Chapter 3

Modularity, Objects and State

Topics of Chapters 1 & 2

	data	procedures
primitive	X	X
combinations	X	X
abstraction	X	X

But this is not sufficient for organizing large systems. Now we study modularity.

Chapter 3: Forms of Modularity

Organize a system
in a modular way

According to the objects that live in the system (viewing a system as a collection of objects)

Raises the linguistic issue of "state"

According to the streams of information that flow in the system

Raises the linguistic issue of "delayed evaluation"

Objects: Here's What we Want

Not a mathematical function anymore!

```
> (withdraw 25)
75
> (withdraw 25)
50
> (withdraw 60)
"Insufficient Funds"
> (withdraw 15)
35
```

It seems to "remember" stuff.

Two New Special Forms

(or not so new?)

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

Change the binding of an existing variable

```
(begin <exp1> <exp2> ... <expk>)
```

"One after the other" becomes meaningful

From now on, we leave the realm called functional programming and move on to imperative programming.

First Solution

Global variable

```
(define balance 100)

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

Having multiple accounts is problematic

There is no “protection”

Second Solution

local

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

but still only 1

protectio

```
> (new-withdraw 10)
90
> (set! balance 30000)
⊕ set!: cannot set variable before its definition: balance
```

Third Solution

parametrize

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

make-withdraw returns a lambda!

w1 and w2 are independent "objects"

The Full Example (cf. 3rd solution)

```scheme
(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 request" m))))
  dispatch)
```

That lambda "contains" the balance variable and 2 lambdas

make-account returns a lambda!

```scheme
> (define acc (make-account 100))
> ((acc 'withdraw) 50)
50
> ((acc 'withdraw) 60)
"Insufficient funds"
> ((acc 'deposit) 40)
90
> ((acc 'withdraw) 60)
30
> (define acc2 (make-account 100))
```

message

The Cost of Introducing Assignment

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

Compare these two under the substitution model of evaluation

```
((make-decrementer 25) 20)

⇒ ((lambda (amount)
      (- 25 amount)) 20)

⇒ (- 25 20)

⇒ 5
```

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

```
((make-simplified-withdraw 25) 20)

⇒ ((lambda (amount)
          (set! balance (- 25 amount))
          25) 20)

⇒ (set! balance 5)
   25

⇒ 25
```

This is plain wrong. The substitution model doesn't work anymore!

Functional vs. Imperative Programming

Every expression has a value.
Identifiers always have the same value

Imperative Programming

The trouble here is that substitution is based ultimately on the notion that the symbols in our language are essentially names for values. But as soon as we introduce set! and the idea that the value of a variable can change, a variable can no longer be simply a name. Now a variable somehow refers to a place where a value can be stored, and the value stored at this place can change.

Sameness and Change

When are two things "the same" ?

```
(define d1 (make-decrementer 25))
(define d2 (make-decrementer 25))
```

d1 and d2 have the same computational behaviour. Hence, they are "the same"

```
(define w1 (make-simplified-withdraw 25))
(define w2 (make-simplified-withdraw 25))

> (w1 20)
5
> (w1 20)
-15
> (w2 20)
5
```

w1 and w2 have the different computational behaviour. Hence, they are not "the same"

Functional Programming Languages

A languages that supports the concept that "equals can always be substituted for equals" in an expression without changing the value of the expression is said to be referentially transparent. Referential transparency is violated when we include set! in our computer language.

Scheme is NOT an FPL!

Pitfalls of Imperative Programming

Functional variant

```
(define (factorial1 n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Order is not relevant

