[读书笔记]SICP：10[B]求值的环境模型

本章对应于书中的3.2。

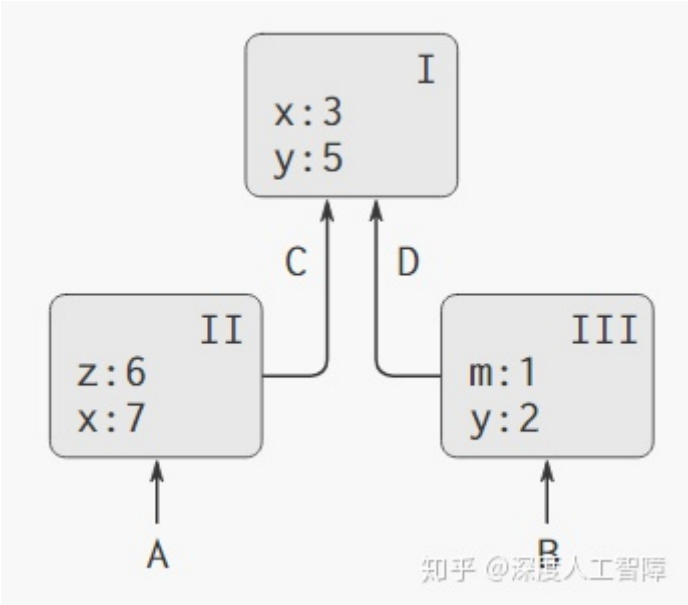
- 函数式程序可以用代换模型进行分析，而引入`set!`赋值后，只能用环境模型进行分析。
- `lambda`表达式是过程对象，对其进行求值时，要得到一个序对，其中一个指向过程体，另一个指向环境
- **创建过程（`define`）：**在当前环境中增加一个变量名，其约束是一个序对，包含过程体，以及指向当前环境的指针。
- **求值过程：**
-
  - 创建一个新的环境
  - 添加形式参数的约束
  - 在外围环境中寻找该变量对应的过程体
  - 对过程体进行求值
  -
    - 如果是`define`，则在当前环境中添加该过程名的约束，其约束是过程体和指向当前新环境的指针
    - 如果是`lambda`表达式，就得到一个序对，其中一个为过程体，另一个是指向当前新环境的指针
    - 如果是对另一个过程的求值，就重复该过程

现在最紧要的是为设计赋值的表达式提供一种计算模型，以便考察在模拟的设计中如何使用具有局部状态的对象。

1 求值的环境模型

在还未引入赋值时，采用**求值的代换模型**定义了将过程应用于实际参数的意义：将一个复合过程应用于一些实际参数，就是用各个实际参数代换过程体中对应的形式参数，然后求值该过程体。

但是引入赋值后，就无法使用该定义了，此时变量已不再是值的名字，而是指向一个保存变量值的“位置”，而在环境模型中，这些位置保存在**环境**结构中。一个环境就是**框架**的序列，每个框架包含一些**约束**的表格以及一个指向该框架的**外围环境指针**（由于环境本身就是框架序列，所以通过外围环境指针将两者的框架序列连接起来了），每个约束就是将变量名关联于对应的值（一个框架中的任意变量只有一个约束）。而一个变量相对于某个环境的值，就是第一个包含该变量约束的框架中的值，如果环境中所有框架都不包含该变量的约束，则该变量在该环境中是无约束的。比如



这里具有三个框架`I`、`II`和`III`，每个框架中包含一些变量的约束，并且框架直接通过外围环境指针连接起来，构成了框架序列，也就是环境。从任意环境指针开始都是环境，比如这里有4个环境`A`、`B`、`C`和`D`。在环境`A`中，变量`x`的值首先在框架`II`中约束，则为7；在环境`B`中，变量`x`的值首先在框架`I`中约束，则为3。

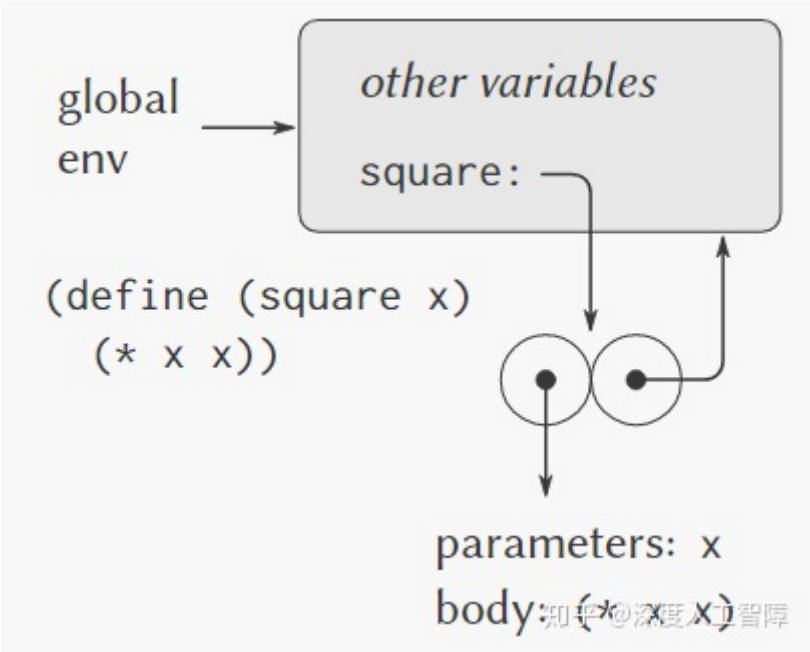
意义：环境确定了表达式求值的上下文，确定了不同变量和符号的约束。这里假设存在一个只包含一个框架的**全局环境**，包含所有关联于基本过程的符号的值。

