋值使得语言失去引用透明性

●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。

●引入赋值后，难以判断什么是“同样的东西”，所以语言失去了引用透明性。难以确定能否通过貌似等价的表达式替换来简化表达式，程序变得难以用机器分析。

```
(define D1 (make-decrementer 25))
(define D2 (make-decrementer 25))

(D1 20)
5
(D2 20)
5
```

D1和D2看上去应该是“同一个”事实上其行为也总是相同的。

赋值使得语言失去引用透明性

●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。

●引入赋值后，难以判断什么是“同样的东西”，所以语言失去了引用透明性。难以确定能否通过貌似等价的表达式替换来简化表达式，程序变得难以用机器分析。

```
(define D1 (make-decrementer 25))
(define D2 (make-decrementer 25))

(D1 20)
5
(D2 20)
5
```

D1和D2看上去应该是“同样的东西”事实上其行为也总是相同的。

```
(define W1 (make-simplified-withdraw 25))
(define W2 (make-simplified-withdraw 25))
(W1  20)
5
(W1  20)
-15
(W2  20)
5
```

W1和W2从形式上看理应是“同样的东西”事实上其行为却不同，说明不是“同一个”

赋值使得语言失去引用透明性

- 总之，事物如果会发生变化，那么判断“同一性”就会变得困难。

今天的你和昨天的你，是同一个你吗 ---- 你的身份证号和名字没变

今天的你和昨天的你，是同一个你吗 ---- 今天你改字名了

同一和变化

假定**Paul** 和**Peter** 有各自银行账户，其中各自有**100** 块钱。下面是这一事实的两种模拟。

第一种：

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

第二种：

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

表面上看都一样，实际上第一种正确，第二种错误。但如果**peter**和**paul**的账户都不允许改变，则两种模拟方式效果无差别。即如果对象内容不可变，则引用两个内容相同的对象，和引用同一个对象，没有区别。

引入赋值带来的坏处

●基于赋值的程序设计是“命令式程序设计”。命令式程序设计需要仔细考虑语句的执行顺序需要仔细考虑。即容易犯与执行顺序，时间，历史有关的错误。纯函数式程序设计很少有这个问题。

函数式求阶乘：

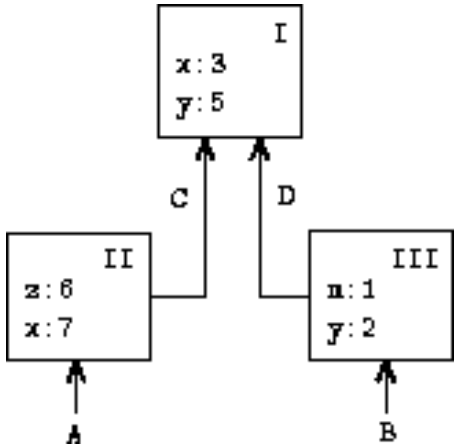
```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

命令式求阶乘：