Imperative variant

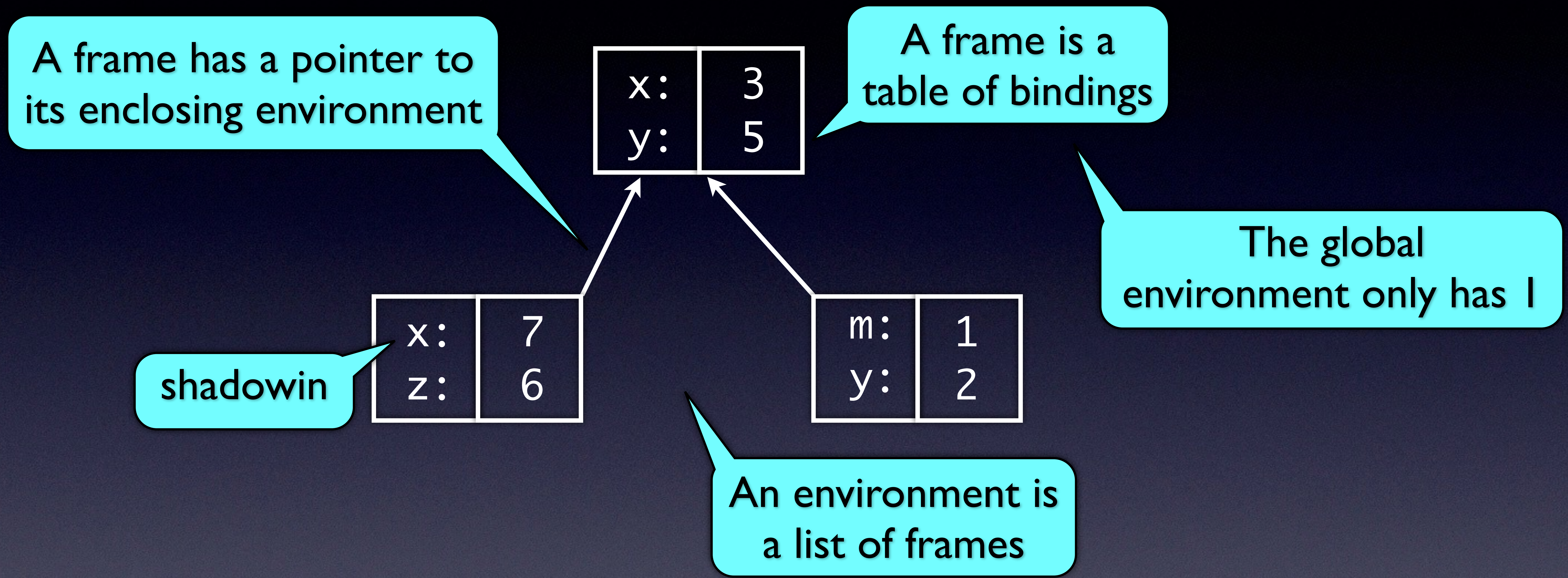
```
(define (factorial2 n)
  (let ((product 1)
        (counter 2))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                 (set! counter (+ counter 1))
                 (iter))))
    (iter)))
```

The order becomes crucial: harder to think about!

Even worse in concurrent programs

The Environment Model of Evaluation

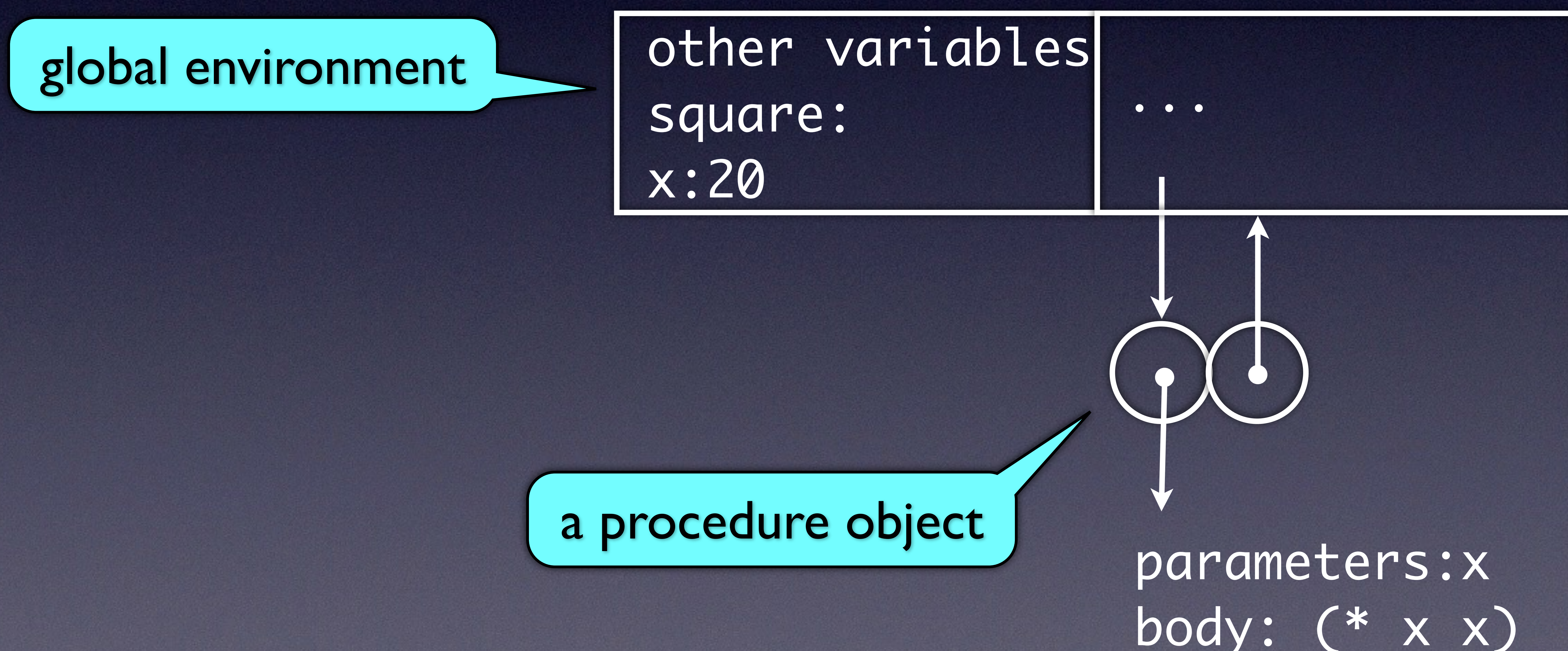
An improved mental model to explain Scheme's behaviour