1.1 创建过程

接下来我们就能用求值的环境模型代替求值的代换模型来对带有赋值的过程进行求值。在求值的环境模型中，过程只能通过求值一个`lambda`表达式进行创建，且由代码和环境组成，其中该过程的**代码**就是`lambda`的正文，而代码的**环境**就是求值该`lambda`表达式产生该过程时的环境。比如在全局环境中对以下代码进行求值

```
 (define (square x)
  (* x x))
; 等价于
(define square
  (lambda (x) (* x x)))
```

- 由于在全局环境中对其进行求值，所以在全局环境中的框架添加一个符号`square`，其值为`(lambda(x)(* x x))`求值的结果。
- 该过程通过对`(lambda(x)(* x x))`进行求值会获得一个序对，其中一个是带有形参`x`的过程体，另一个是指向全局环境的指针



注意：用`define`定义一个符号就是在当前环境中建立一个约束，并赋予该符号指定的值。

1.2 应用过程

创建过程后，环境模型将一个过程应用于一组实际参数时：

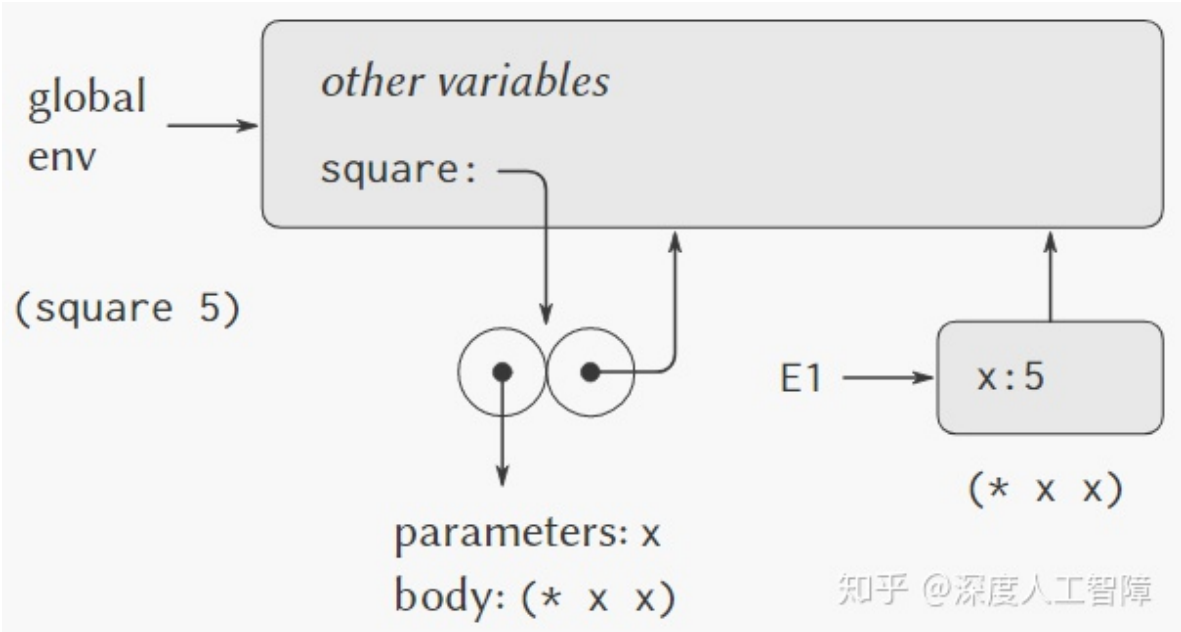
- 创建一个新环境，其中包含所有形参约束于对应的实际参数的框架
- 新框架的外围环境就是该过程的环境（通过该过程指向的环境来确定）
- 在新环境中求值该过程的过程体（通过该过程指向的过程体来确定）

