工学部専門科目
「プログラミング言語」SICP
第3章〜Modularity, Objects, State〜
その2

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学 大学院情報学研究科
通信情報システム専攻

April 22, 2014

今日のメニュー

3.1.3 破壊的代入の導入の代償

- プログラム実行が単純な式の書き換えでは説明できない
- データの同値性とは何かの議論が難しくなる
- **物事を行う順序に気をつかう必要あり**

物事を行う順序について

functional な繰り返し版階乗関数

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

imperative な繰り返し版階乗関数

```
(define (factorial n)
  (let ((product 1) (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin
            (set! product (* counter product))
            (set! counter (+ counter 1))
              ;; この2行を交換すると…？
            (iter))))
    (iter)))
```

C言語で書くならこんな感じ?

```
int factorial(int n){
    int product = 1;
    int counter = 1;

    while (!(counter > n)) {
      product = counter * product;
      counter = counter + 1;
    }
    return product;
}
```

functional vs. imperative

imperative **なプログラミングでは**

- **処理の順序，すなわち状態変化の順序**
- **エイリアシング**

に気をつける必要あり

$\Rightarrow$ imperative **プログラミングは難しい!**

3.1 のまとめ

- 状態 (変数の値) の変化を引きおこす `set!`
- 局所状態をうまく使うと
  - 我々のまわりの「もの」のモデリングがしやすい
  - プログラムがすっきり (modular に) 書ける
- プログラムの実行モデルの複雑化
- 参照透過性の喪失
- エイリアシング，実行順に気をつける必要

今日のメニュー

3.2 評価の「環境モデル」

(1 章の) 代入モデル

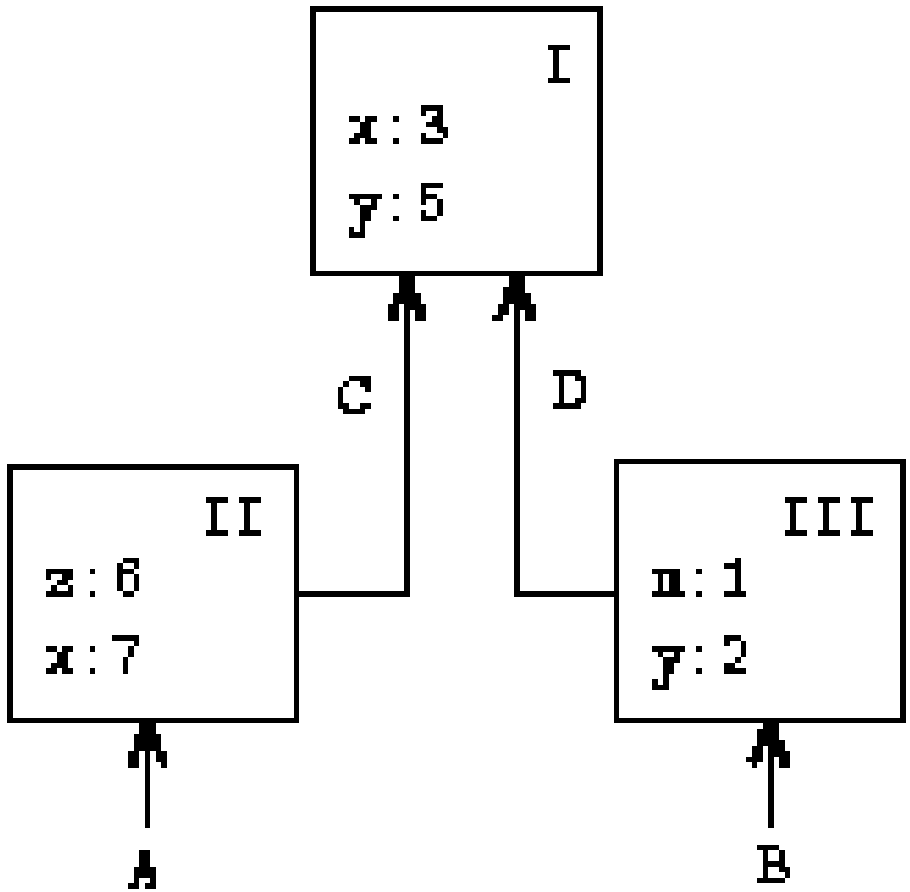
- 手続のパラメータは実引数を指す名前
- 名前が指す値は (有効範囲 (scope) 内で) どこでも同じ

環境モデル

- 手続のパラメータは実引数を**格納した箱**を指す名前
- 名前が指す箱の**中身が** `set!` **で変わる**
- 環境: 実引数を格納する箱の集まり

