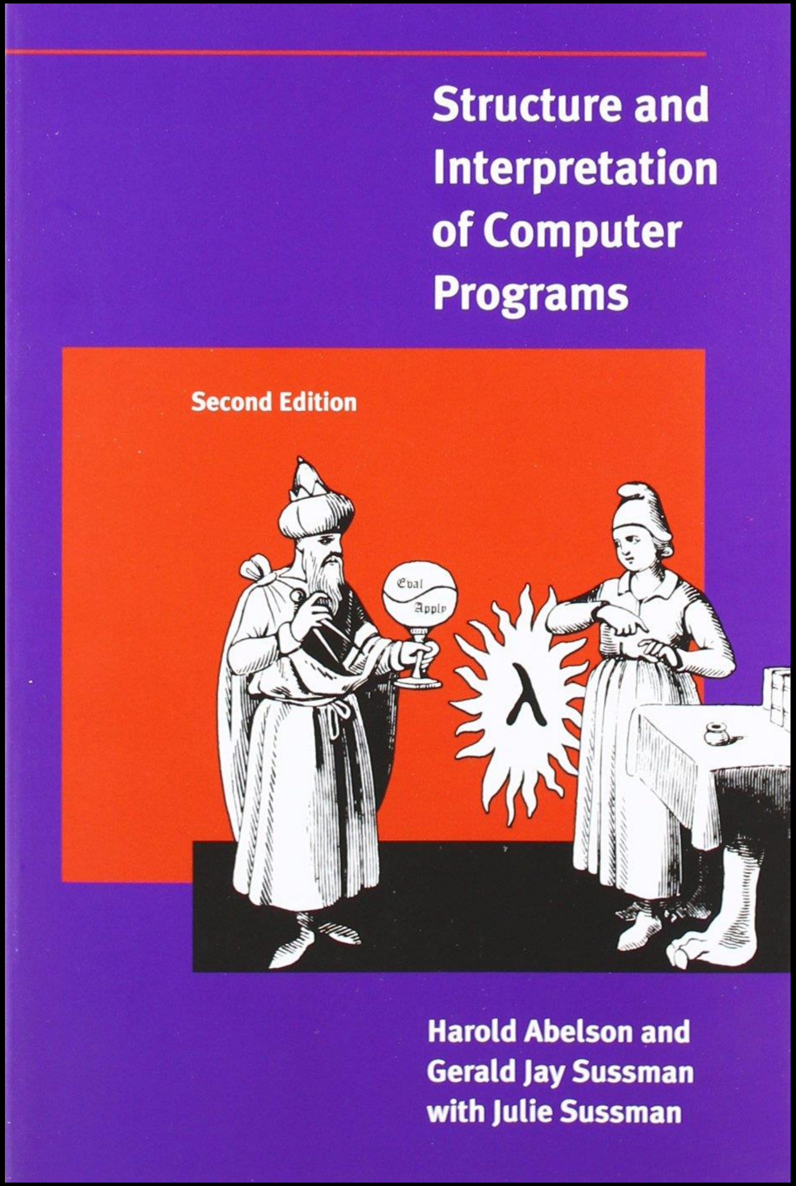
Java

Meetup


Structure and
Interpretation
of Computer
Programs
Second Edition
Eval
Apply
λ
Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

Structure and Interpretation of Computer Programs

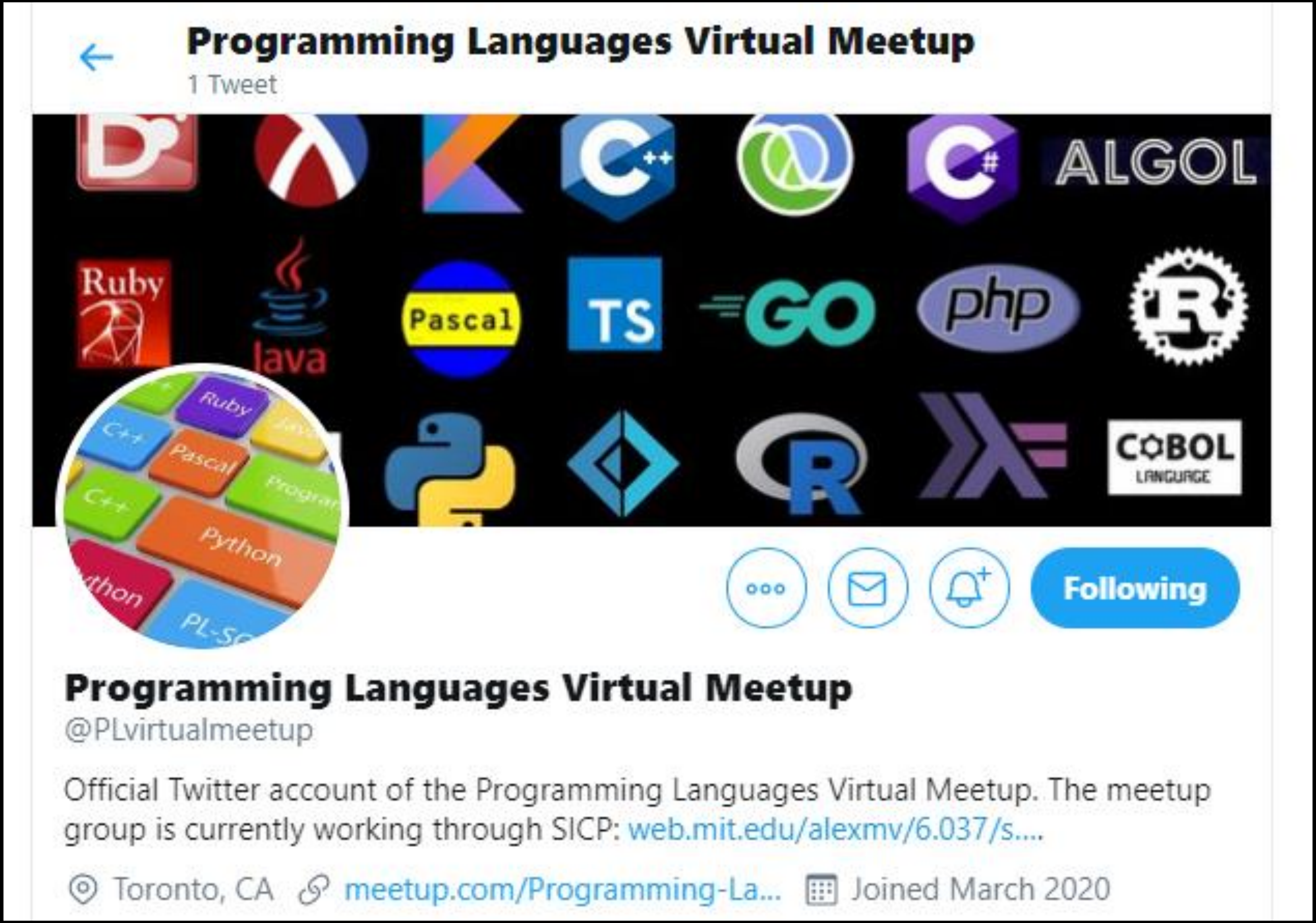
Chaper 3.2

Before we start …

Friendly Environment Policy

Berlin Code of Conduct

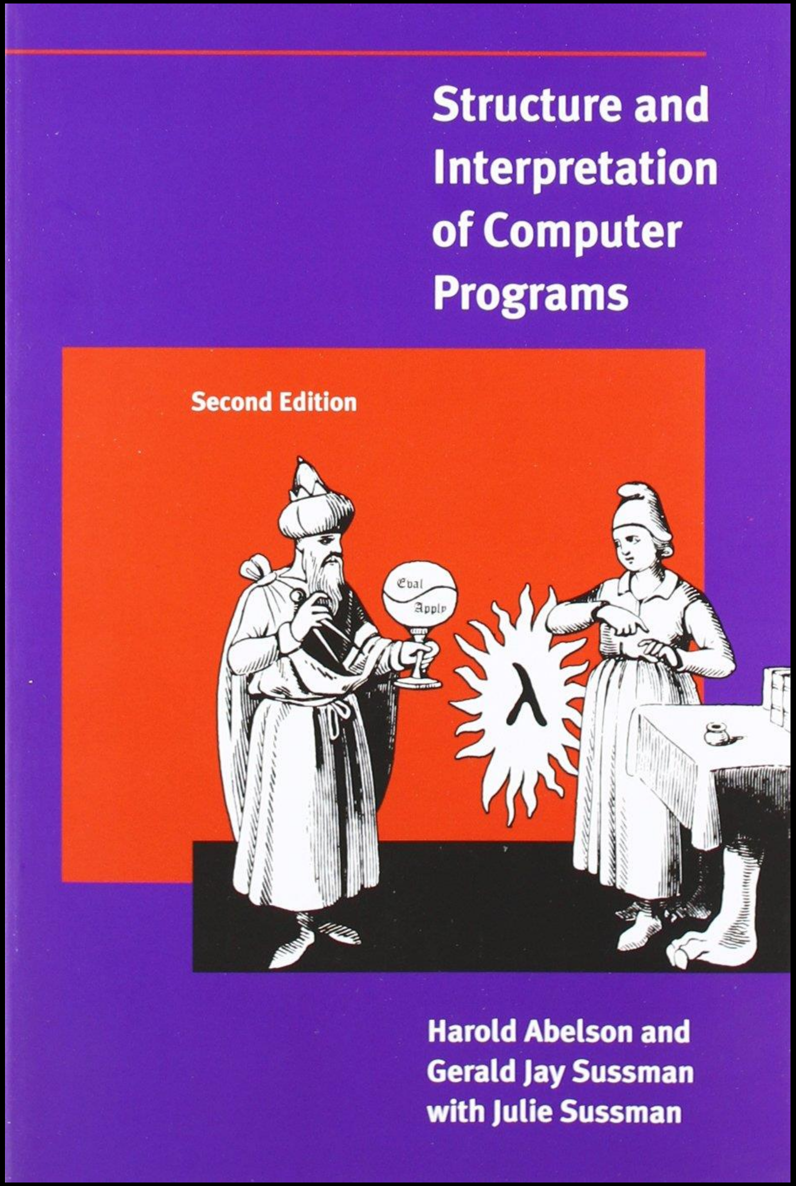
DISCORD



Programming Languages Virtual Meetup
@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020