而其求值规则也和之前相同：

- 求值该组合式中的各个子表达式
- 将运算符子表达式的值应用于运算对象子表达式的值

比如这里使用`(square 5)`，它的过程为：

- 创建一个新环境，其中包含只有一个约束`x:5`的框架
- 该新环境有一个指向全局环境的指针
- 在新环境中对`(* x x)`进行求值，则结果为25



这里专门为`set!`创建了环境模型，由此我们就能**解释`(set! )`的求值过程：**

- 首先在环境中确定有关变量``的约束位置

- 修改该约束

接下来介绍几个使用环境模型的实例。

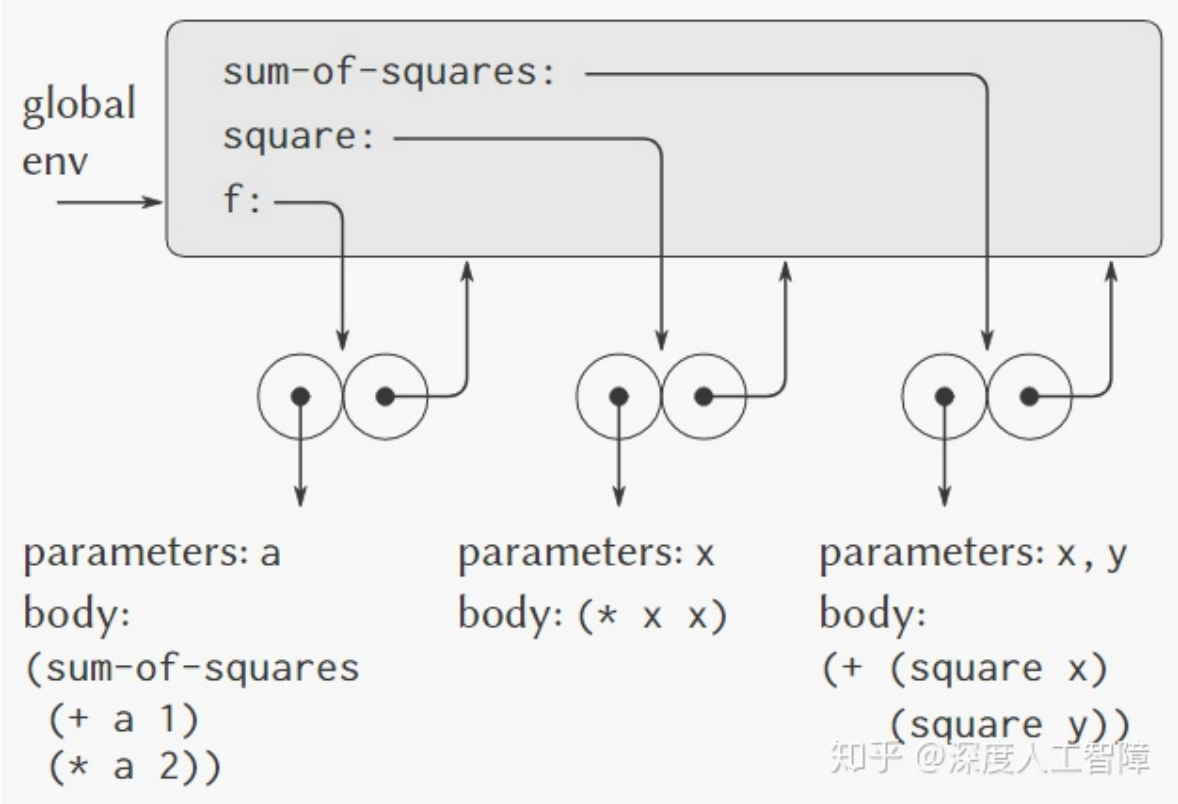
2 实例

2.1 无赋值的过程

我们这里首先以一个不含有赋值的过程为例来介绍环境模型

```
(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)))
```