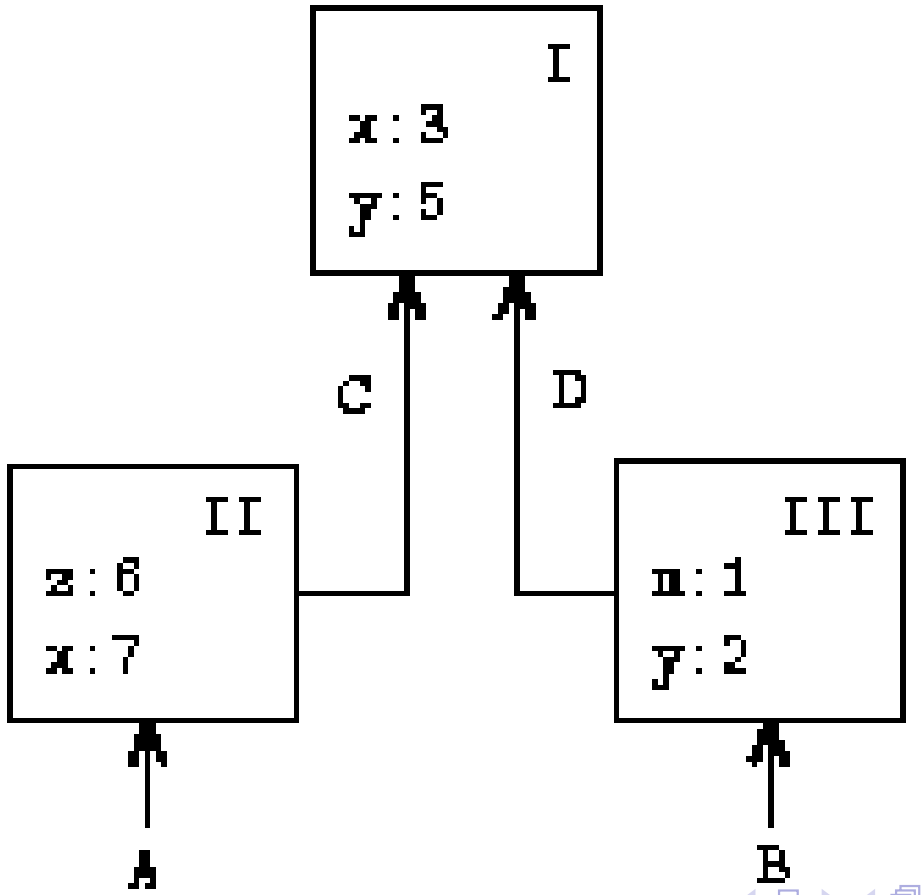
環境

- 環境 = フレームの列
- フレーム = 束縛 (binding) の列
- 束縛 = 変数とその値の組



環境

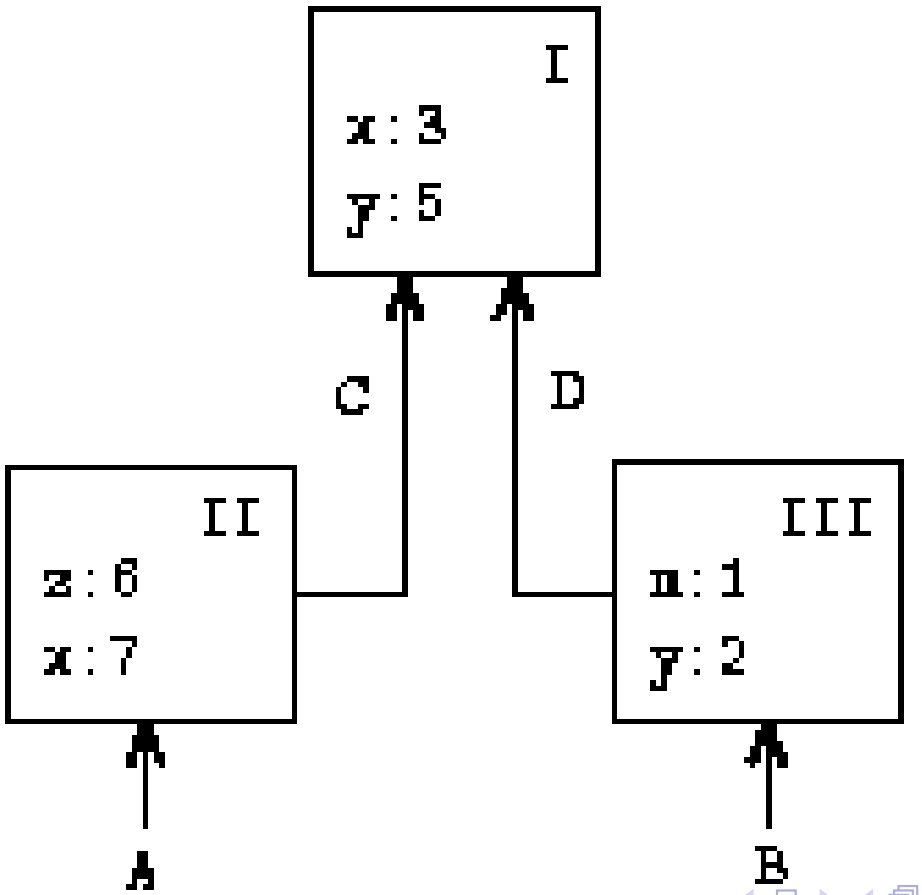
- **環境** C (**もしくは** D) **のもとで** y **の値は** 5
- **環境** B **のもとで** x **の値は** 3
- **環境** A **のもとで** x **の値は** 7 (shadowing)



式の値についての語り方

「環境○○のもとで式 e の値は v」

- 環境 C のもとで (+ x y) の値は 8
- 環境 A のもとで (+ x y) の値は 12



`define`の動作

環境 A のもとでの (define x ⟨式⟩) の動作

- A のもとでの ⟨式⟩ の値 v を求める
- A の最初のフレーム F に x の束縛が
  - あれば，その値を v に**変更**
  - なければ x:v を F に**追加**

let の動作

環境 A のもとでの

$$(\texttt{let}\ ((x_1\ e_1)\ \ldots\ (x_n\ e_n))\ e_0)$$

の値:

- **環境 A のもとで各式 e_i の値を計算し，これを v_i とする**
- **束縛 $x_i : v_i$ から成る新しいフレームを作り，A を指すようにする．この環境を B とする**
- **B のもとで式 e_0 の値を求めたものが let 式全体の値**