Structure and Interpretation of Computer Programs
Second Edition
Eval
Apply
λ
Harold Abelson and Gerald Jay Sussman with Julie Sussman

Structure and Interpretation of Computer Programs Chaper 3.2

An environment is a sequence of *frames*. Each frame is a table (possibly empty) of *bindings*, which associate variable names with their corresponding values. (A single frame may contain at most one binding for any variable.) Each frame also has a pointer to its *enclosing environment*, unless, for the purposes of discussion, the frame is considered to be *global*. The *value of a variable* with respect to an environment is the value given by the binding of the variable in the first frame in the environment that contains a binding for that variable. If no frame in the sequence specifies a binding for the variable, then the variable is said to be *unbound* in the environment.

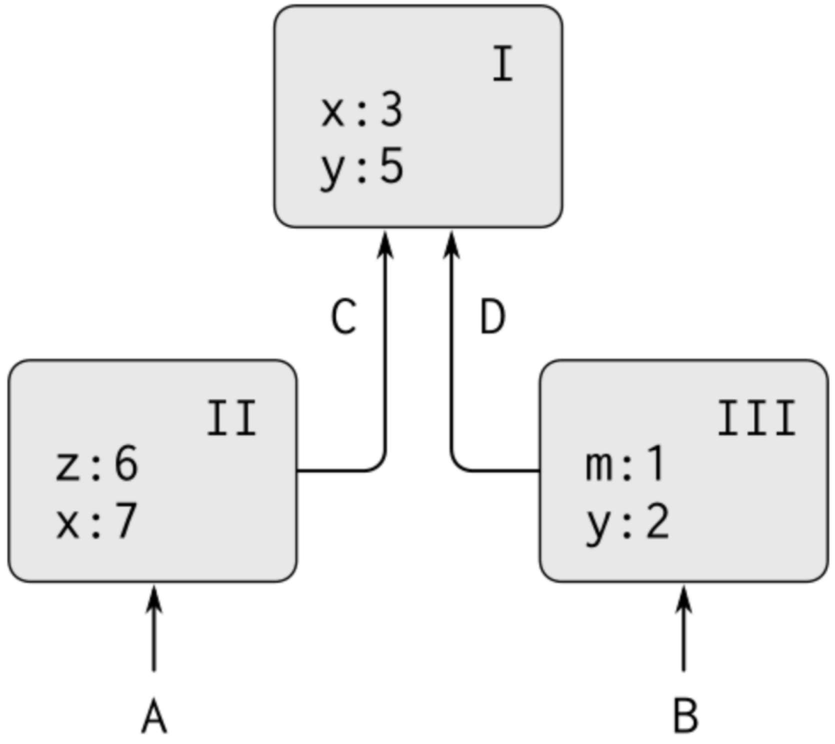


Figure 3.1: A simple environment structure.

- To evaluate a combination:

1. Evaluate the subexpressions of the combination.[12]
2. Apply the value of the operator subexpression to the values of the operand subexpressions.

The environment model of evaluation replaces the substitution model in specifying what it means to apply a compound procedure to arguments.

In the environment model of evaluation, a procedure is always a pair consisting of some code and a pointer to an environment. Procedures are created in one way only: by evaluating a λ-expression. This produces a procedure whose code is obtained from the text of the λ-expression and whose environment is the environment in which the λ-expression was evaluated to produce the procedure. For example, consider the procedure definition

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

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

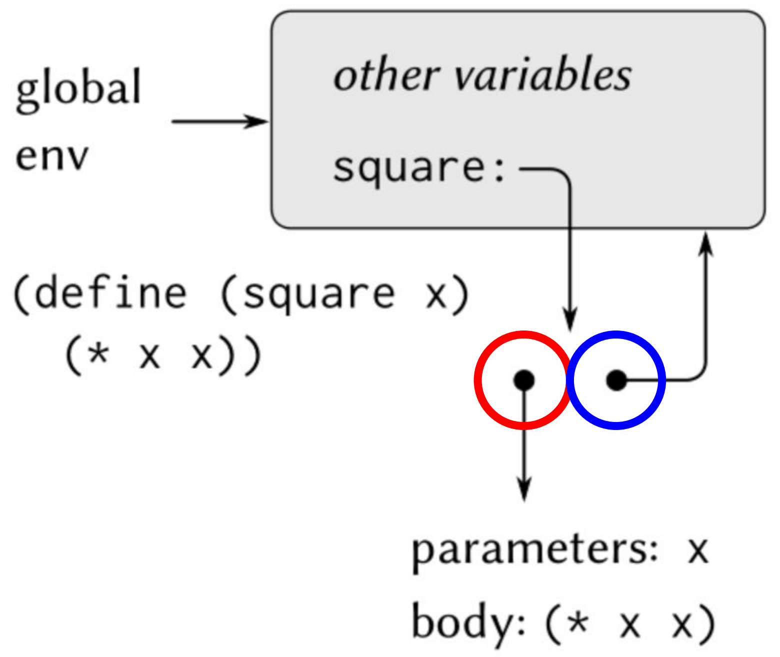


Figure 3.2: Environment structure produced by evaluating `(define (square x) (* x x))` in the global environment.

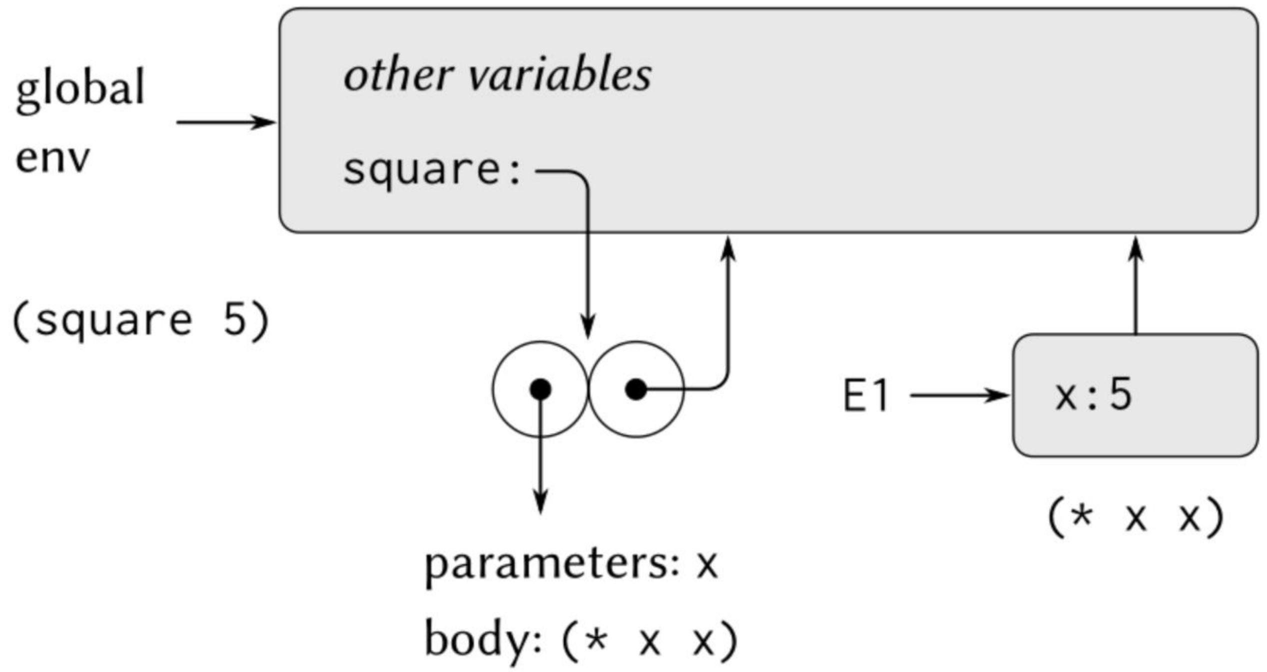


Figure 3.3: Environment created by evaluating `(square 5)` in the global environment.