```
(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))) ;需要注意两条set语句的顺序
```

求值的环境模型

- 在对表达式求值时，用值替换变量的时候，需要找到变量对应的值在哪里
- 变量的值可修改，则需要为它分配存储空间
- 记录变量及其值的结构称为“环境”。
- 表达式需要在具体的“环境”中，才能进行求值。没有环境，就找不到表达式中变量对应的值，也找不到表达式中过程名所对应的函数体在哪里（即使求值(+ 1 1)，也需要环境为 + 提供意义）。
- 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。在该环境里找不到约束值，就到其外围环境继续寻找。



在环境中求值

如果要对一个组合表达式求值:

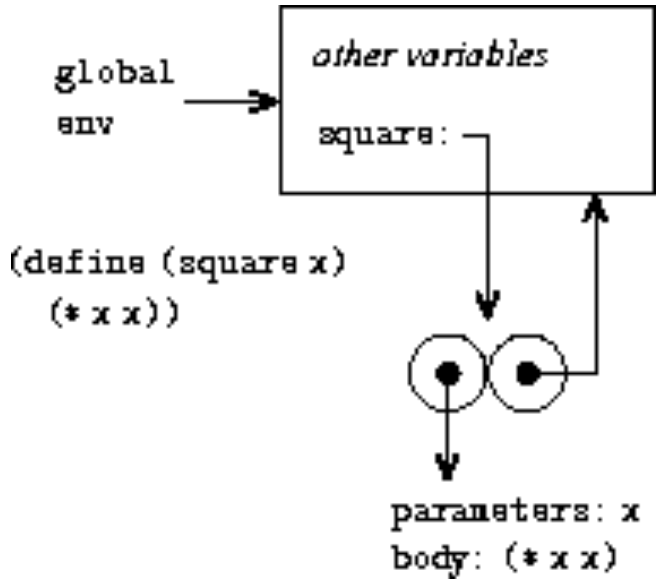
1) 求值这一组合表达式中的各个子表达式(程序员不应依赖求值顺序)
2) 将运算符子表达式的值运用于运算对象子表达式的值

在环境中求值

3) 在求值环境中，一个过程总是一个“序对”（第四章里过程实现为一个列表），其car是过程参数和过程体代码，cdr是指向该过程求值的环境的指针 。过程序对（过程对象）只能通过对lambda表达式的define求值来创建。过程体代码来自于lambda表达式的过程体，cdr指向的环境，就是对该lambda表达式的define进行求值，产生出这个过程对象的环境。

(define (square x) (* x x)) 等价于：
(define square (lambda (x) (* x x))

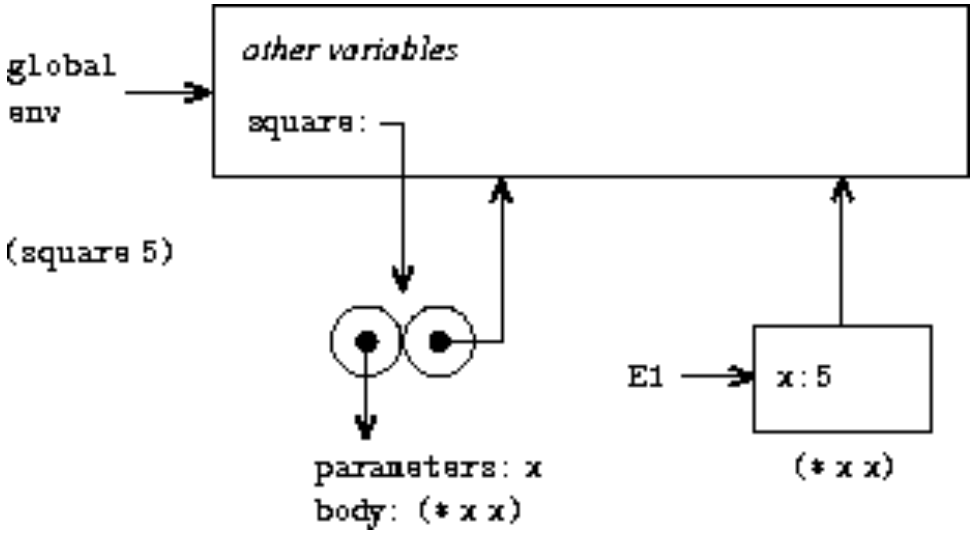
对其求值的结果:



在环境中求值

4) 将一个过程应用于一组实际参数时，将会建立一个新环境，其中包含了将所有形式参数约束于对应实际参数的框架。该环境的外围环境就是包含过程定义的那个环境，其后就在新环境中求值过程的体。

在全局环境里对(square 5)求值的结果：



在环境中求值

5）对 define 表达式求值，会在当前环境中新增一个变量的约束
6）对 set!表达式求值，要找到变量在当前环境中的约束的位置，然后修改这个位置上的值。找不到则到外围再外围找。一直到全局环境都找不到，则报错。

“环境”的结构详解

●“环境” 是一个列表，列表中的每个元素都是一个“框架”。也可以认为环境是一个序对，car是一个“框架”，cdr是指向上一层(外围)“环境”的指针。最上层(最外围的)的“环境”的cdr是空指针。

●"框架"(frame) 是一个列表，其car是一个列表，该列表中的每一项都是一个变量名（过程名也可以看做是个变量）。cdr里的项和 car里的变量依次一一对应，是变量的值。car里的变量不能重名。

●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。在该环境里找不到约束值，就到其外围环境继续寻找。

●程序开始运行时，有一个"顶层环境"(后文称为glb-env)，其中包含基本过程，比如 + ，-， cdr， car， map ....等的约束（变量及其值的对应关系）。

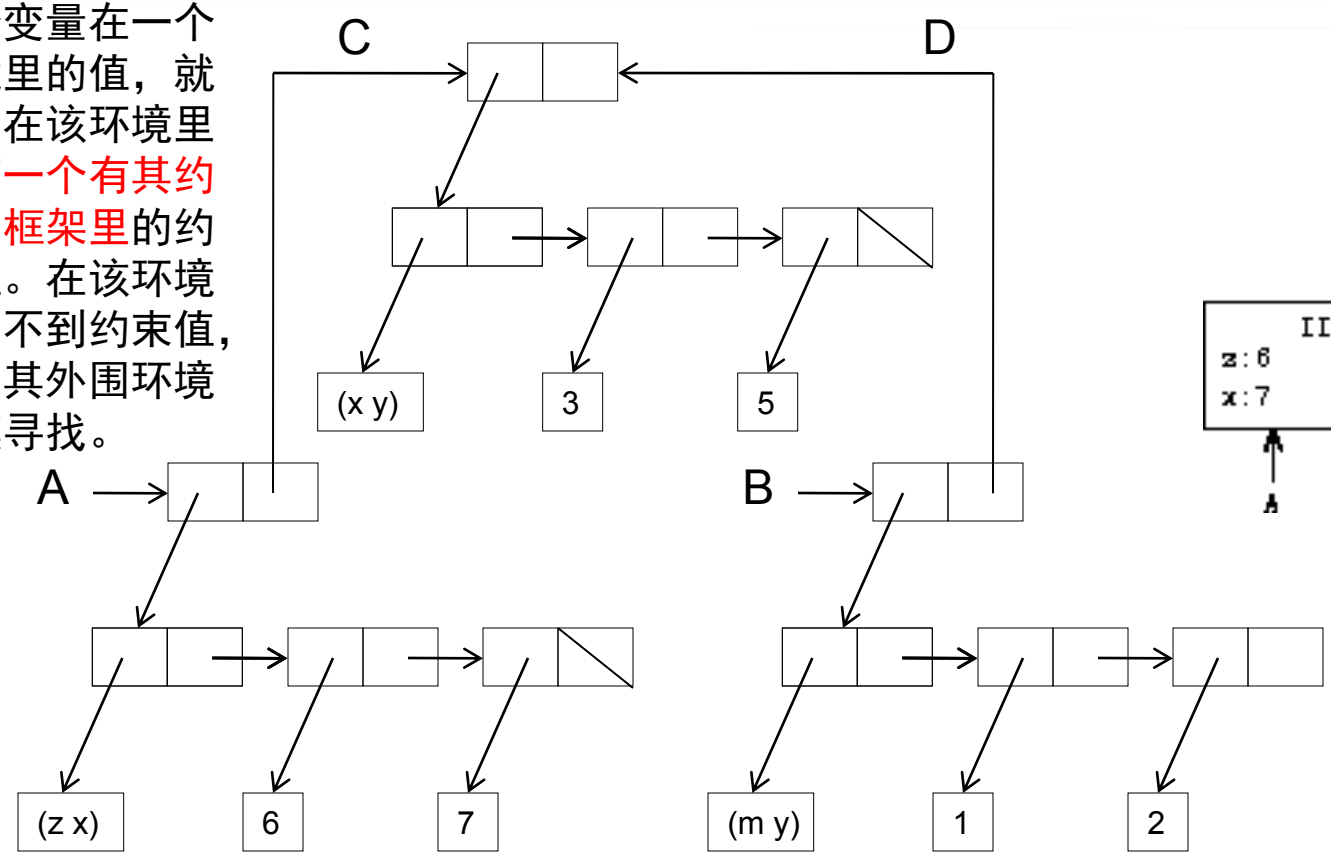
“环境”的结构详解

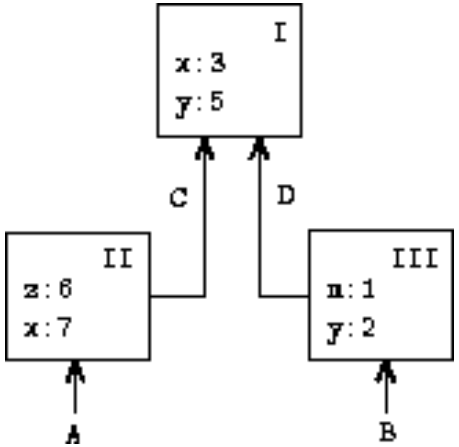
随着程序的运行：

- 可能在已有环境（包括全局环境）中增加新约束(对 define 求值时)
- 可能修改某个（某些）已有环境里的约束（对set!求值时）
- 可能增加新的环境作为当前环境，原来的当前环境变成新环境的外围环境(执行函数调用时)

“环境”的结构详解

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。在该环境里找不到约束值，就到其外围环境继续寻找。



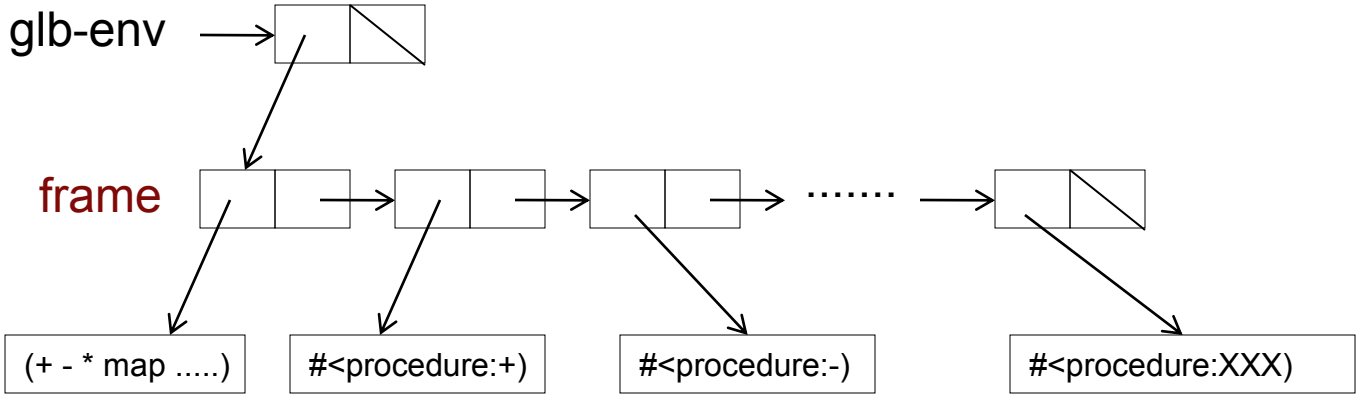


三个环境：
A, B, C(D和C相同)
C是A,B的外层环境

“环境”的结构详解

程序开始运行时的 glb-env：

((+ - * map .....) #<procedure:+> #<procedure:-> #<procedure:*>
#<procedure:map> .....)) ;红色括号内部为 一个 frame



“环境”的结构详解