A variable is no longer a name for a value, but a place in which values can be "stored". The value of a variable with respect to an environment is the value given by the binding of the variable in the first frame of the environment that contains a binding for that variable.

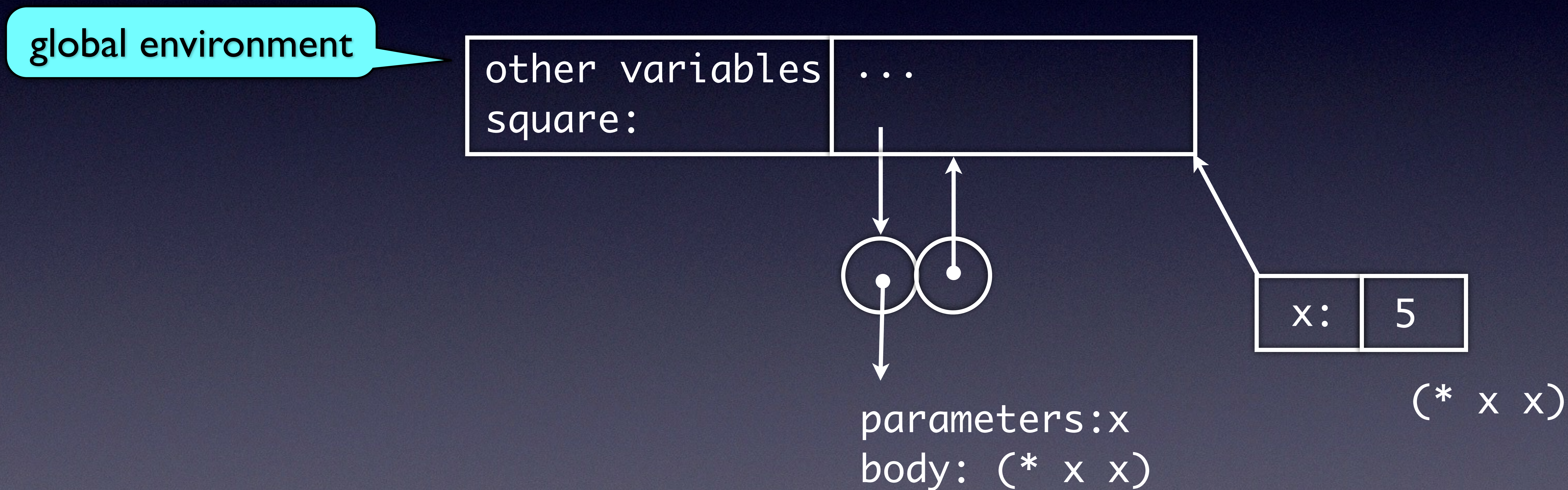
Procedure Creation

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

(define x 20)
```



Procedure Application

(square 5)



Evaluation Rules: Version 2

To evaluate an expression w.r.t. an environment:

changed

- numerals evaluate to numbers
- identifiers evaluate to their value in the environment
- combinations:
  - evaluate all the subexpressions in the combination in the environment
  - apply the procedure that is the value of the leftmost expression (= the operator) to the arguments that are the values of the other expressions (= the operands)