The environment model of procedure application can be summarized by two rules:

- A procedure object is applied to a set of arguments by constructing a frame, binding the formal parameters of the procedure to the arguments of the call, and then evaluating the body of the procedure in the context of the new environment constructed. The new frame has as its enclosing environment the environment part of the procedure object being applied.
- A procedure is created by evaluating a λ-expression relative to a given environment. The resulting procedure object is a pair consisting of the text of the λ-expression and a pointer to the environment in which the procedure was created.

```
(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)))
```

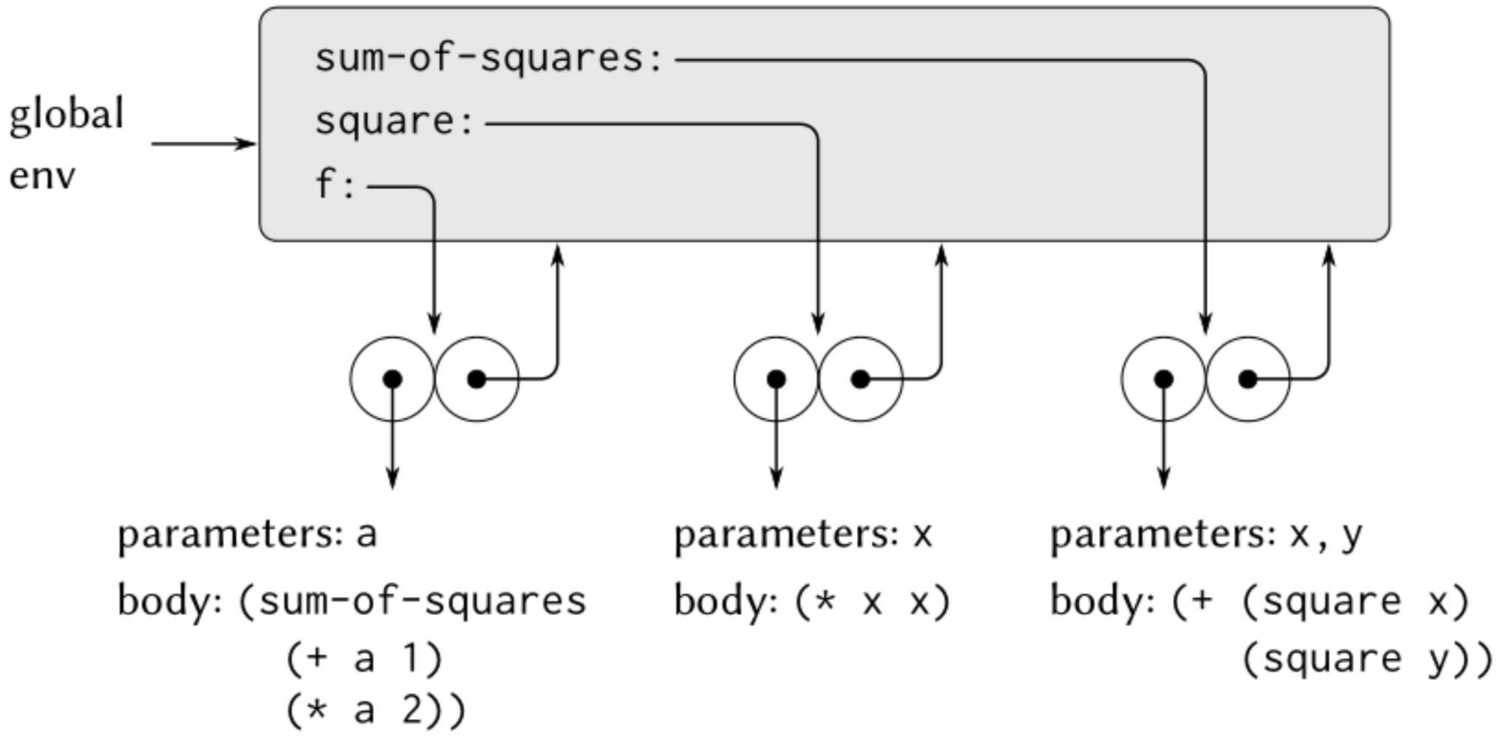


Figure 3.4: Procedure objects in the global frame.

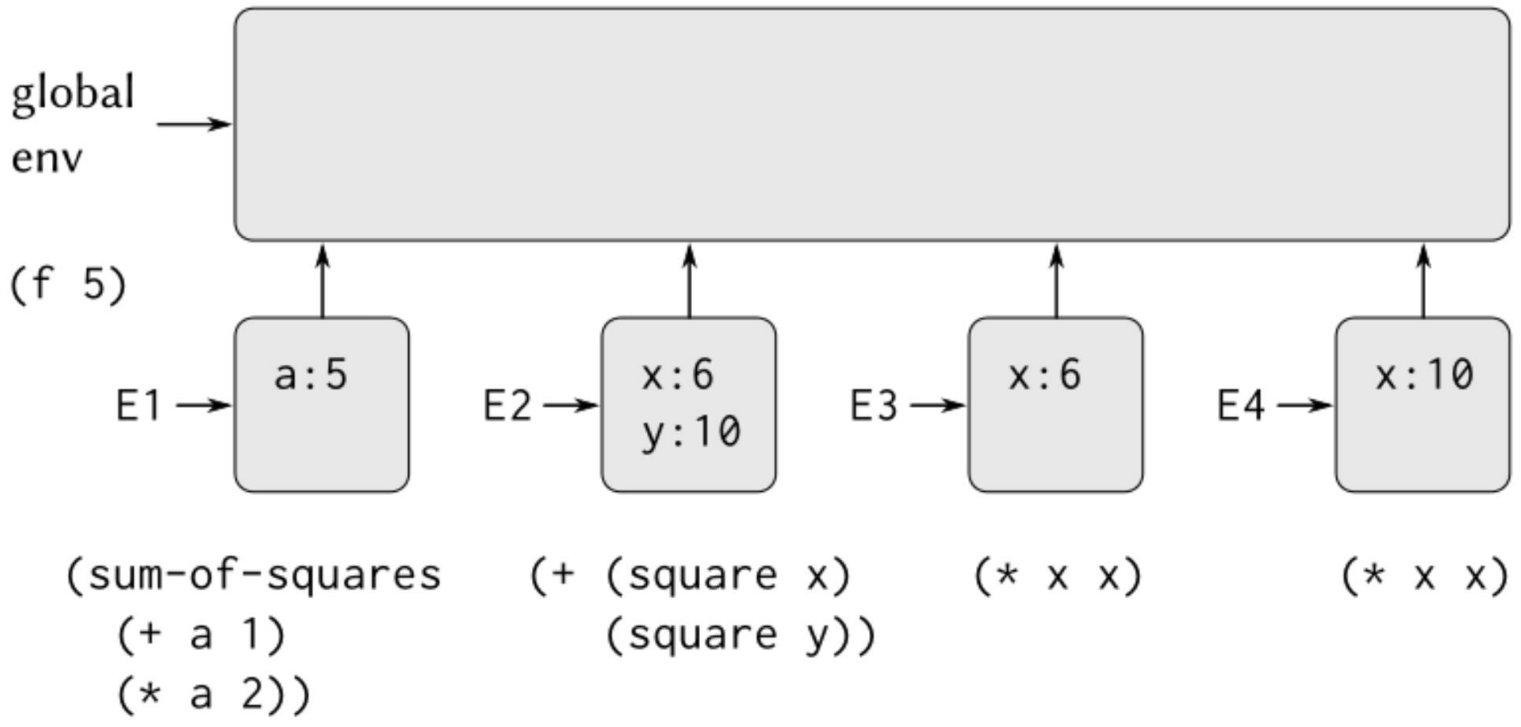


Figure 3.5: Environments created by evaluating `(f 5)` using the procedures in Figure 3.4.