```
(define a 5)
(define b 4)
(define (square x) (* x x))
(define c 3)
```

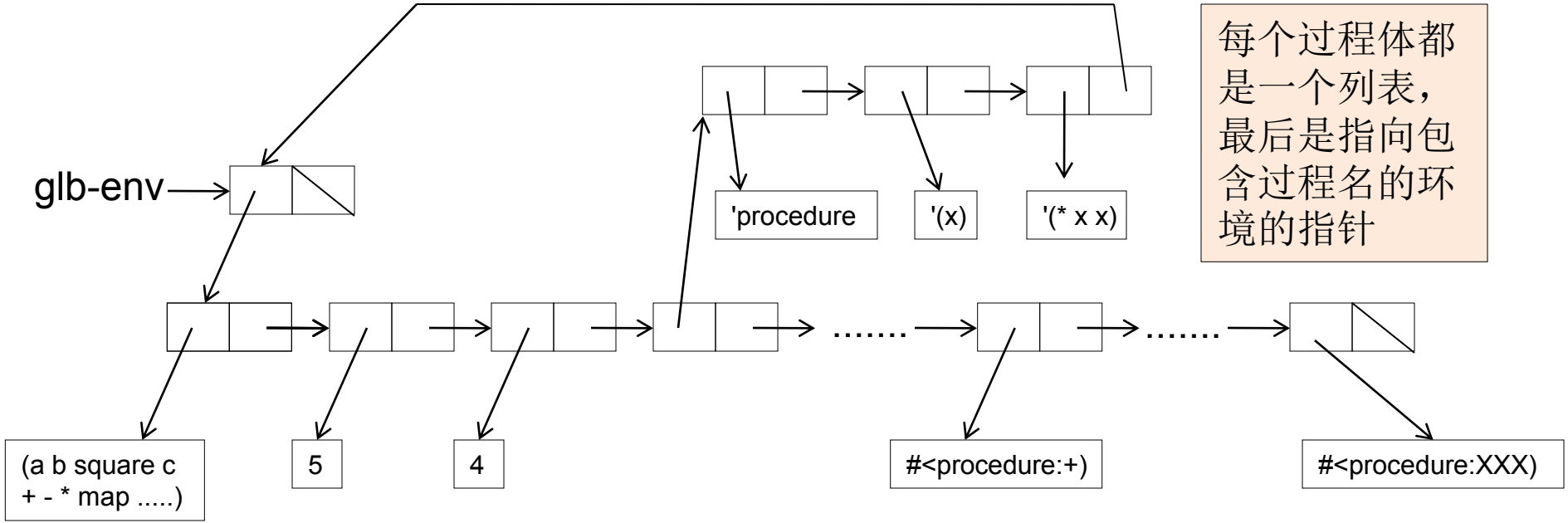
对以上语句求值后glb-env为：