lambda の値

手続式の値 =

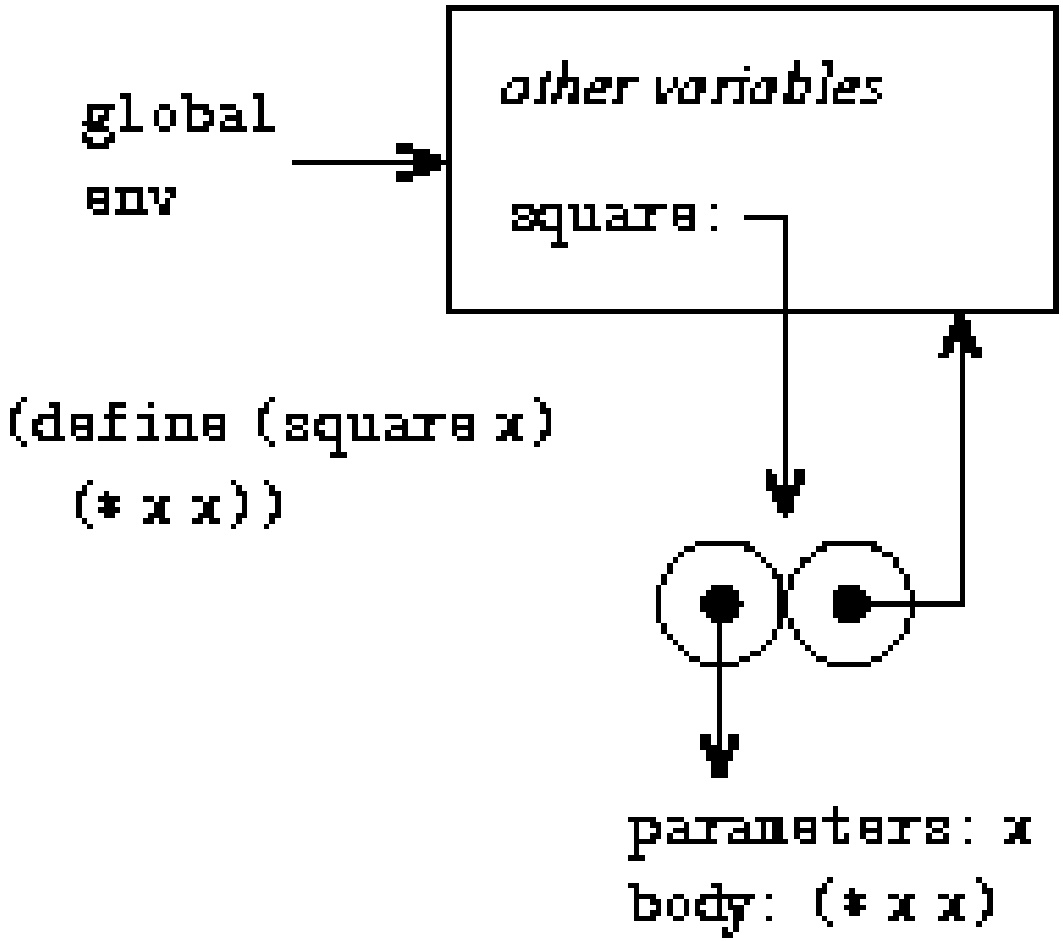
- **パラメータと本体式**
  - **後で手続が呼び出された時に必要**
- **lambda を評価した時点での環境 (へのポインタ)**
  - **パラメータ以外の変数の値を知るため**

の組

$\Longrightarrow$ この組を関数閉包 (function closure) という

(define (square x) (* x x)) 実行後の環境

(define square (lambda (x) (* x x))) と同じ

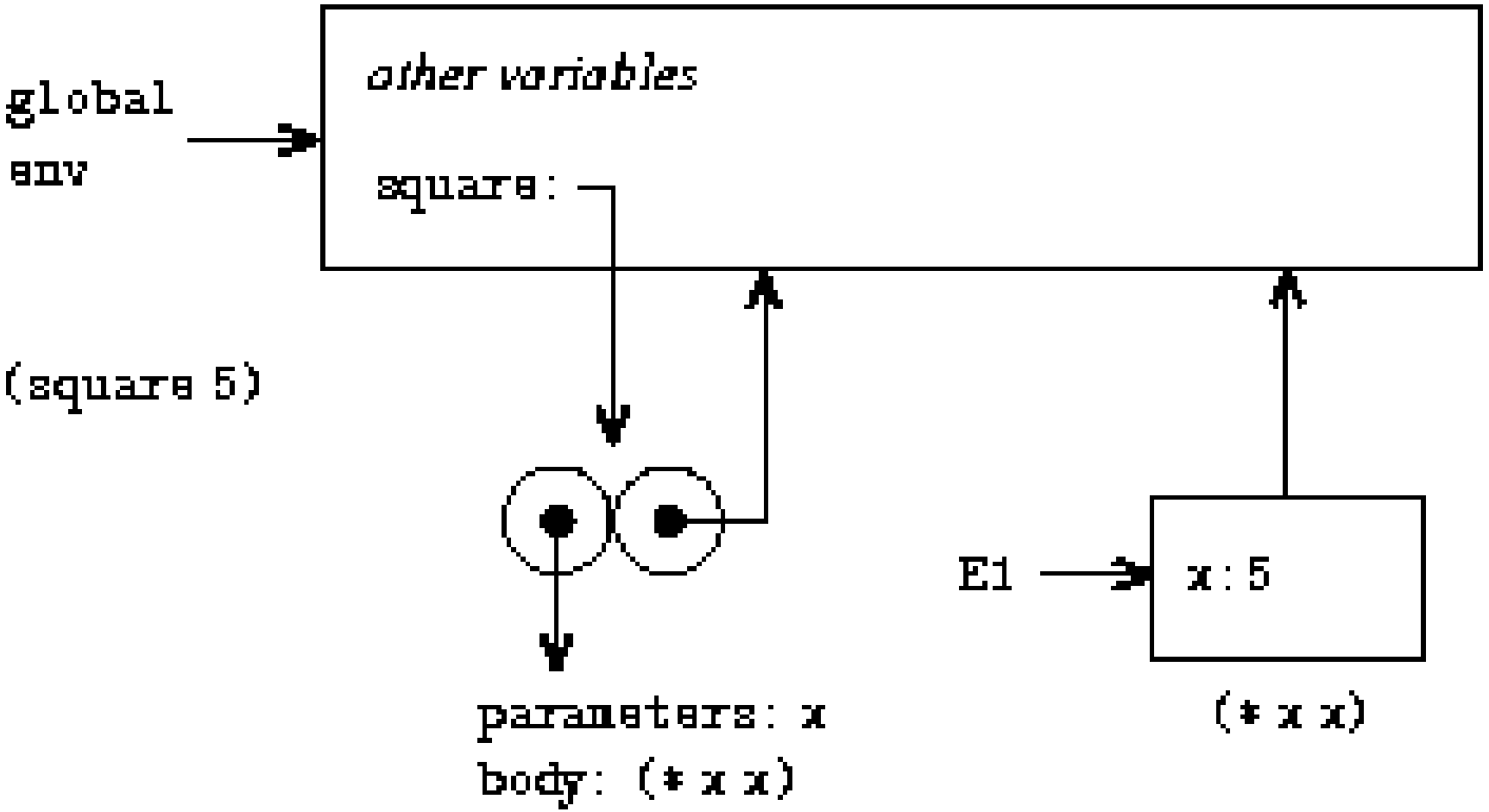


関数適用式の評価

環境 A のもとでの関数適用式 $(e_0\ e_1\ \ldots e_n)$ の値:

- A のもとでの e_i の値 を v_i とする
- 関数閉包 v_0 の中身の環境を B，パラメータを $x_1, \ldots, x_n$ とする
  - v_0 が関数閉包でなかったり，パラメータの数が n でない場合はエラー
- 束縛 $x_i : v_i$ から成る新しいフレームを作り，B を指すようにする．この環境を C とする
- C のもとで関数閉包の本体式 e の値を求めたものが，関数適用式全体の値

(square 5) 評価途中の環境



チェックポイント!