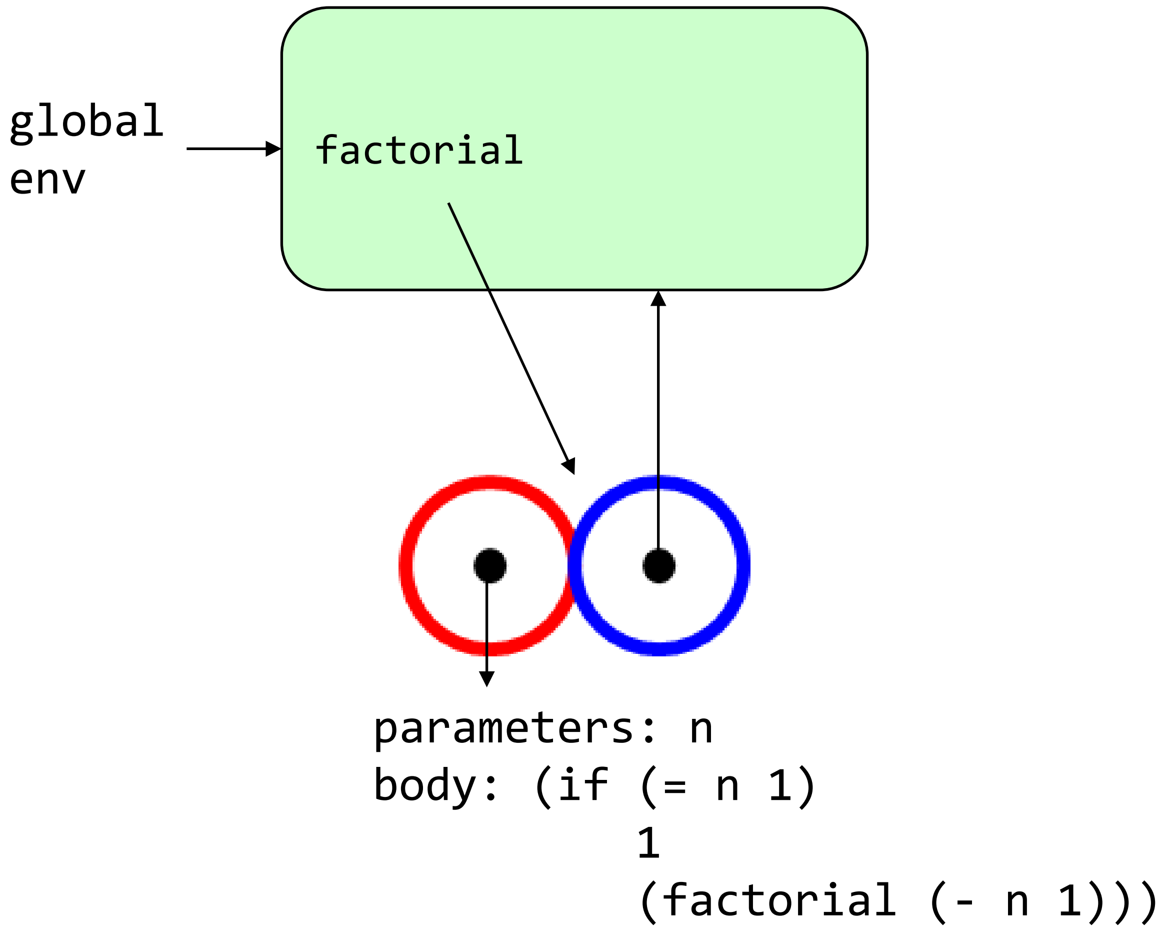
Exercise 3.9: In Section 1.2.1 we used the substitution model to analyze two procedures for computing factorials, a recursive version

```
(define (factorial n)
  (if (= n 1) 1 (* n (factorial (- n 1)))))
```