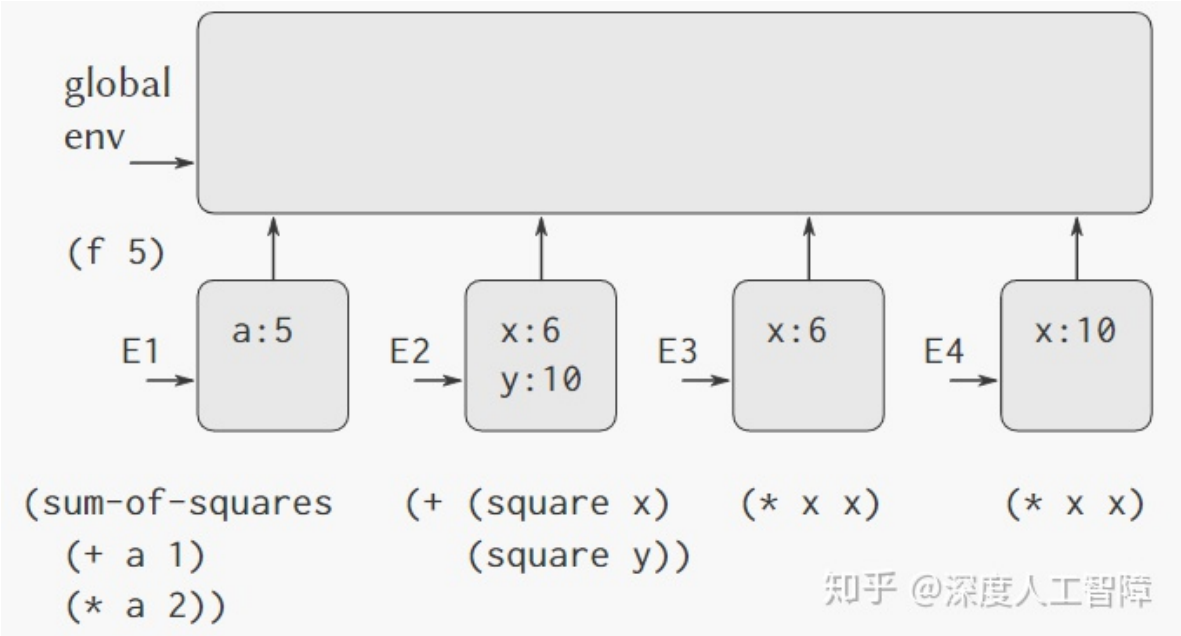
首先，会在全局环境中创建这些过程，得到以下过程对象



当我们应用过程 `(f 5)` 时，它的求值过程为：

- 创建一个新的环境 `E1`，其中包含形式参数约束 `a:5`，并且 `E1` 指向全局环境。然后在 `E1` 对 `f` 的过程体 `(sum-of-squares (+ a 1)(* a 2))` 进行求值
- - 首先，在 `E1` 中没有找到 `sum-of-squares` 的约束，就尝试在其外围环境寻找，在全局环境中找到了对应的约束 `(+(square x)(square y))`，也就获得了运算符子表达式的值。其次对于 `(+ a 1)` 和 `(* a 2)` 都能在当前环境 `E1` 中找到 `a` 对应的约束 `5`，所以可以直接求值组合式得到6和10。
  - 然后尝试将 `sum-of-squares` 的值应用于6和10
- 当将 `sum-of-squares` 应用于6和10时，也会创建一个新的环境 `E2`，其中包含形式参数 `x:6` 和 `y:10`，并且 `E2` 指向全局环境。然后对其过程体进行求值
- - 首先对 `(square x)` 进行求值，由于在 `E2` 中没有找到 `square` 的约束，所以就尝试在外围环境中找到了对应的约束，然后将 `square` 应用于6
    - - 首先创建一个新环境 `E3`，其中包含形式参数 `x:6`，可以直接对 `(* x x)` 进行求值
  - 其次对 `(square y)` 进行求值，也是在全局环境中找到 `square` 的约束，然后将 `squre` 应用于10

- 首先创建一个新环境 `E4`，其中包含形式参数 `x:10`，可以直接对 `(* x x)` 进行求值



注意：

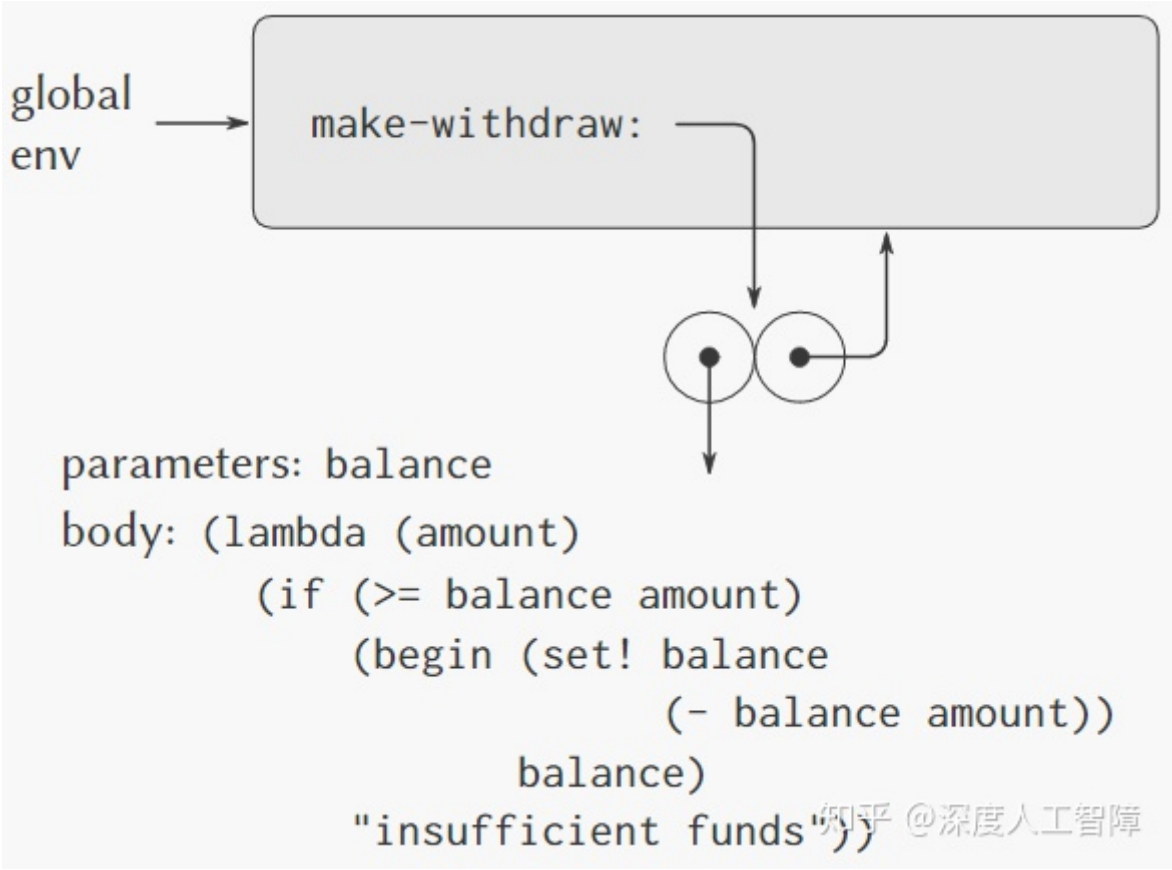
- 可以发现 `square` 的每次调用都会创建一个新环境
- 由于框架的存在，使得具有相同名字的局部变量互不干扰
- 我们这里只关心环境结构，对于如何在调用之间传递值，在后面会讨论。