以下のふたつの式の評価の結果が同じであることを確認しよう

- `(let ((`x_1 e_1`) ...(`x_n e_n`)) `e_0`)`
- `((lambda (`x_1 `...`x_n`) `e_0`) `e_1 `...`e_n`)`

set! の評価

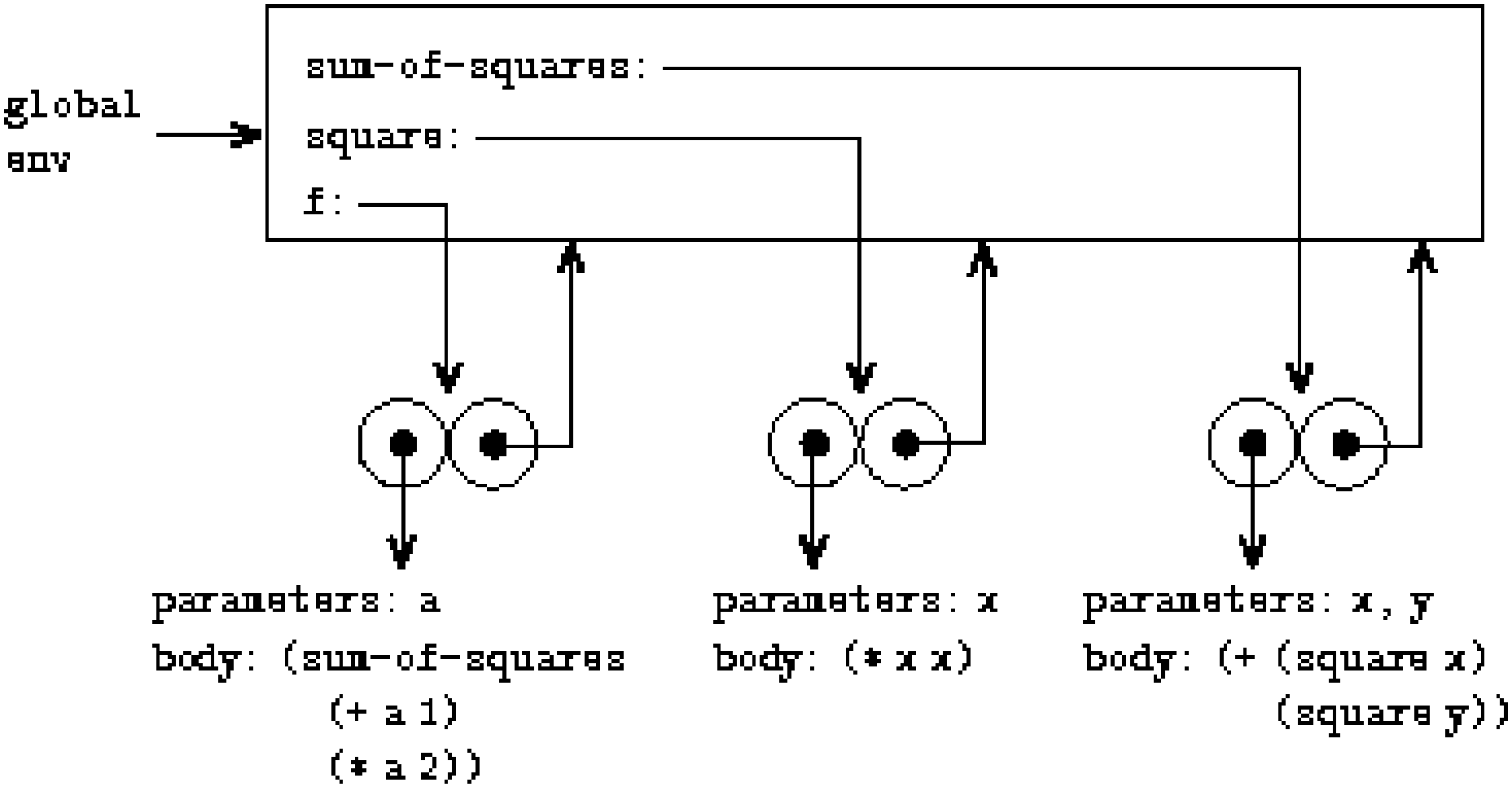
環境 A のもとでの (set! x e) の評価:

- **A のもとでの e の値を v とする**
- **A の中の (最初の) x の束縛を v に変更**
  - **x の束縛がなければエラー**
  - **(define との違いに注意)**

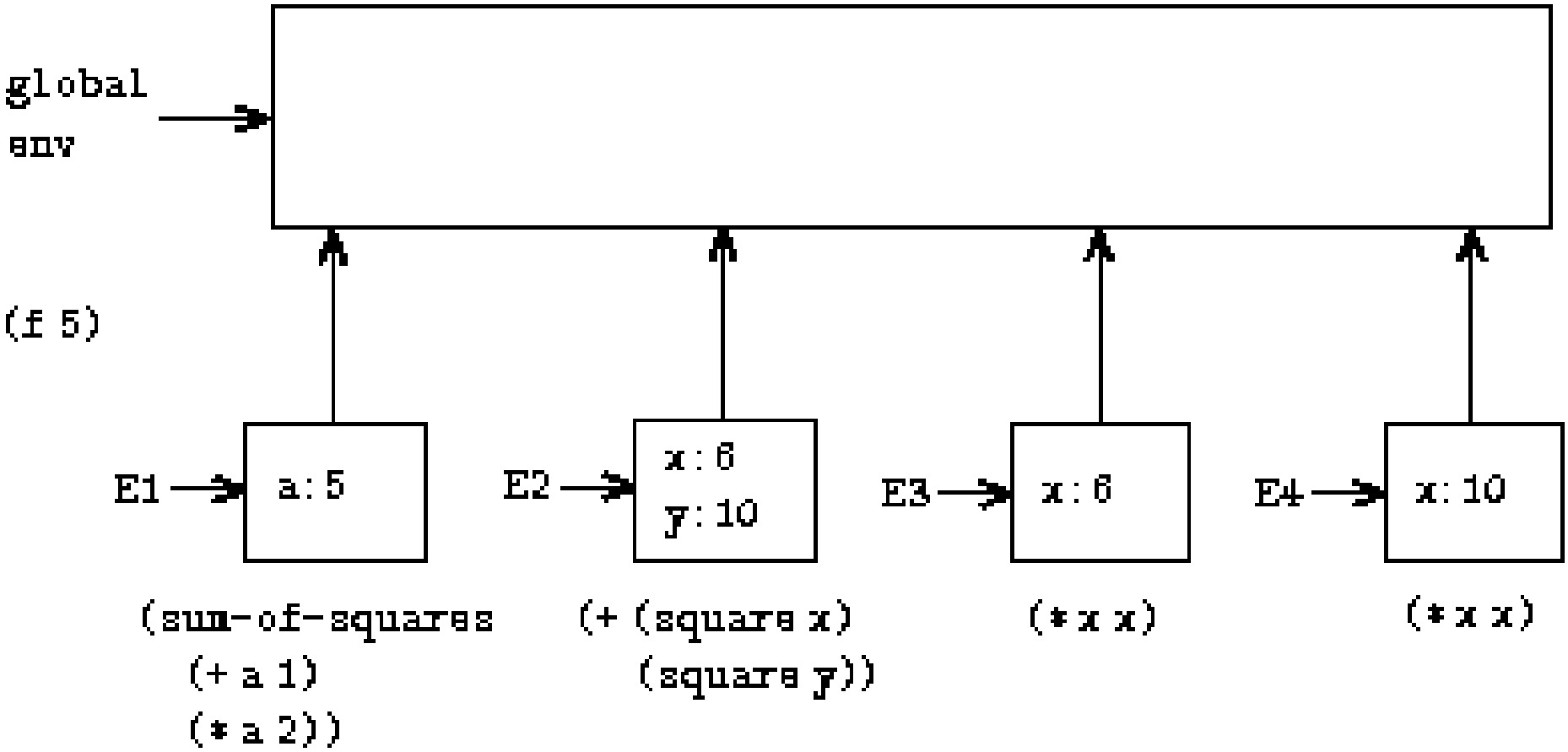
3.2.2 単純な手続の適用

```
(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)))
(f 5)
```