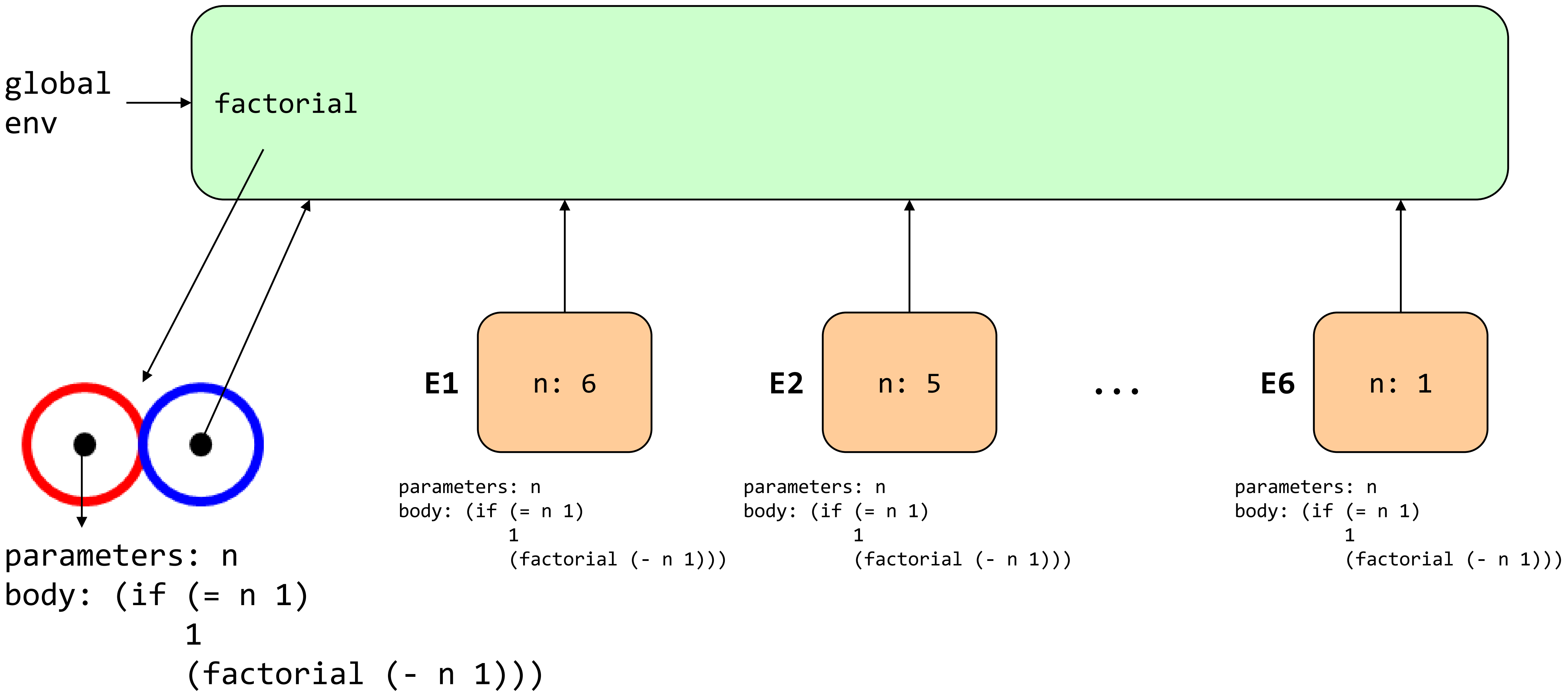
and an iterative version

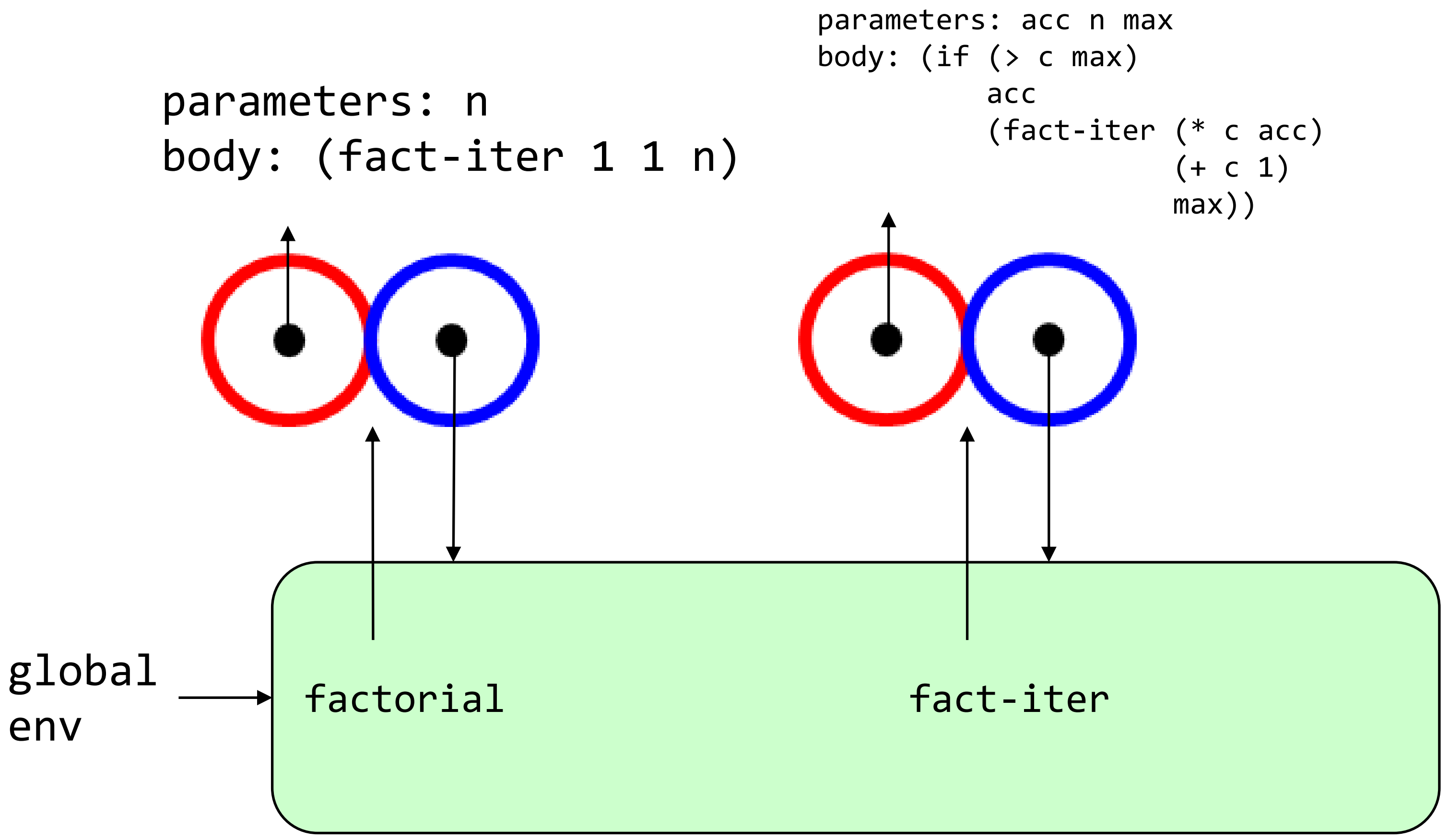
```
(define (factorial n) (fact-iter 1 1 n))
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

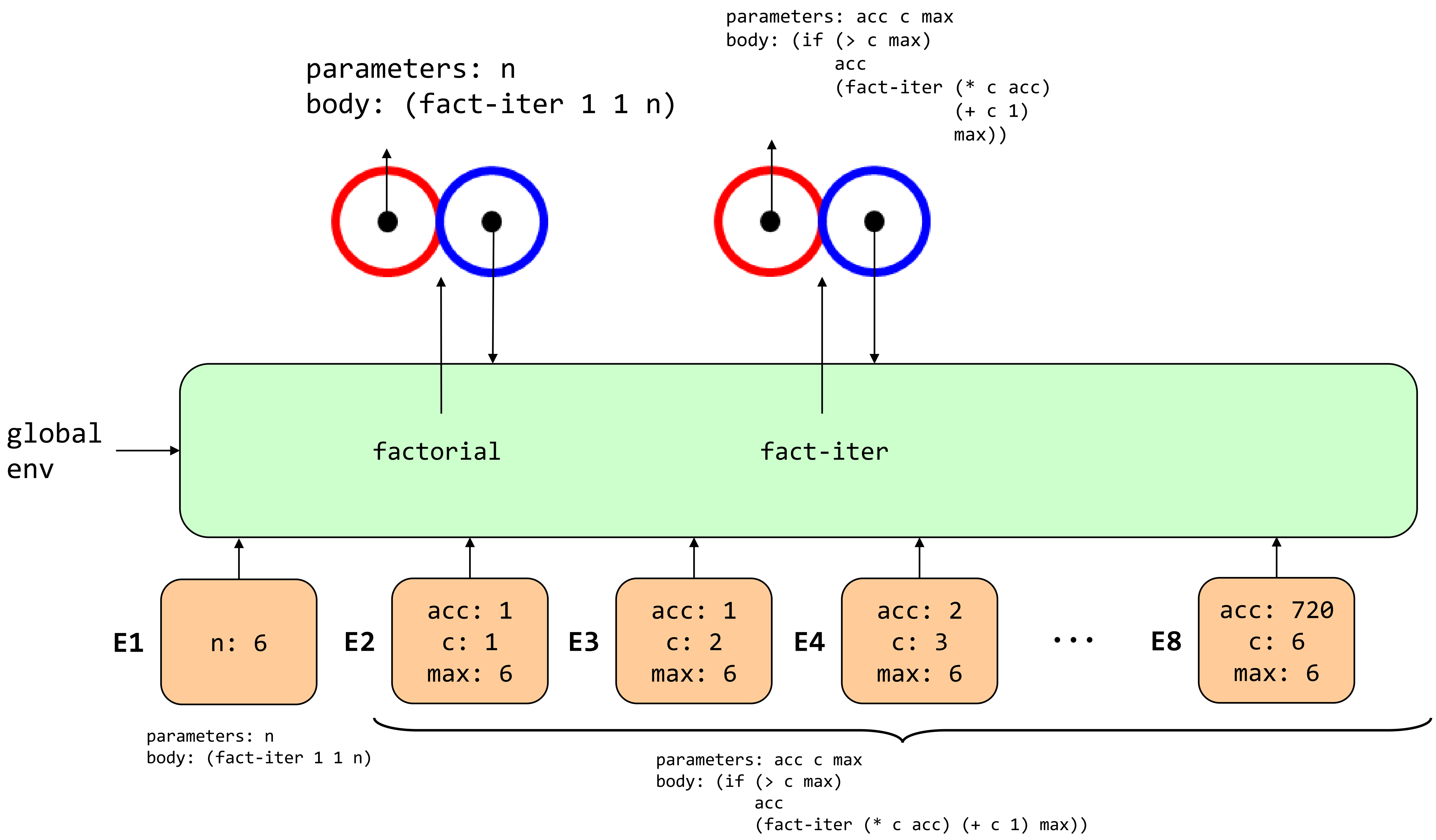
Show the environment structures created by evaluating `(factorial 6)` using each version of the `factorial` procedure.[14]



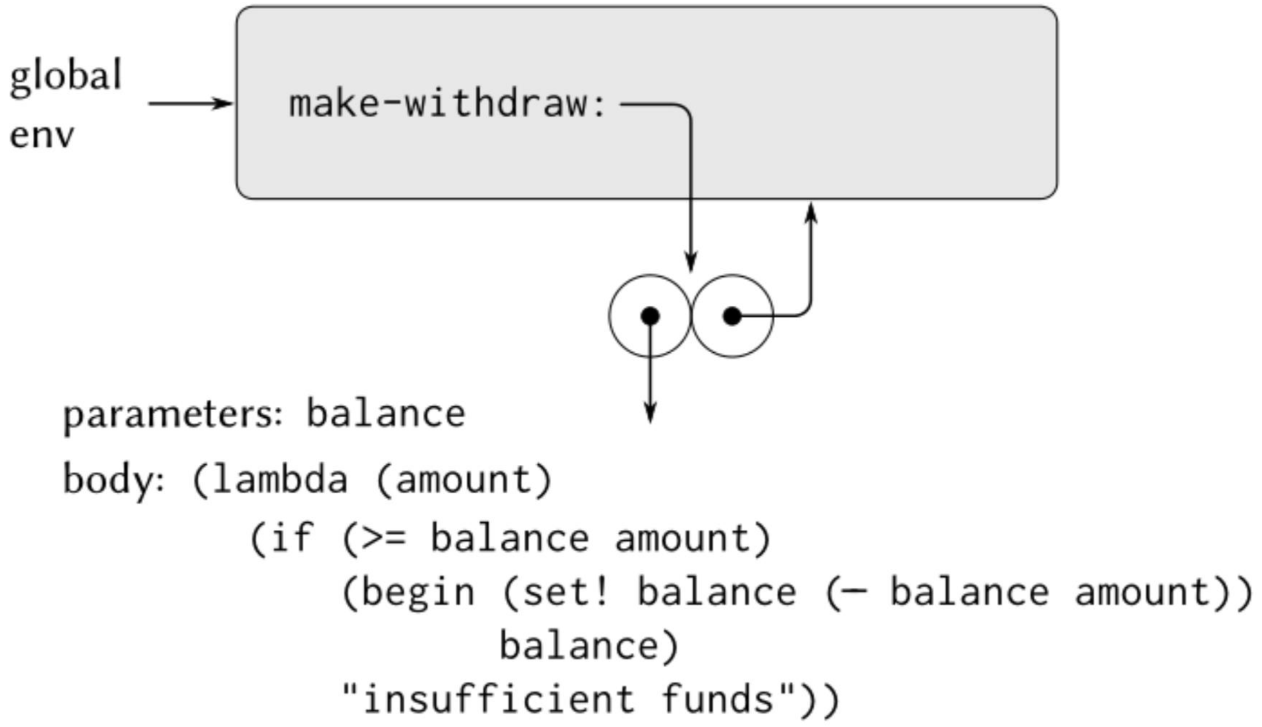
```
parameters: n
body: (if (= n 1)
          1
          (factorial (- n 1)))
```


global
env
factorial
E1
n: 6
E2
n: 5
...
E6
n: 1
parameters: n
body: (if (= n 1)
1
(factorial (- n 1)))
parameters: n
body: (if (= n 1)
1
(factorial (- n 1)))
parameters: n
body: (if (= n 1)
1
(factorial (- n 1)))
parameters: n
body: (if (= n 1)
1
(factorial (- n 1)))


parameters: n
body: (fact-iter 1 1 n)
parameters: acc n max
body: (if (> c max)
acc
(fact-iter (* c acc)
(+ c 1)
max))
global env
factorial
fact-iter


parameters: acc c max
body: (if (> c max)
acc
(fact-iter (* c acc)
(+ c 1)
max))
parameters: n
body: (fact-iter 1 1 n)
global
env
factorial
fact-iter
E1
n: 6
E2
acc: 1
c: 1
max: 6
E3
acc: 1
c: 2
max: 6
E4
acc: 2
c: 3
max: 6
...
E8
acc: 720
c: 6
max: 6
parameters: n
body: (fact-iter 1 1 n)
parameters: acc c max
body: (if (> c max)
acc
(fact-iter (* c acc) (+ c 1) max))