2.2 有赋值的过程

我们以以下代码为例

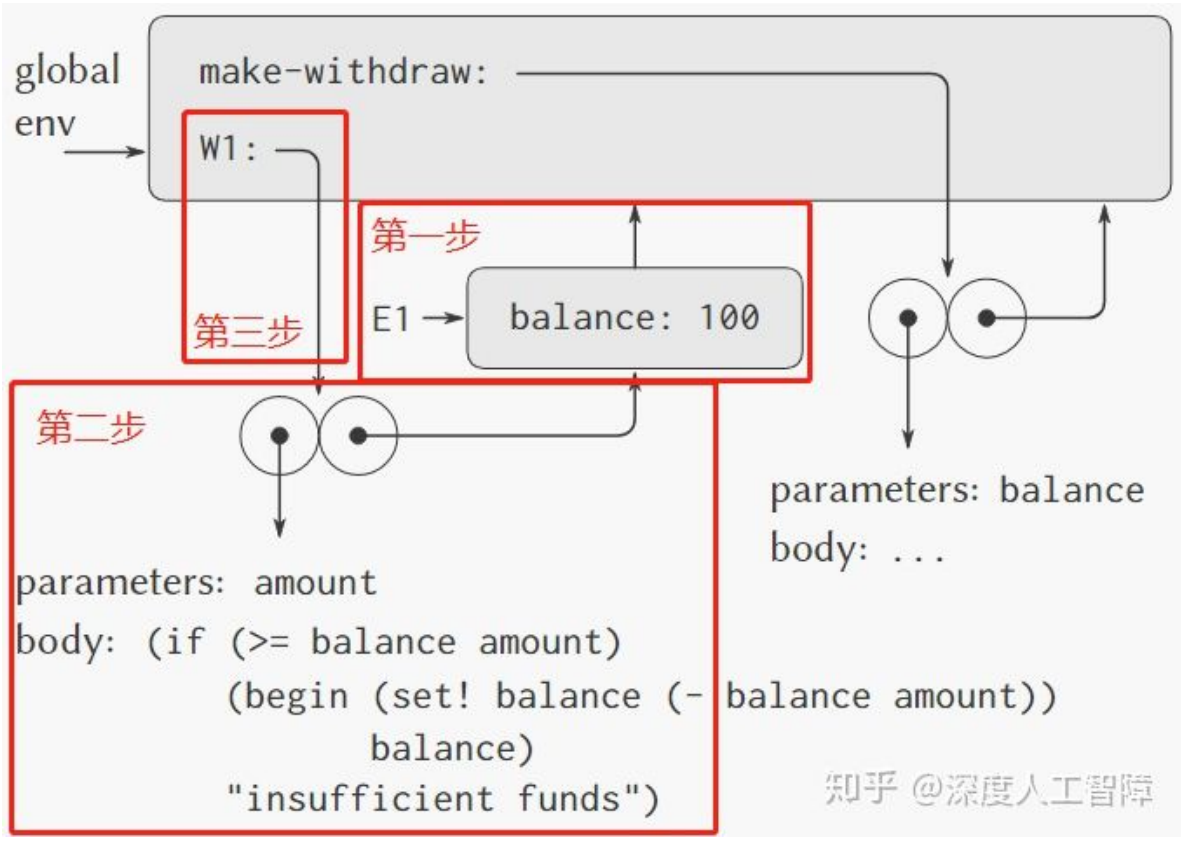
```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

创建过程时，会在全局环境中的框架添加一个符号 `make-withdraw`，然后对内部的 `lambda` 进行求值，得到如下过程对象



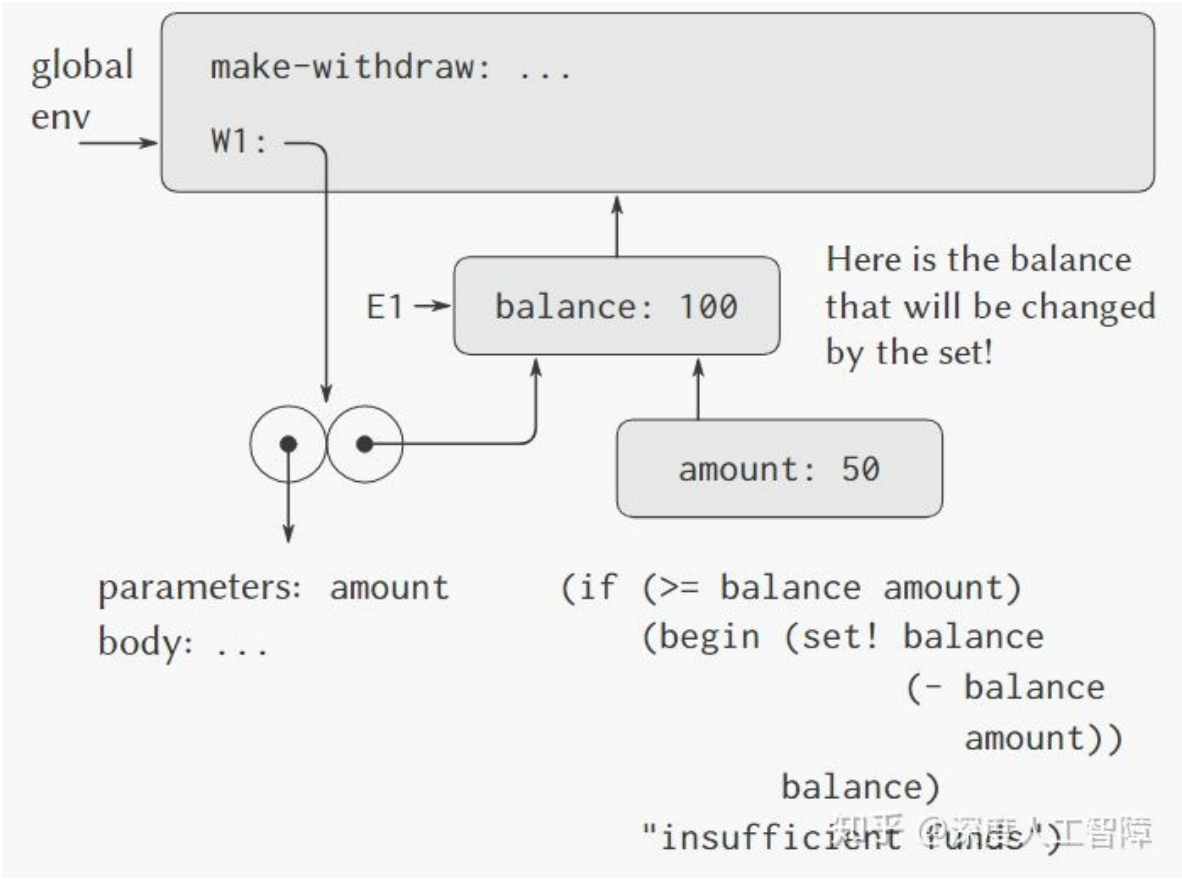
当我们对`(define W1 (make-withdraw 100))`进行求值时：

- 当将过程`make-withdraw`应用于100时，首先创建一个新环境`E1`，其中包含形式参数`balance`的约束，然后新环境的外围环境指针指向全局环境，然后求值过程体`lambda`
- - 对过程对象进行求值，会有一个指向过程体的指针，以及一个指向环境`E1`的指针