```
((a b square c + - * map .....)   5  4  '(procedure (x) (* x x) glb-env)  3
     #<procedure:+>    #<procedure:->   #<procedure:*>    #<procedure:map>  .....))
```

“环境”的结构详解

glb-env：

((a b square c + - * map .....) 5 4 '(procedure (x) (* x x) glb-env) 3
    #<procedure:+> #<procedure:-> #<procedure:*> #<procedure:map> .....))

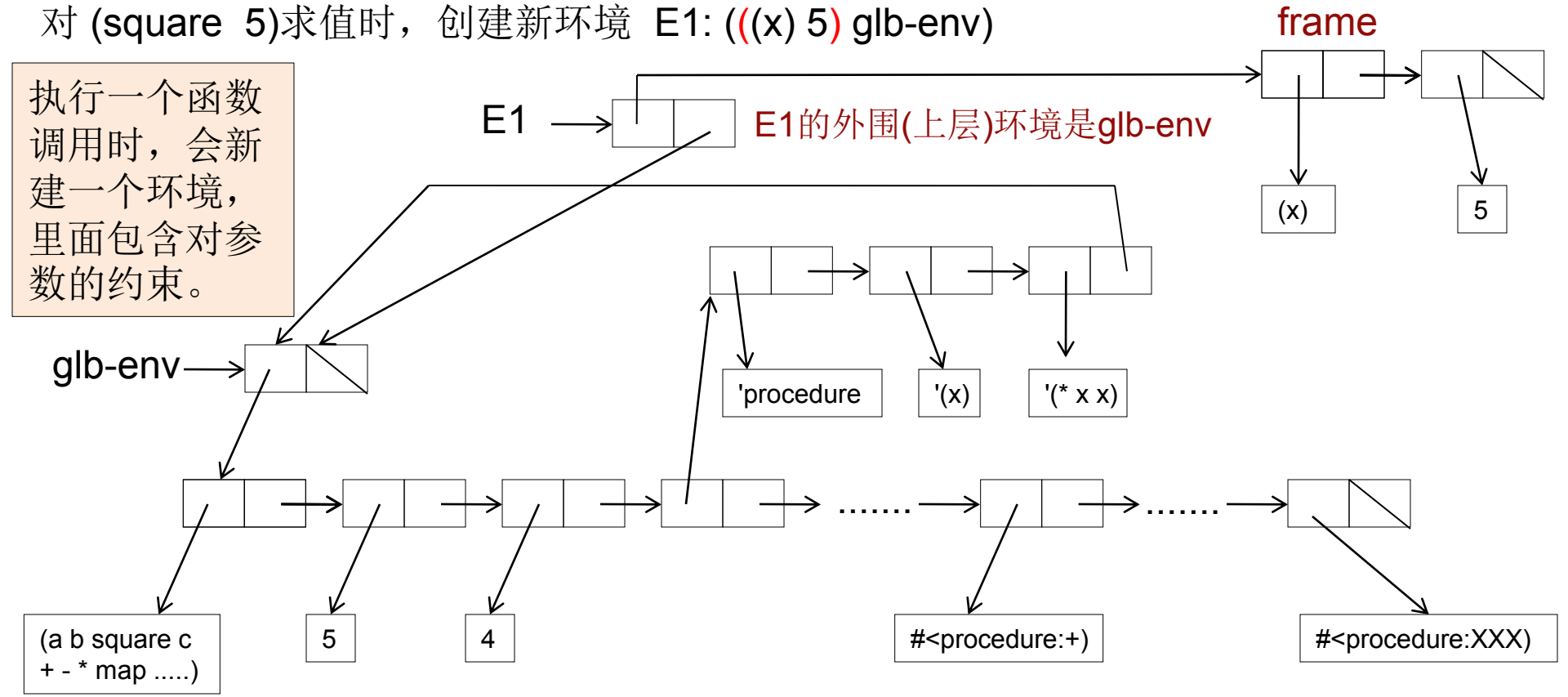


每个过程体都是一个列表，最后是指向包含过程名的环境的指针

“环境”的结构详解

对 (square 5)求值时，创建新环境 E1: (((x) 5) glb-env)

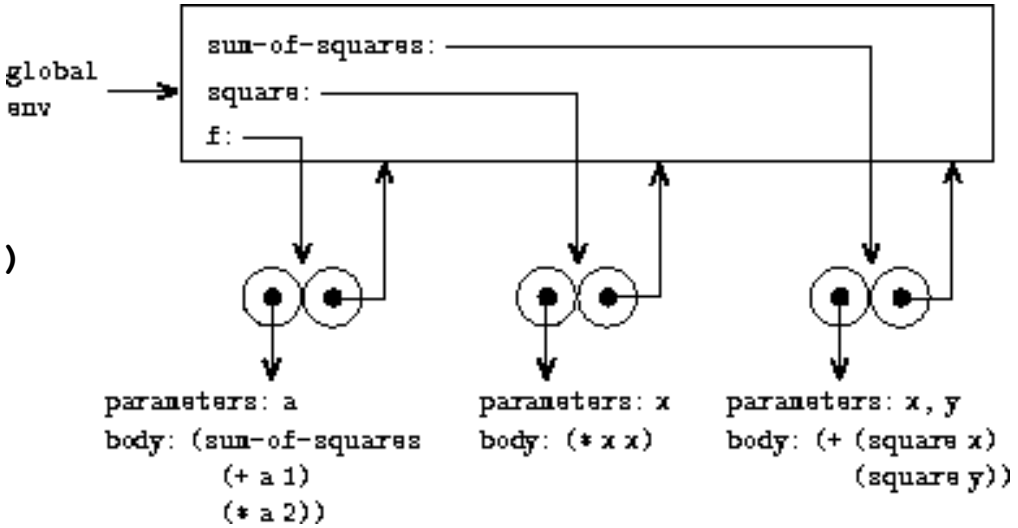
执行一个函数调用时，会新建一个环境，里面包含对参数的约束。



简单过程的应用

对以下程序进行求值后的环境：