最初の 3 つの define 実行後の環境



(f 5) の評価中に作られるフレーム E1〜E4



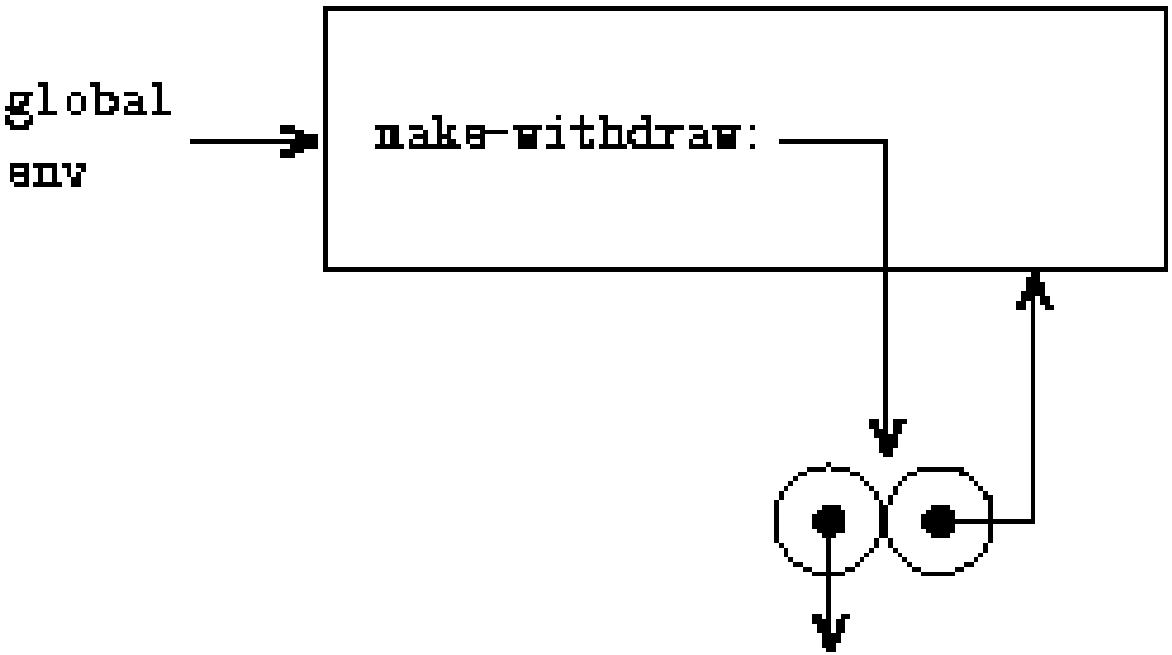
見逃しがちだけど大事なポイント

- E1 のもとで `(sum-of-squares ...)` を評価する際，`sum-of-squares` の束縛がどう見つかるか
- E2 のもとで `(+ (square x) (square y))` を評価する際，`x` の値が異なるふたつのフレームができる
  - 局所変数の「局所的」たるゆえん