```
(define (make-withdraw balance)
  (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"))
```

Figure 3.6: Result of defining `make-withdraw` in the global environment.

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

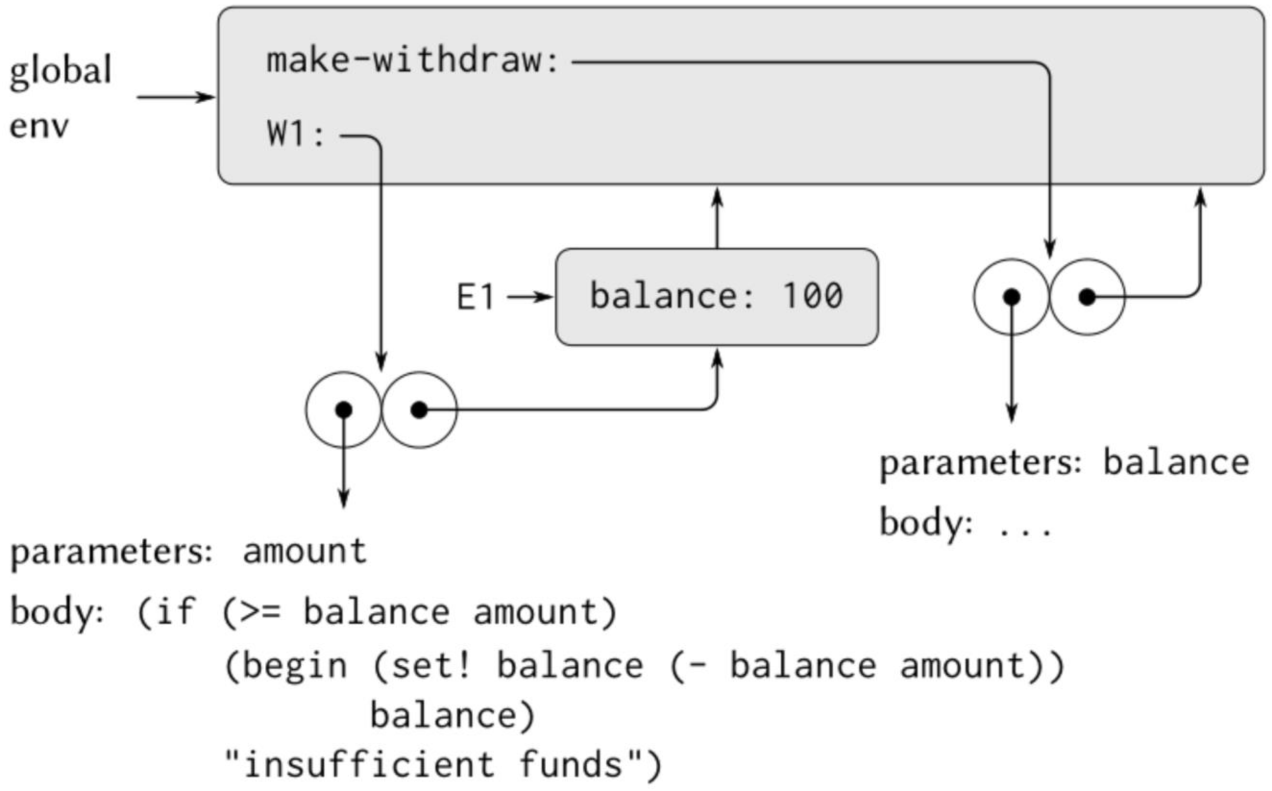


Figure 3.7: Result of evaluating `(define W1 (make-withdraw 100))`.

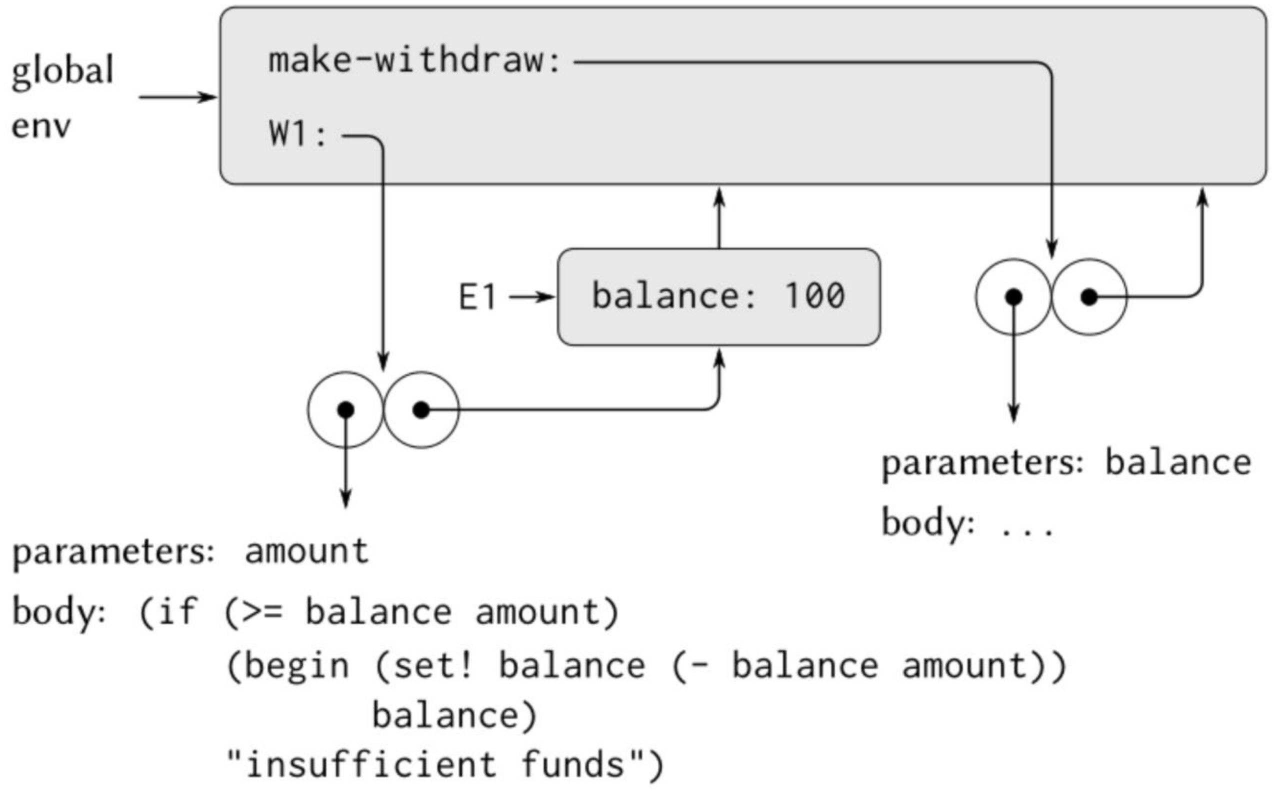
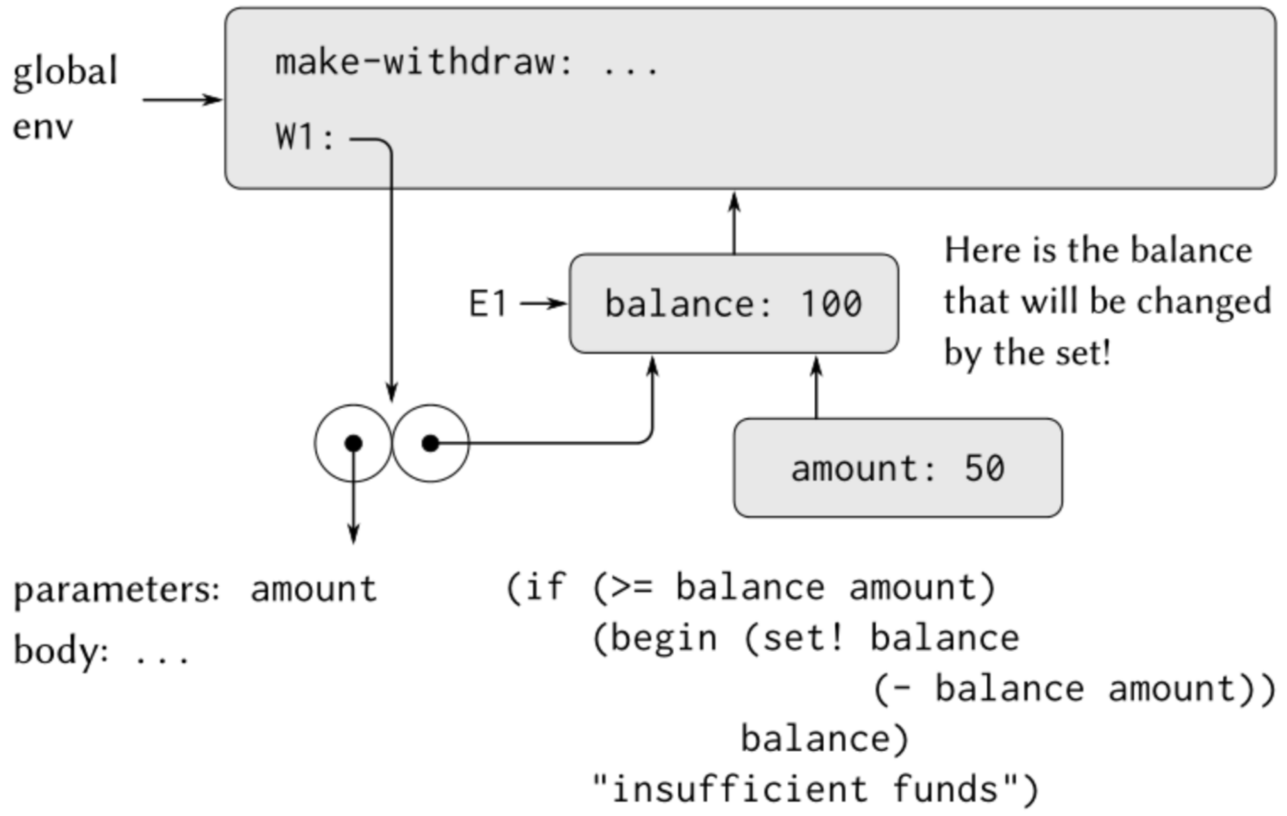


Figure 3.7: Result of evaluating `(define W1 (make-withdraw 100))`.