```
(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)))
```



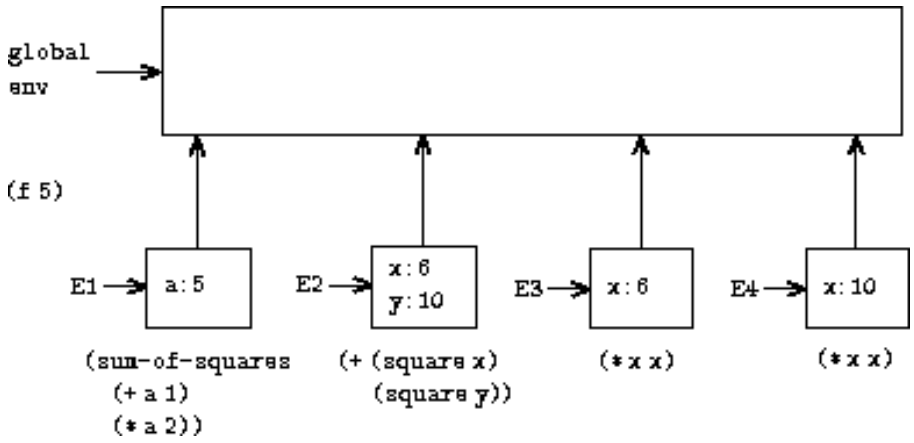
简单过程的应用