3.2.3 局所状態貯蔵庫としてのフレーム

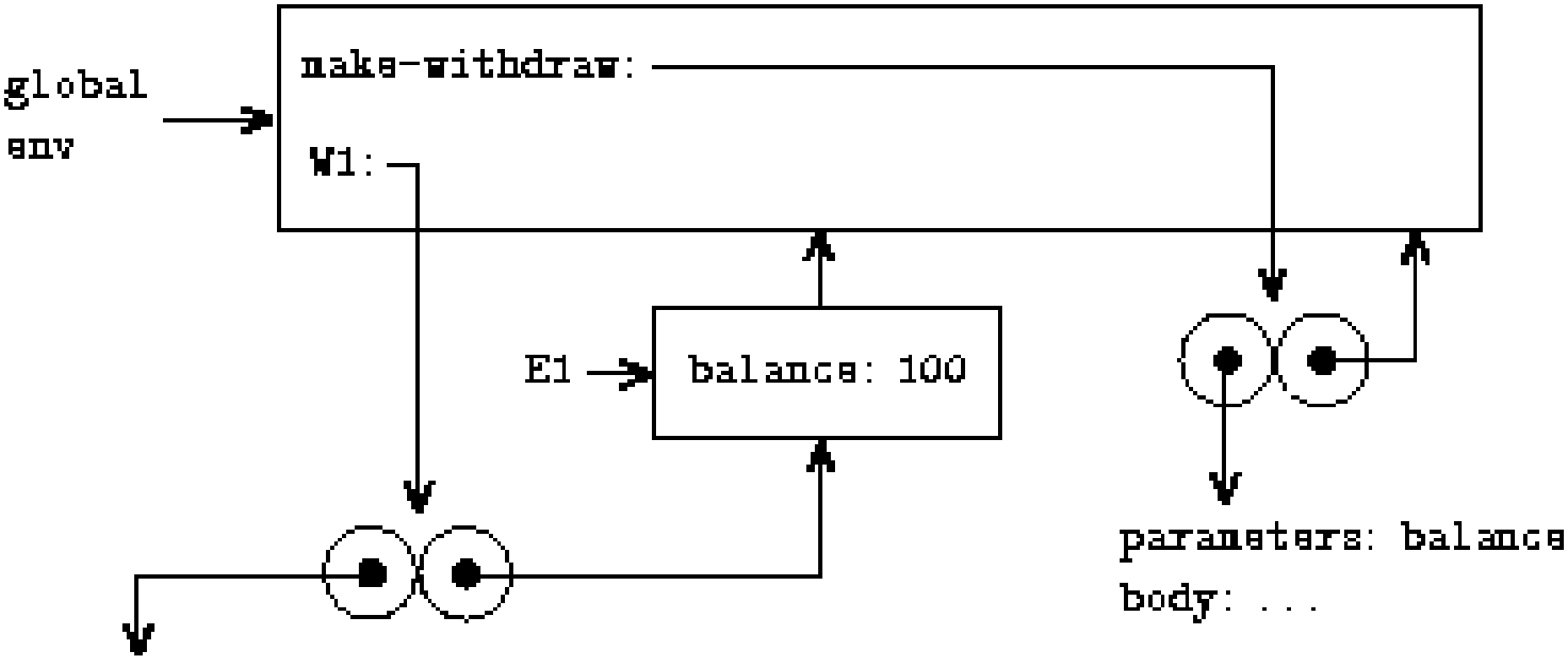
```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "Insufficient funds")))

(define W1 (make-withdraw 100))

(W1 50)
```

(define (make-withdraw balance) ...) 実行直後の様子



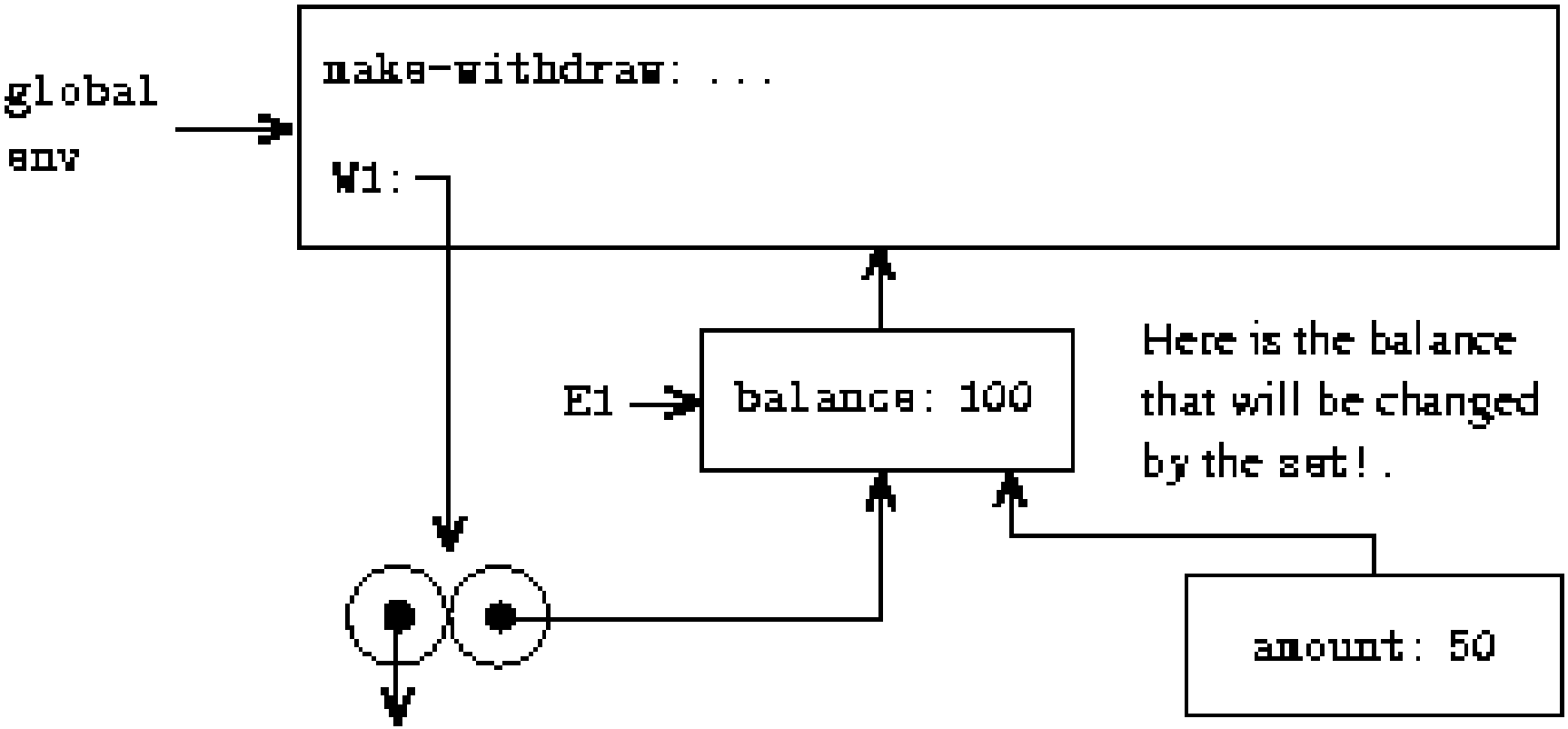

```
parameters: balance
body: (lambda (amount)
        (if (>= balance amount)
            (begin (set! balance (- balance amount))
                   balance)
            "Insufficient funds"))
```