```
(W1 50)
50
```



```
(if (>= balance amount)
    (begin (set! balance
                 (- balance amount))
           balance)
    "insufficient funds")
```

Figure 3.8: Environments created by applying the procedure object W1.

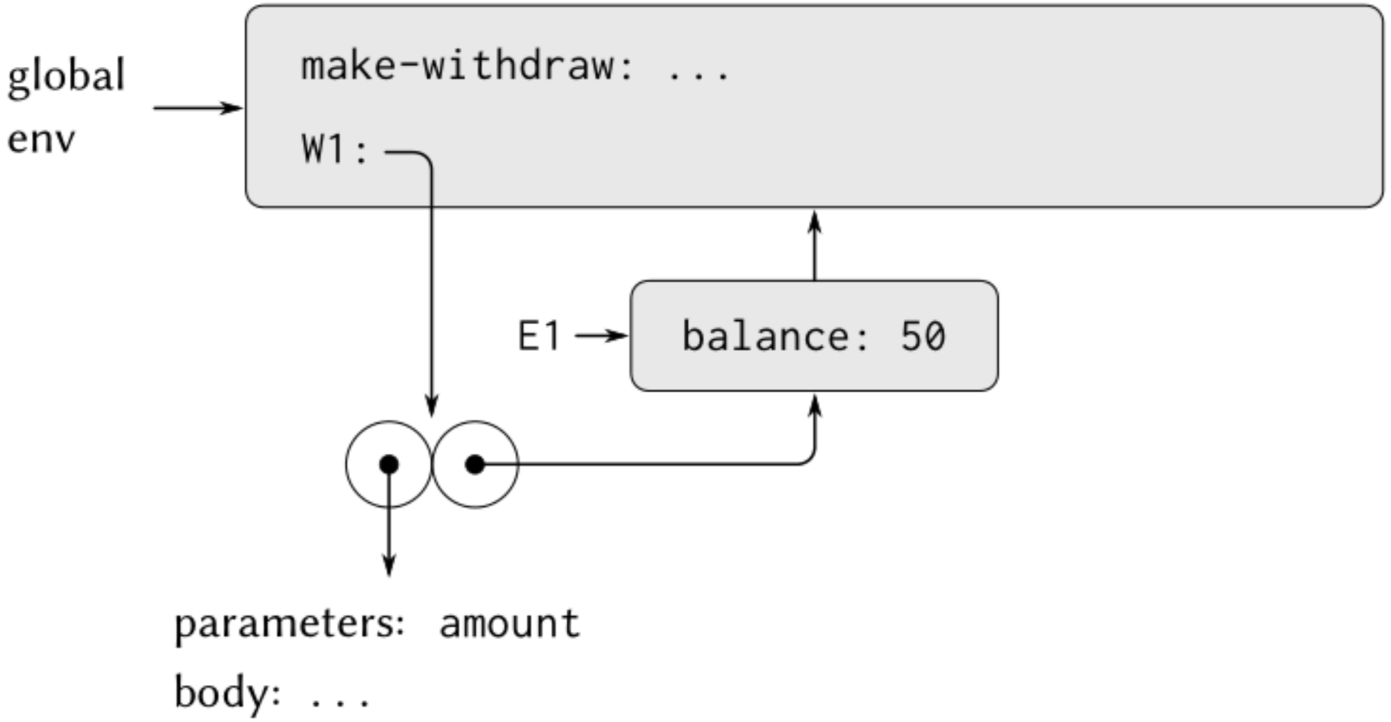


Figure 3.9: Environments after the call to `W1`.

```
(define W2 (make-withdraw 100))
```

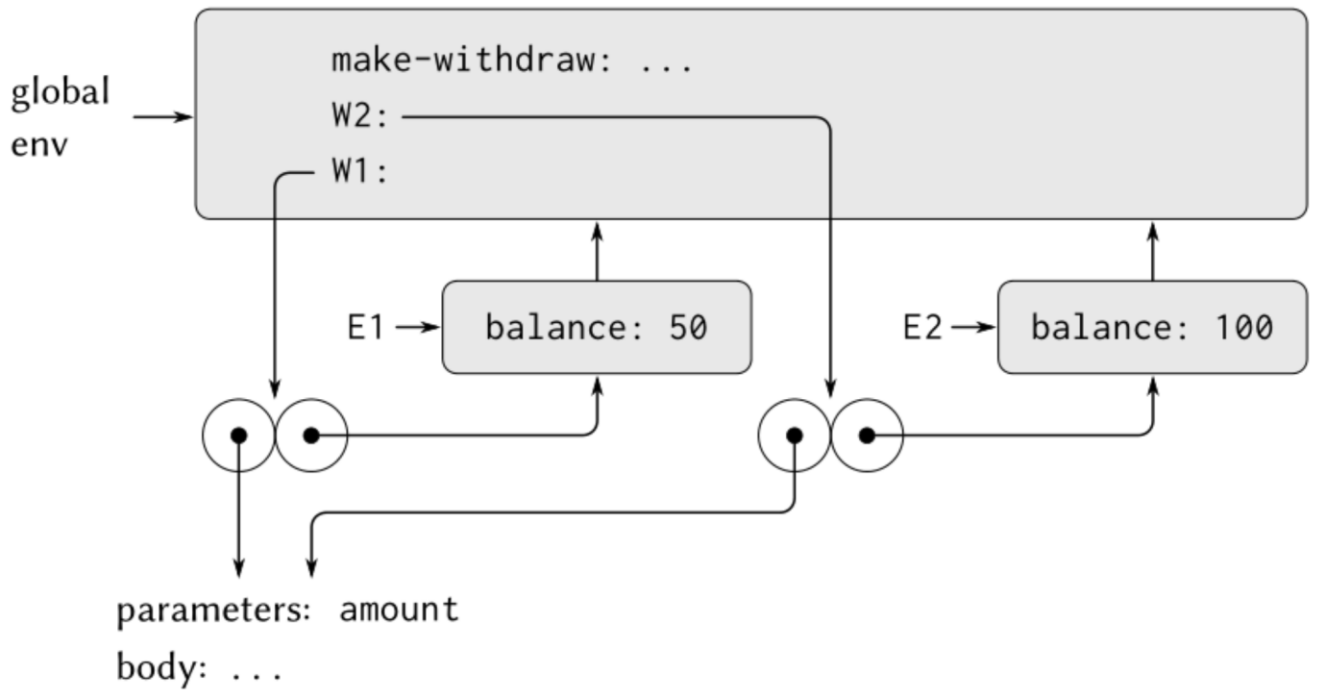


Figure 3.10: Using `(define W2 (make-withdraw 100))` to create a second object.

Exercise 3.10: In the `make-withdraw` procedure, the local variable `balance` is created as a parameter of `make-withdraw`. We could also create the local state variable explicitly, using `let`, as follows:

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

Recall from Section 1.3.2 that `let` is simply syntactic sugar for a procedure call:

```
(let ((⟨var⟩ ⟨exp⟩)) ⟨body⟩)
```

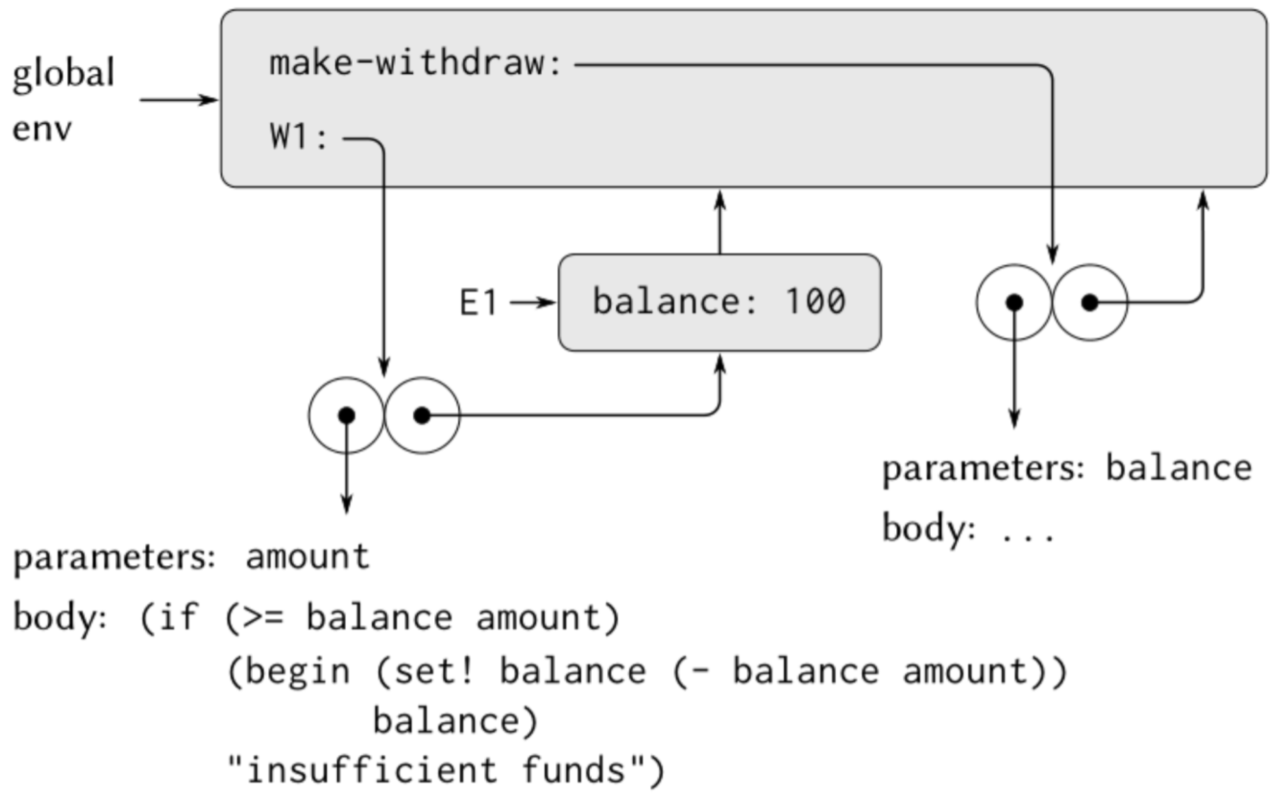
is interpreted as an alternate syntax for