```
(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)))
```

square的函数体每次被执行都会创建一个新的环境，各个环境中的参数 x的值不相关。

函数调用结束后环境的回收，是解释器需要处理的问题。

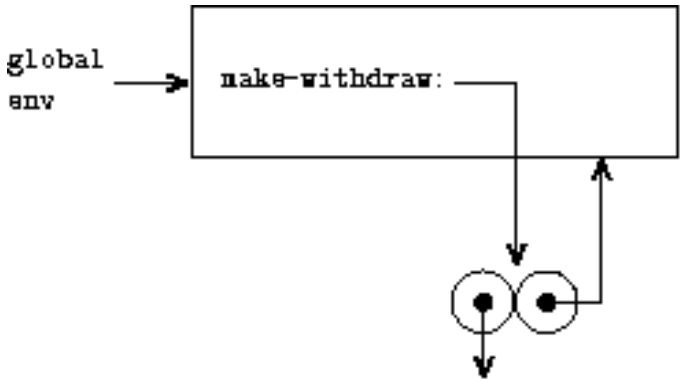
对(f 5)进行求值时的环境：



框架和局部状态

对：

```
(define (make-withdraw balance);形参balance是局部变量
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```




```
parameters: balance
body: (lambda (amount)
        (if (>= balance amount)
            (begin (set! balance (- balance amount))
                   balance)
            "Insufficient funds"))
```