  - `define`会在全局环境中添加一个符号`W1`，然后将其指向求值的结果

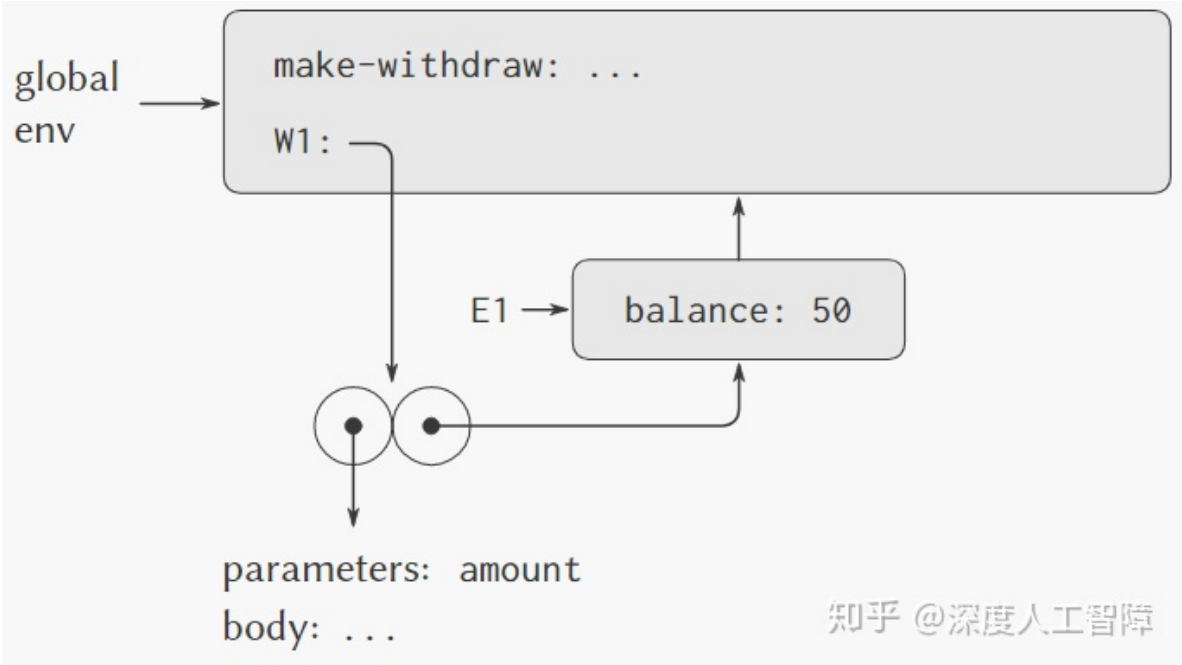


当我们对`(W1 50)`进行求值时：

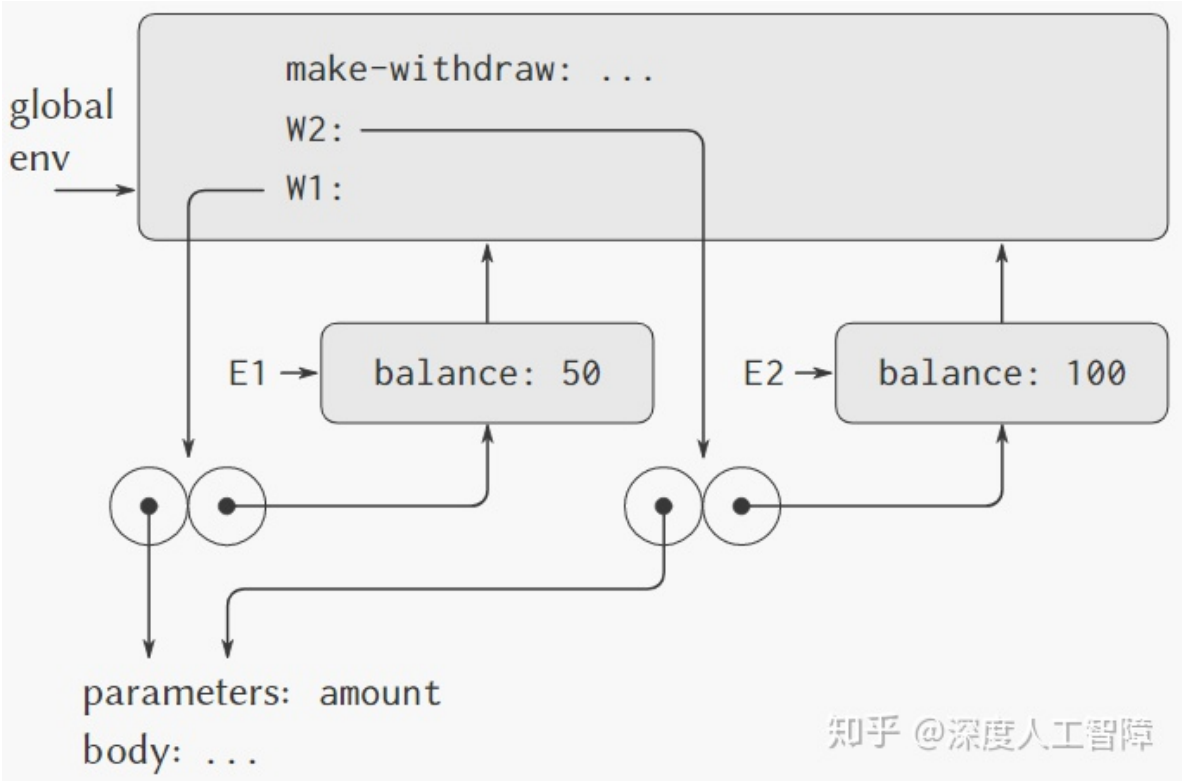
- 首先创建一个新环境，其中添加一个约束`amount:50`，通过`W1`指向的过程对象的环境，可知当前的外围环境为`E1`，因此有一个指向`E1`的指针。然后在当前环境对`W1`指向的过程体进行求值
- 这里的`amount`可以从当前环境中获得，而`balance`可以从`E1`中获得。当执行`set!`时，会修改位于`E1`中的`balance`



当求值完毕时，由于不存在任何指向约束`amount`框架的指针，所以可以直接将其忽略掉，则环境结构变为



同理，当我们再对`(define W2 (make-withdraw 100))`进行求值时，会得到以下环境结构