- some expressions (e.g. define) have a specialized evaluation rule.

These are called special forms.

Evaluation Rules: Version 2 (ctd)

A procedure is <u>applied</u> 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.

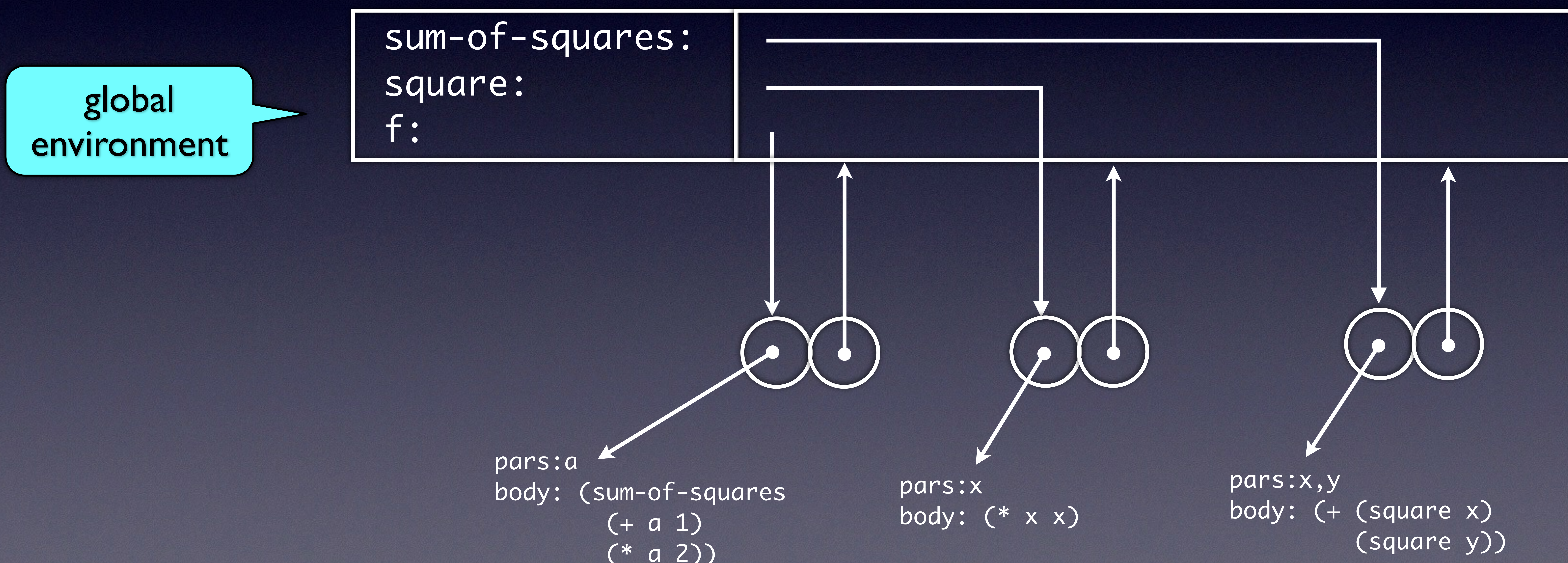
A procedure is <u>created</u> by evaluating a lambda expression relative to a given environment. The resulting procedure object is a pair consisting of the text of the lambda expression and a pointer to the environment in which the procedure was created.

Evaluating the expression (set! <var> <value>) in some environment locates the binding of the variable in the environment and changes that binding to indicate the new value.

Example from Chapter 1: Creation

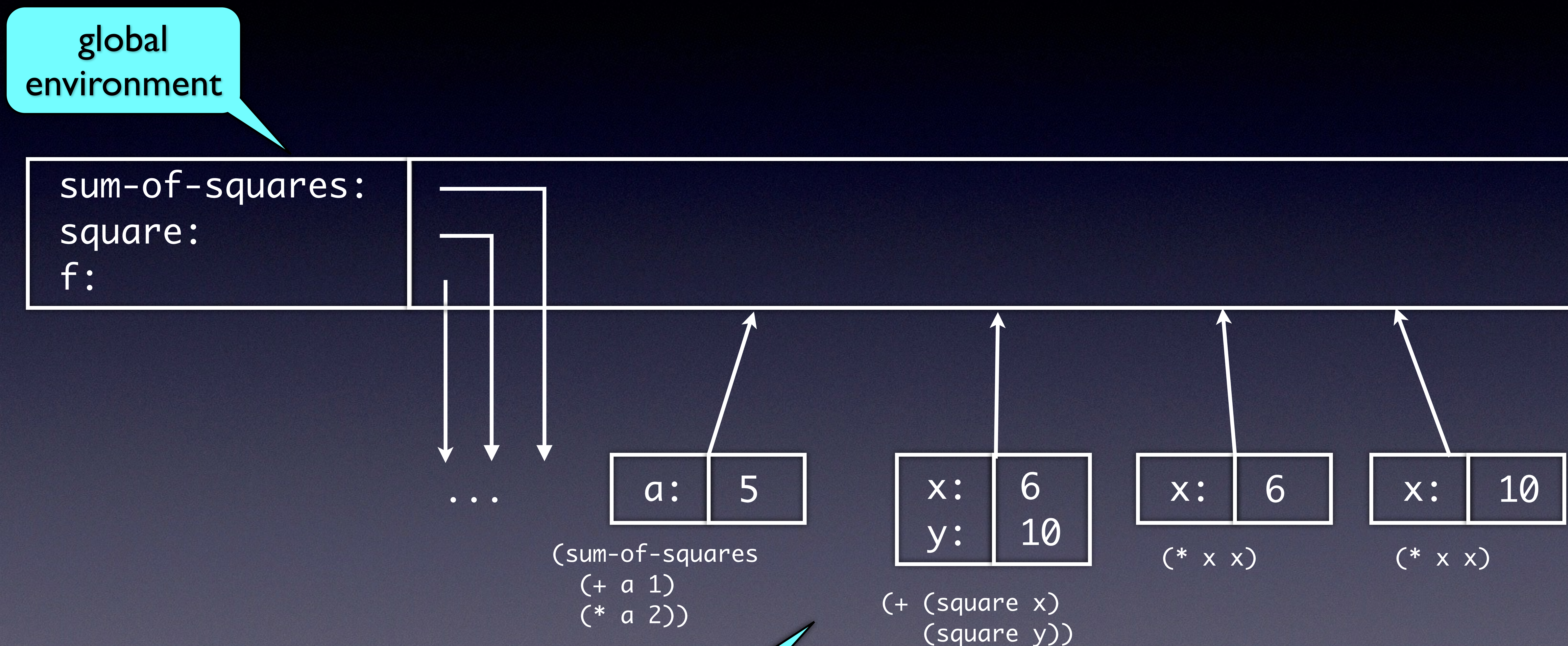
c.f. Substitution Model