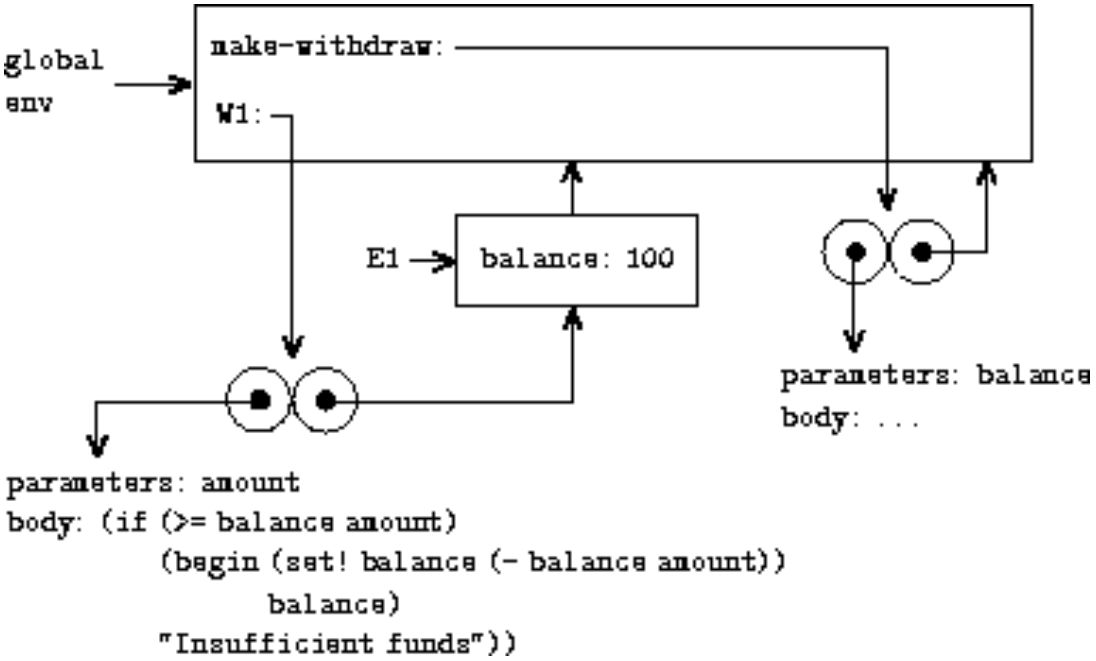
求值：

框架和局部状态

对：

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

求值



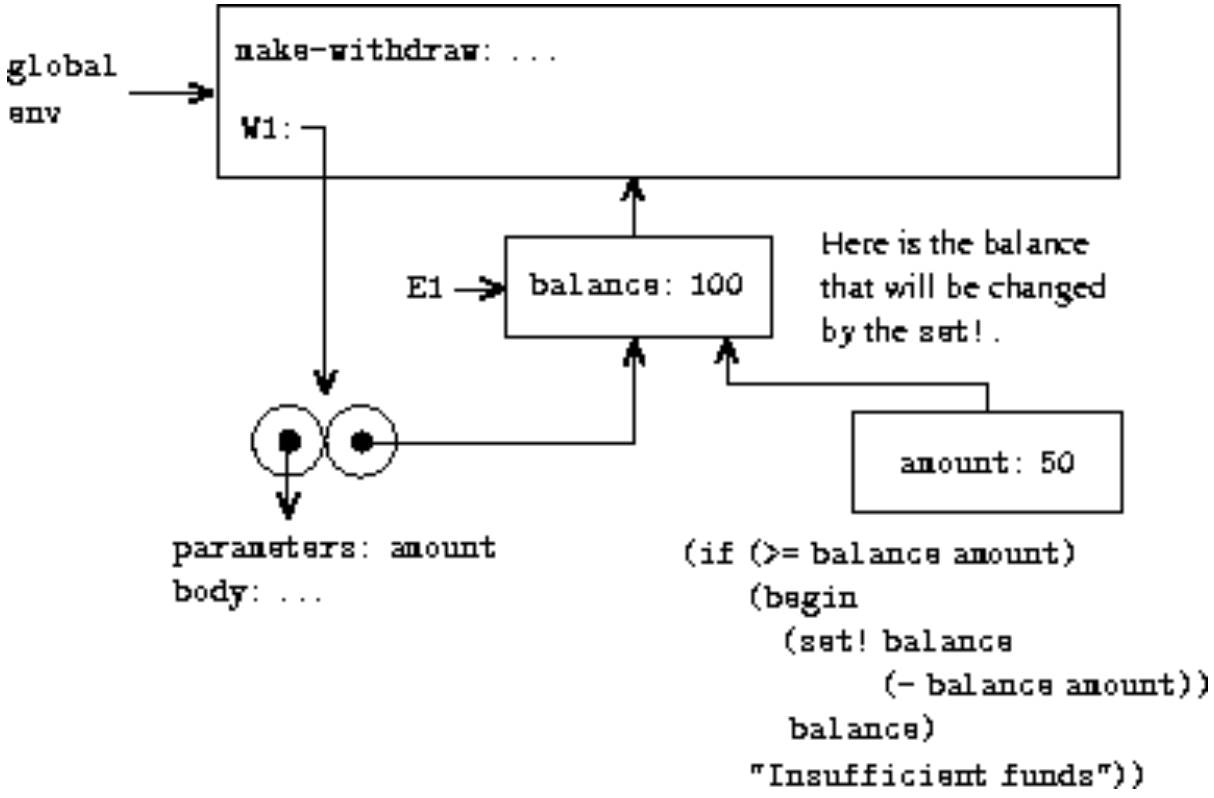
框架和局部状态

对：

```
(W1 50)
```

求值：

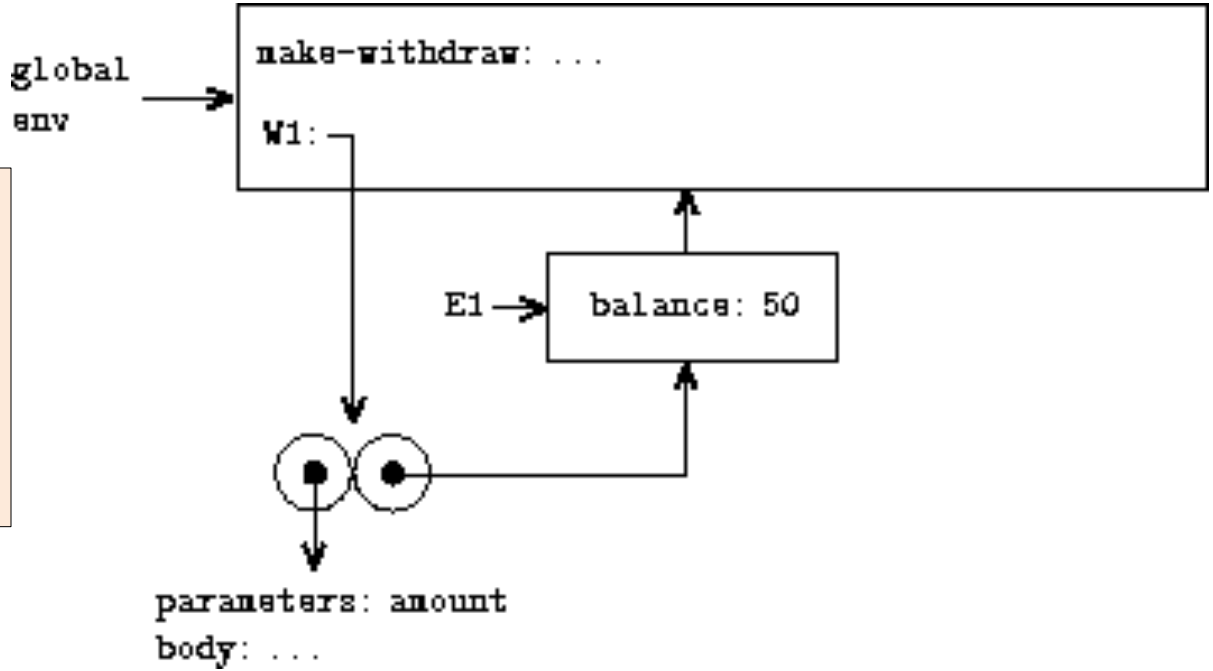
```
(define (make-withdraw balance
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance
                 (- balance amount))
               balance)
        "Insufficient funds"))))
```