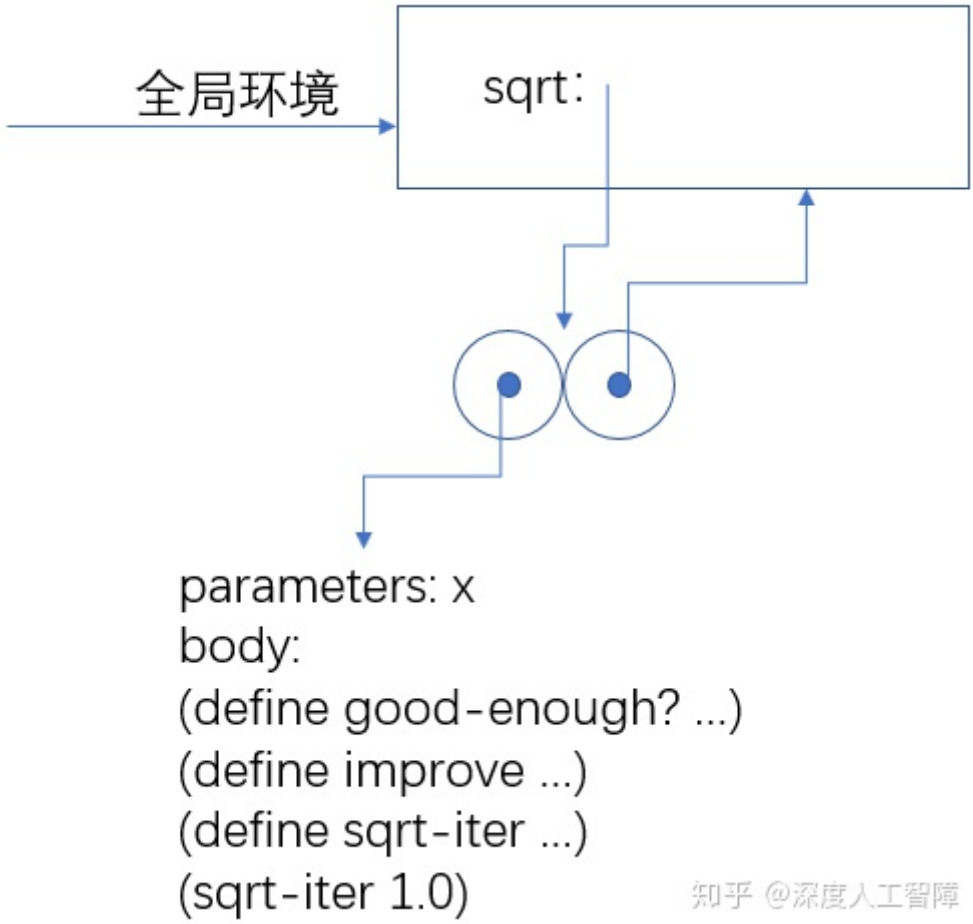
从中我们可以看出：每次对相同的`make-withdraw`进行求值，都会创建一个新的环境，在该环境中维护了自己的局部状态变量`balance`；而对内部的`lambda`表达式进行求值，就会产生指向自己环境的指针，从而只利用和修改自己的局部状态变量；而`define`会指向不同的过程对象序对，从而操作不同的过程对象（由过程对象的环境指针来区分）。

2.3 内部定义

以以下代码为例，探讨内部定义为什么有效

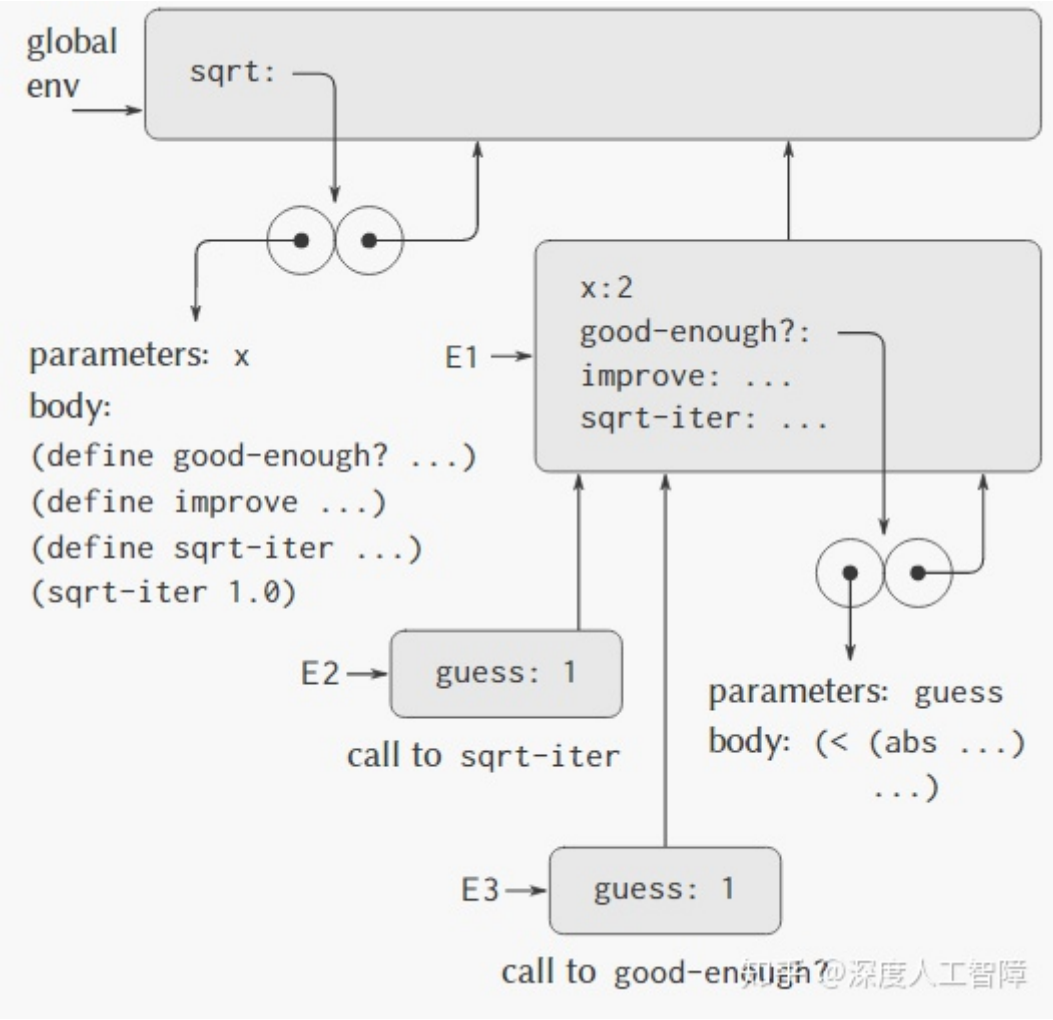
```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

首先创建该过程会得到以下环境结构



当对`(sqrt 2)`进行求值时，过程为：

- 首先创建一个新的环境`E1`，添加一个约束`x:2`，然后依次对`sqrt`的过程体进行求值
-
  - 其过程体内，首先是若干个`define`，它们将在环境`E1`中添加过程名的约束，其值为过程体的求值结果，注意环境指针指向的是`E1`
  - 最后是对`(sqrt-iter 1.0)`进行求值
- 首先创建一个新的环境`E2`，添加一个约束`guess`，并将其指向`E1`，然后在`E2`中对`sqrt-iter`的过程体进行求值
- 在里面会碰到对`(good-enough? guess)`，这里又创建一个新的环境`E3`，添加约束`guess`，并将其指向`E1`，然后在`E3`中对`good-enough?`过程体进行求值
- 以此类推



由此得到**两个性质**：

- 由于局部过程名都是在该过程运行时创建的框架内进行约束的，并不是在全局环境中进行约束的，所以局部过程的名字不会与外面的名字互相干扰。
- 局部过程只需将包含着它们的过程的形参作为自由变量，就可以访问该过程的实际参数。因为对于局部过程体的求值是外围过程求值所在的环境的下属。