```
> (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)))
```



Example from Chapter 1: Application

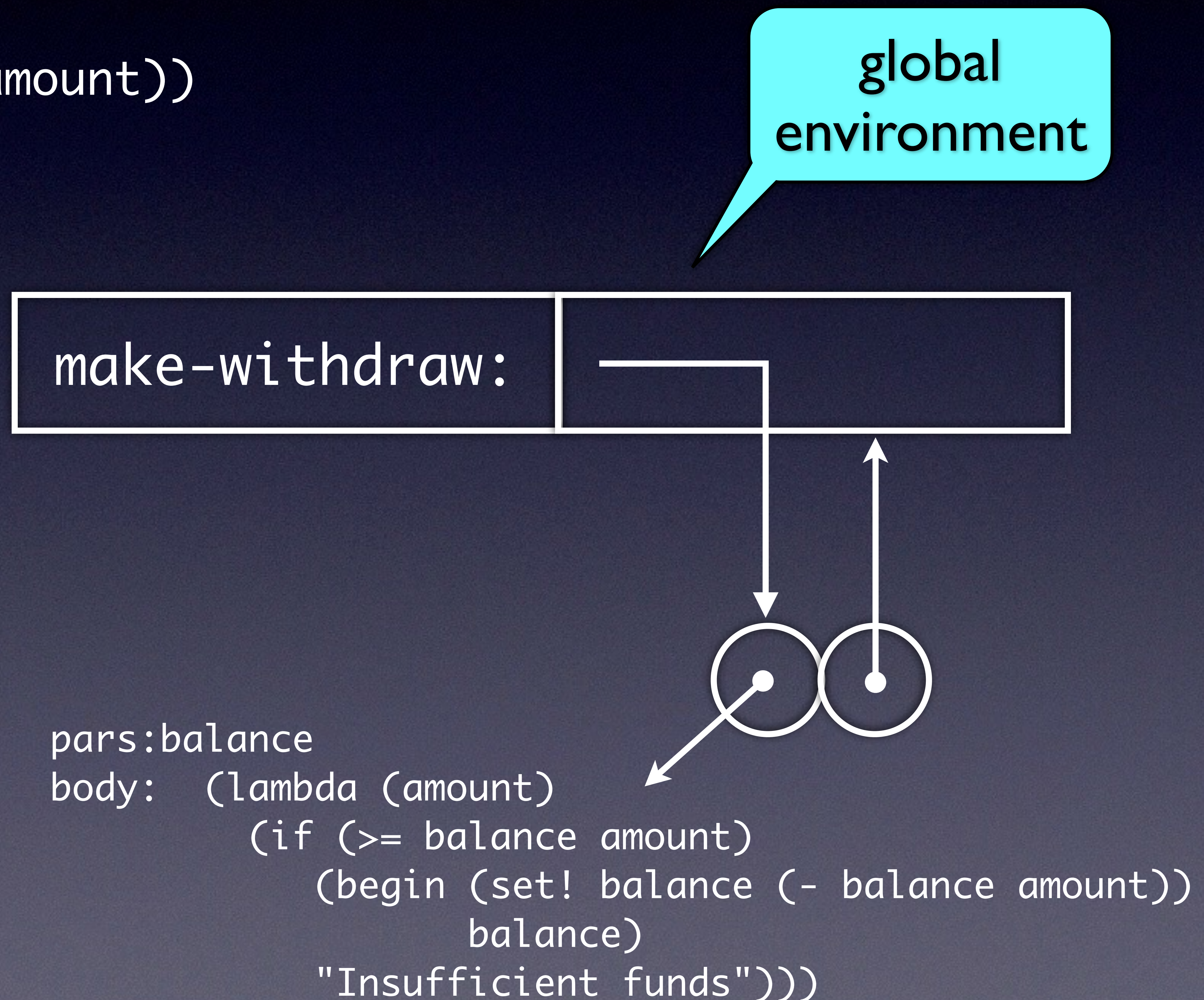
```
> (f 5)
```



each call creates a new environment!