(define W1 ...) 実行直後の様子

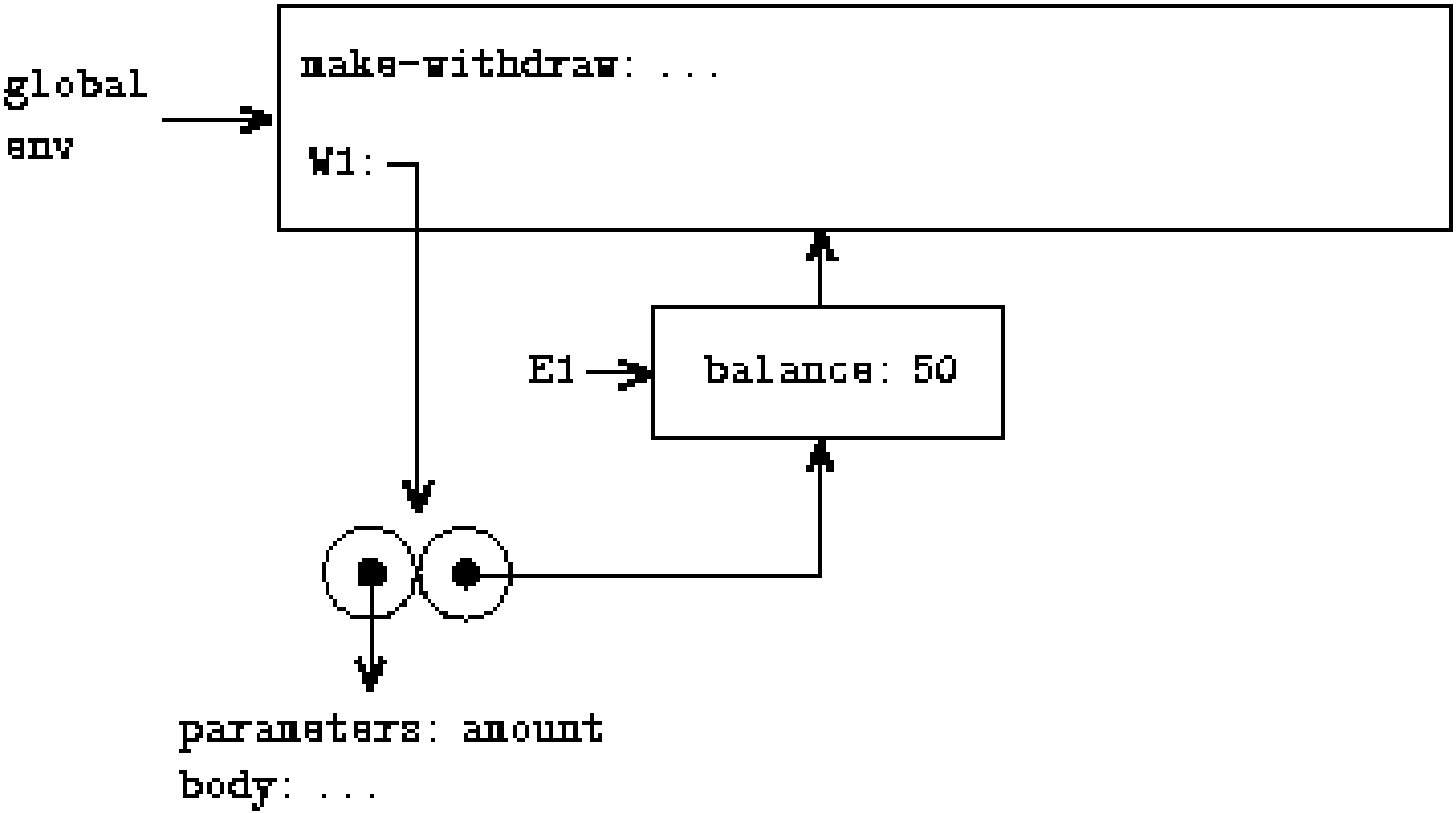



```
parameters: amount
body: (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))
```