框架和局部状态

(W1 50)
求值完成后

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



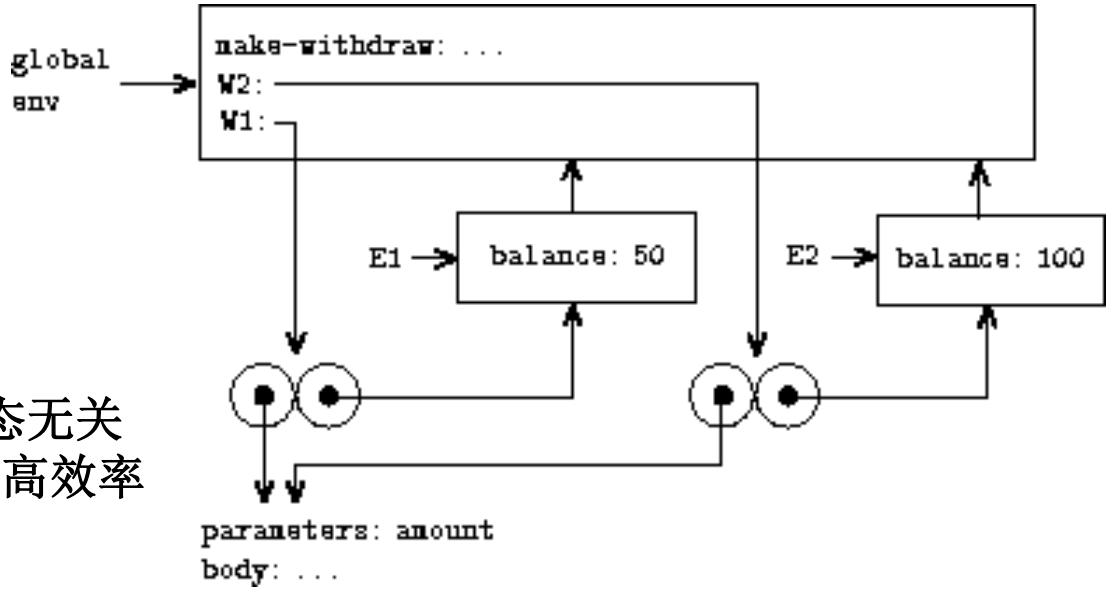
框架和局部状态

再对

(define W2 (make-withdraw 100))

求值：

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



W2 的局部状态与**W1** 的局部状态无关

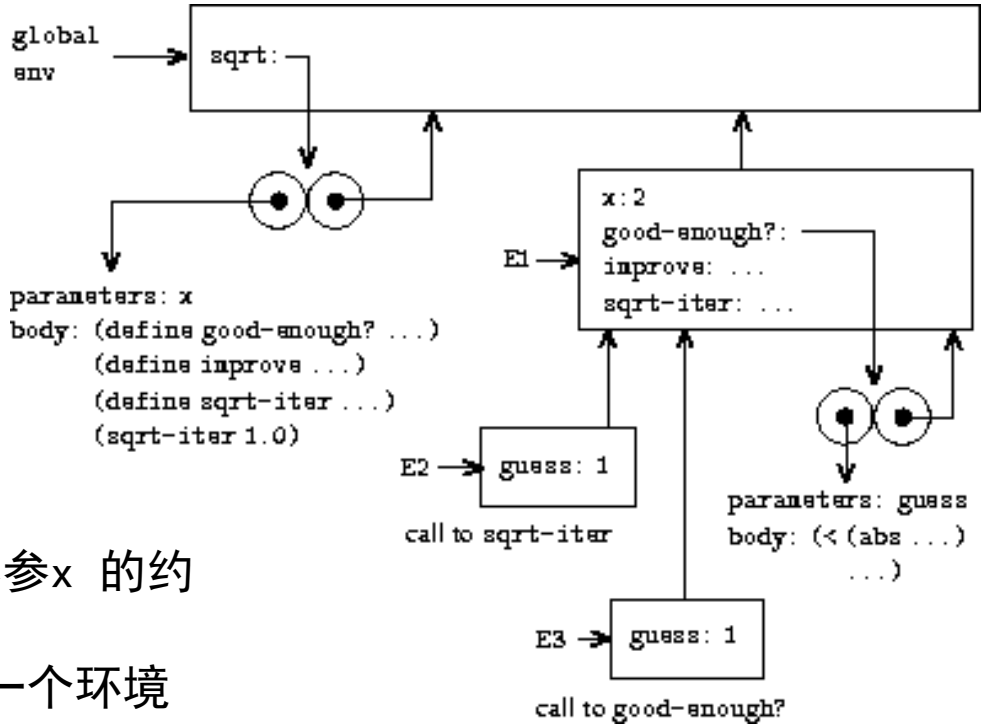
W1和**W2**应共享一份代码，以提高效率

内部定义

以下过程有内部定义：

```
(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))
```

对 (sqrt 2)求值，首次调用good-enought?时的环境:



●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

●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。

内部定义

- 每次过程A被调用时，都会新建一个环境，过程的内部定义都会被重新求值（会重新建立内部过程对象）。这些内部定义的变量或者过程，都会和过程的形式参数一样，被约束在这个新建的环境之中。 该新建的环境，其外部环境就是直接包含A的约束的那个环境。

- 因此多个过程中的内部过程，可以重名，互相不影响。因为这些内部过程被调用时，求值用的环境不同。