```
((lambda (⟨var⟩) ⟨body⟩) ⟨exp⟩)
```

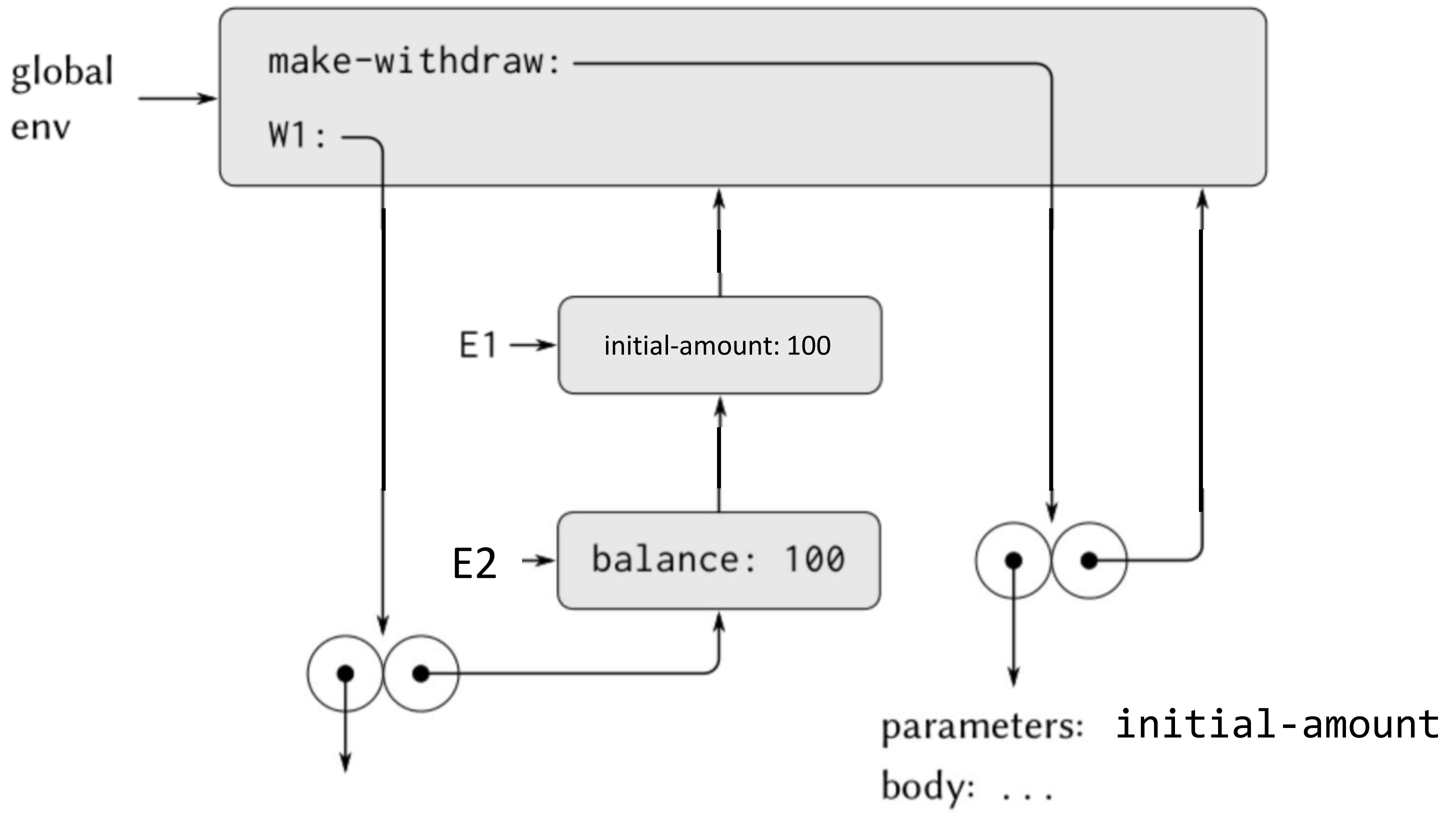
Use the environment model to analyze this alternate version of `make-withdraw`, drawing figures like the ones above to illustrate the interactions

```
(define W1 (make-withdraw 100))
(W1 50)
(define W2 (make-withdraw 100))
```

Show that the two versions of `make-withdraw` create objects with the same behavior. How do the environment structures differ for the two versions?



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



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

```
(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))
```

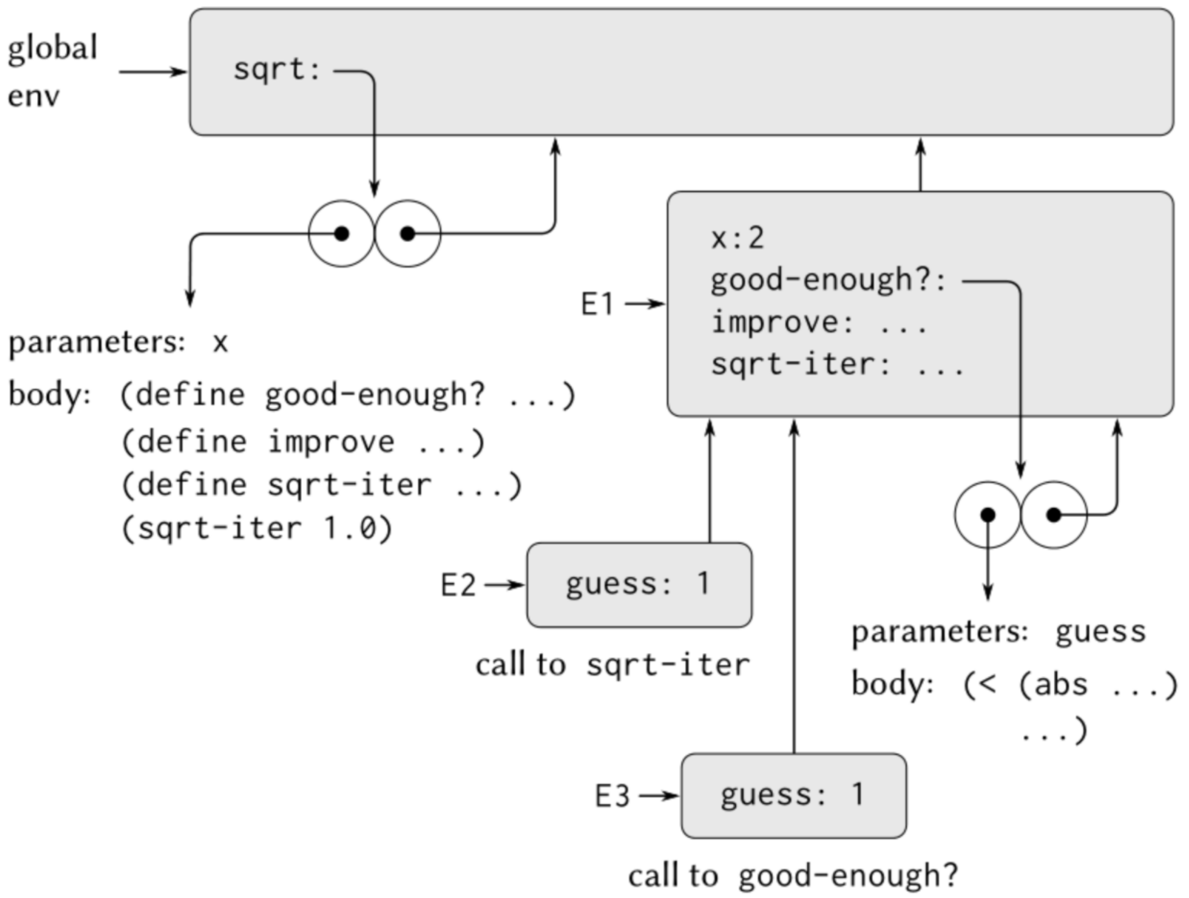


Figure 3.11: `sqrt` procedure with internal definitions.

The environment model thus explains the two key properties that make local procedure definitions a useful technique for modularizing programs:

- The names of the local procedures do not interfere with names external to the enclosing procedure, because the local procedure names will be bound in the frame that the procedure creates when it is run, rather than being bound in the global environment.
- The local procedures can access the arguments of the enclosing procedure, simply by using parameter names as free variables. This is because the body of the local procedure is evaluated in an environment that is subordinate to the evaluation environment for the enclosing procedure.

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

Java

Meetup