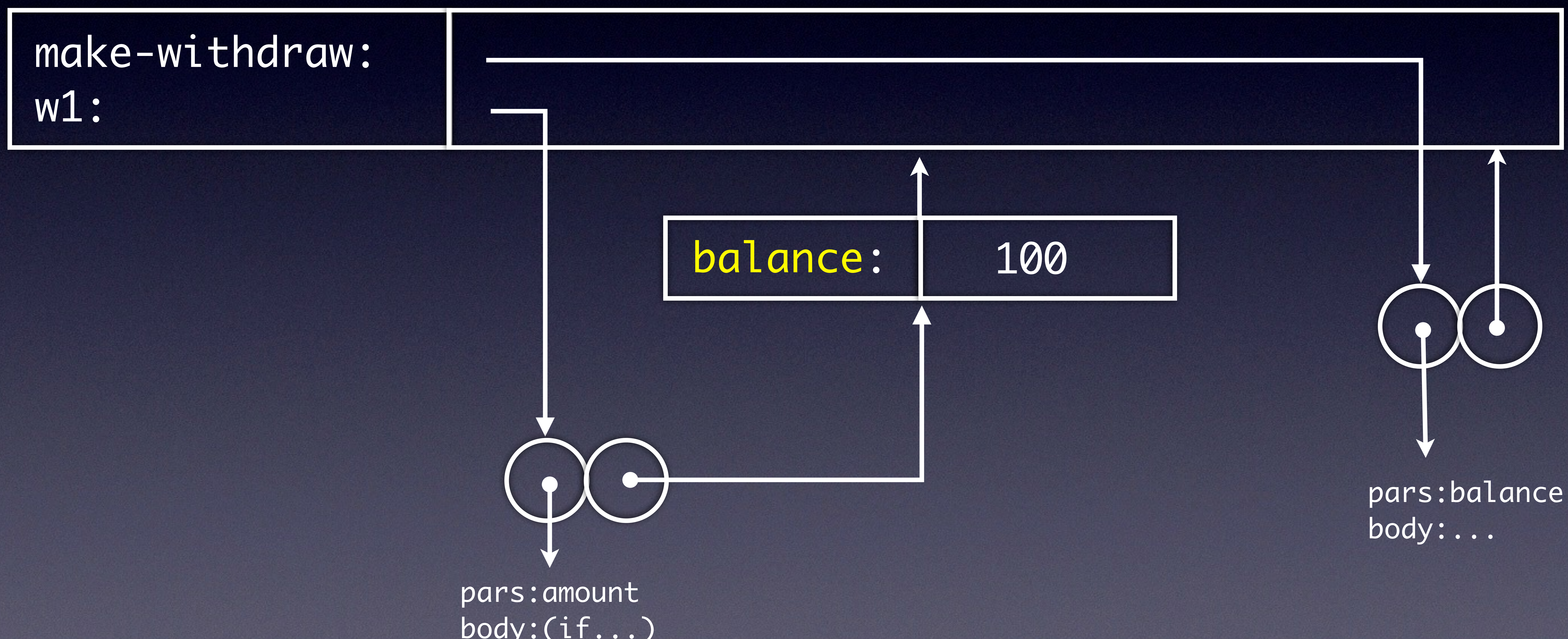
Objects with Local State (1/4)

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



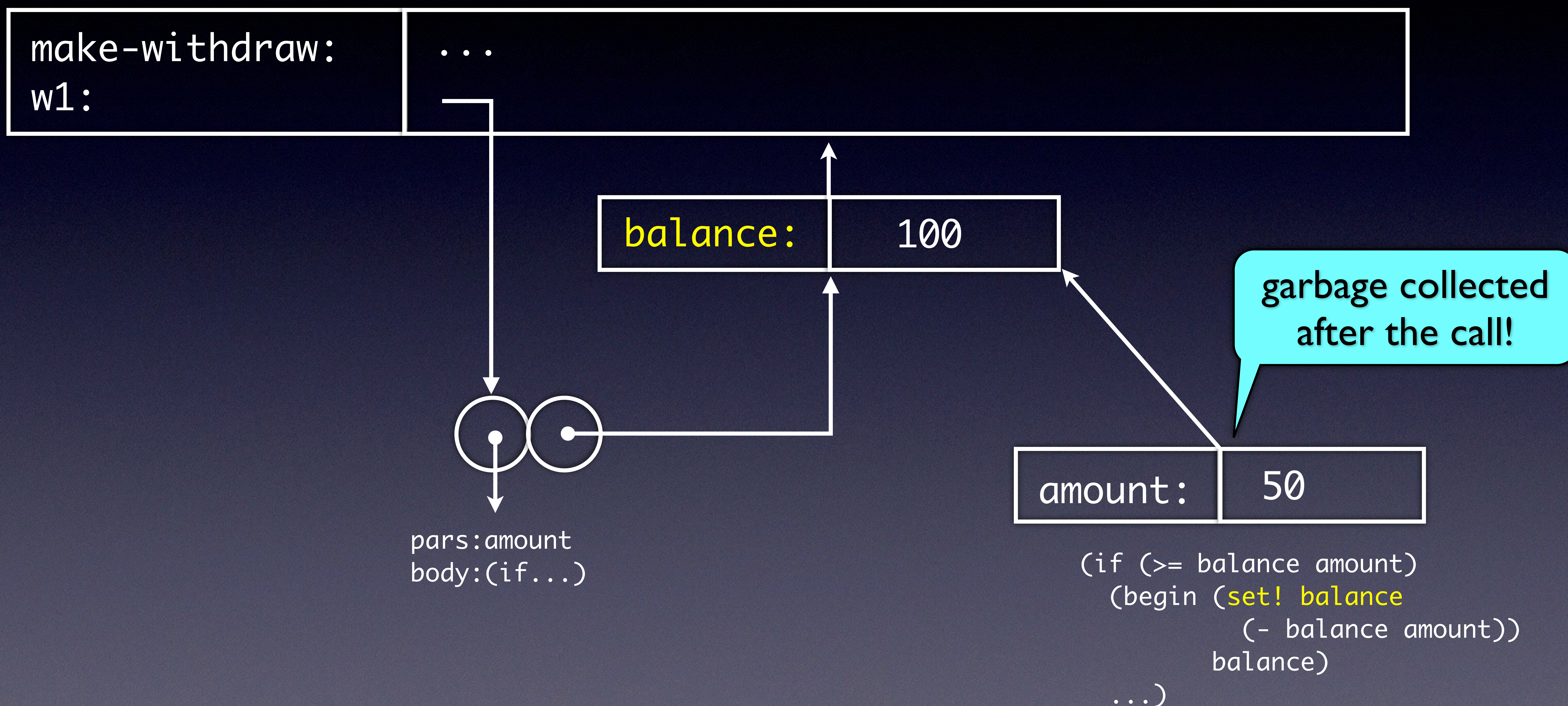
Objects with Local State(2/4)