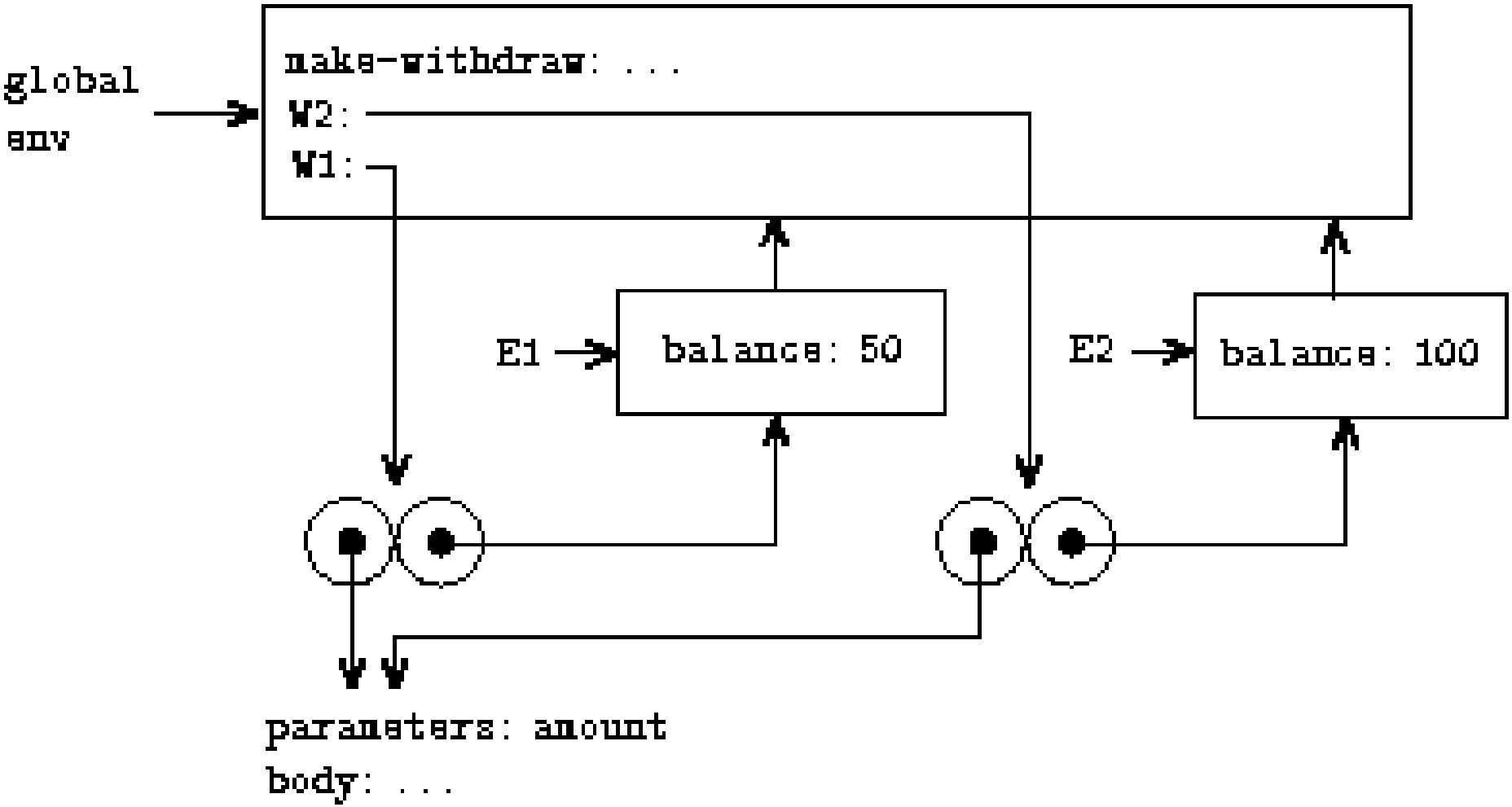
(W1 50) の W1 呼び出し直後の様子



(W1 50) 実行直後の様子



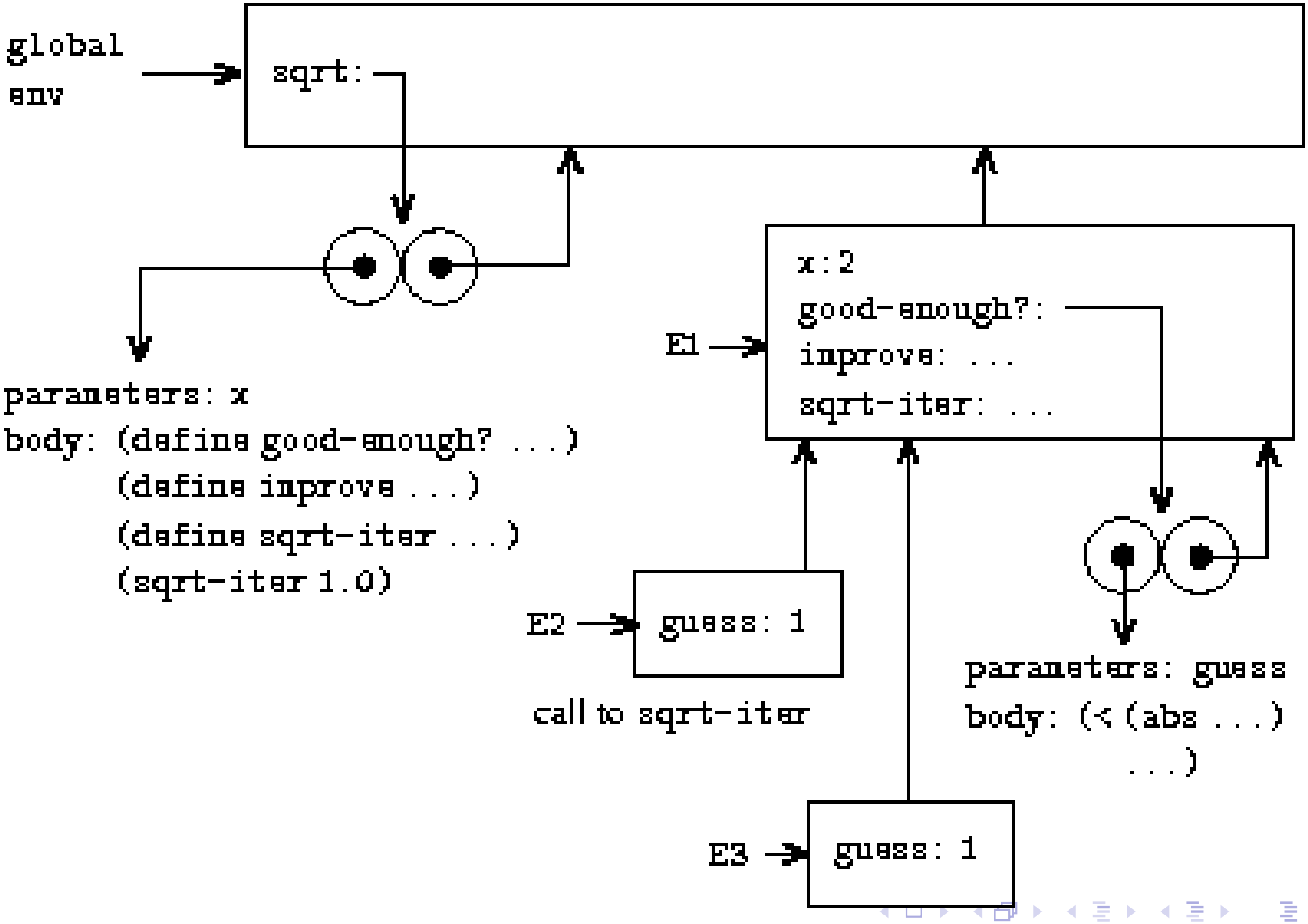
(deine W2 (make-withdrawal 100)) の結果



3.2.4 内部定義

```
(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))
(sqrt 2)
```

(sqrt 2) の評価中，最初の good-enough? 呼び出し直後の様子



評価の環境モデルまとめ

- 局所定義された手続きで使われる名前と外側の名前が干渉しない
  - $\Leftarrow$ 束縛されるフレームが違うため
- 局所定義された手続き内では，shadowing がない限り外側で定義された名前が参照できる
  - 定義を表すフレームが内側 (先頭) から数珠繋ぎになっていて，順に探索していくため
- `set!` は単なる束縛の書き換え

3.3 節「変更可能データを使ったモデリング」

予習ポイント:

- `set-car!`, `set-cdr!` **の動作は何か？**
- **以下のふたつのプログラムの違いは何か？**
  - ```
    (define y
      (cons (list 'a 'b) (list 'a 'b)))
    ```
  - ```
    (define x (list 'a 'b))
    (define y (cons x x))
    ```
- **キュー** (queue) **とはどのようなデータ構造か？** FIFO **とは何？**

宿題： 5/13(火) 午前10:30 締切

- Ex. 3.9 とチェックポイント (19 枚目) についての説明
- 今回はプログラムは不要
- レポート (pdf) を提出システムを通じて提出
- 友達に教えてもらったら、その人の名前を明記
- web は出典を明記 (「同じ」回答は減点)