```
(define w1 (make-withdraw 100))
```



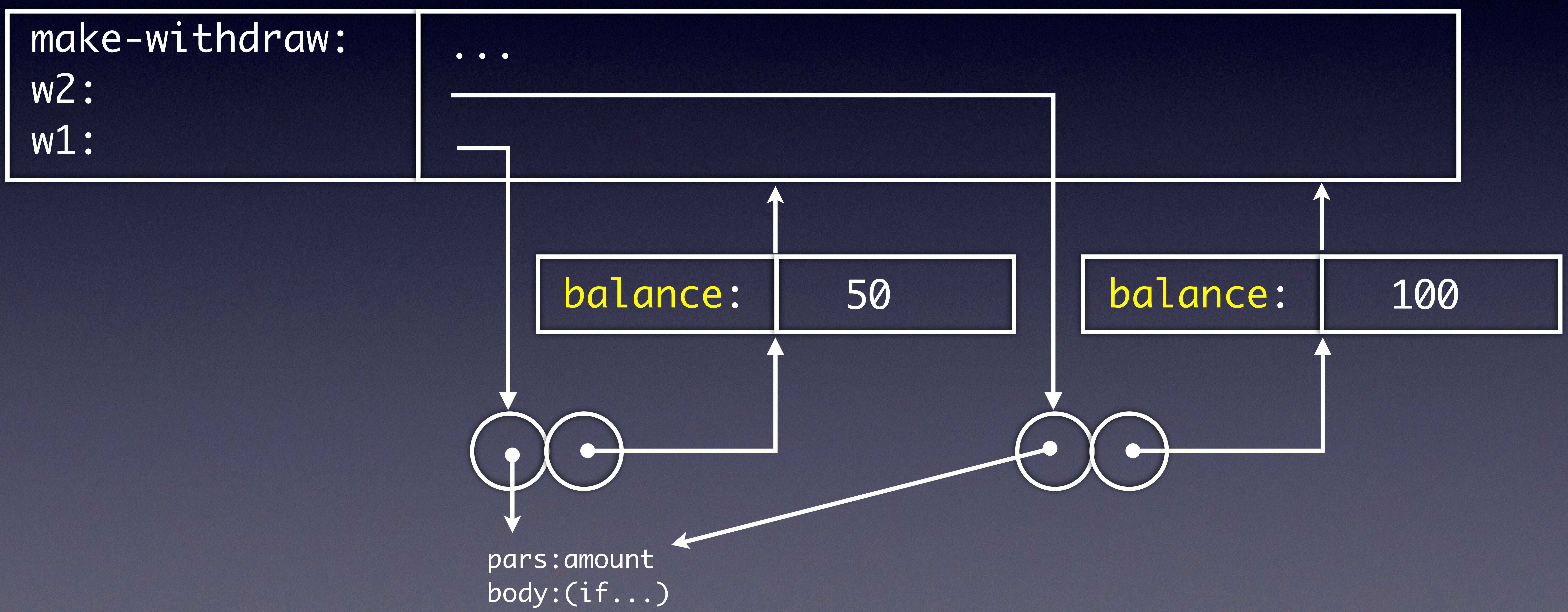
Objects with Local State(3/4)

```
(w1 50)
```



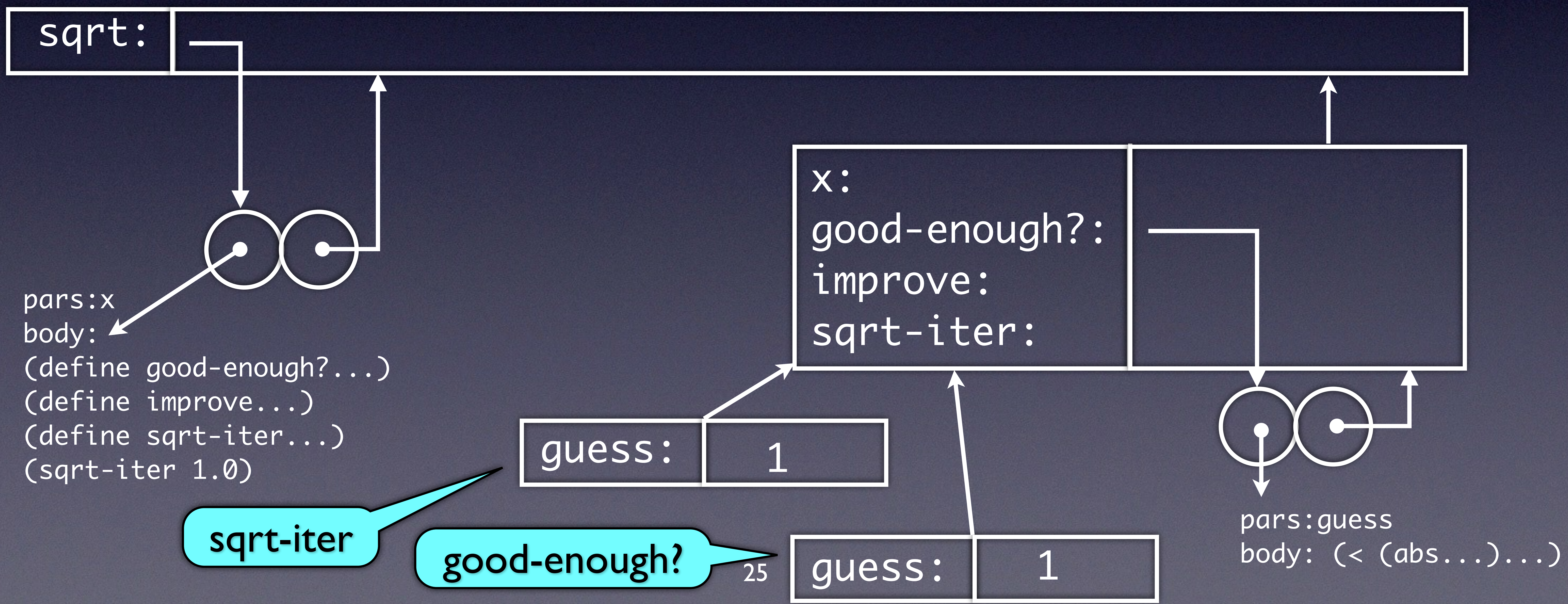
Objects with Local State(4/4)

```
(define w2 (make-withdraw 100))
```



Internal Definitions

```
(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))
```



Environment Model Advantages

The environment model explains 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.

Adding Another Dimension

	data	procedures
primitive	X	X
combinations	X	X
abstraction	X	X

Let's now investigate the interaction with mutable state.

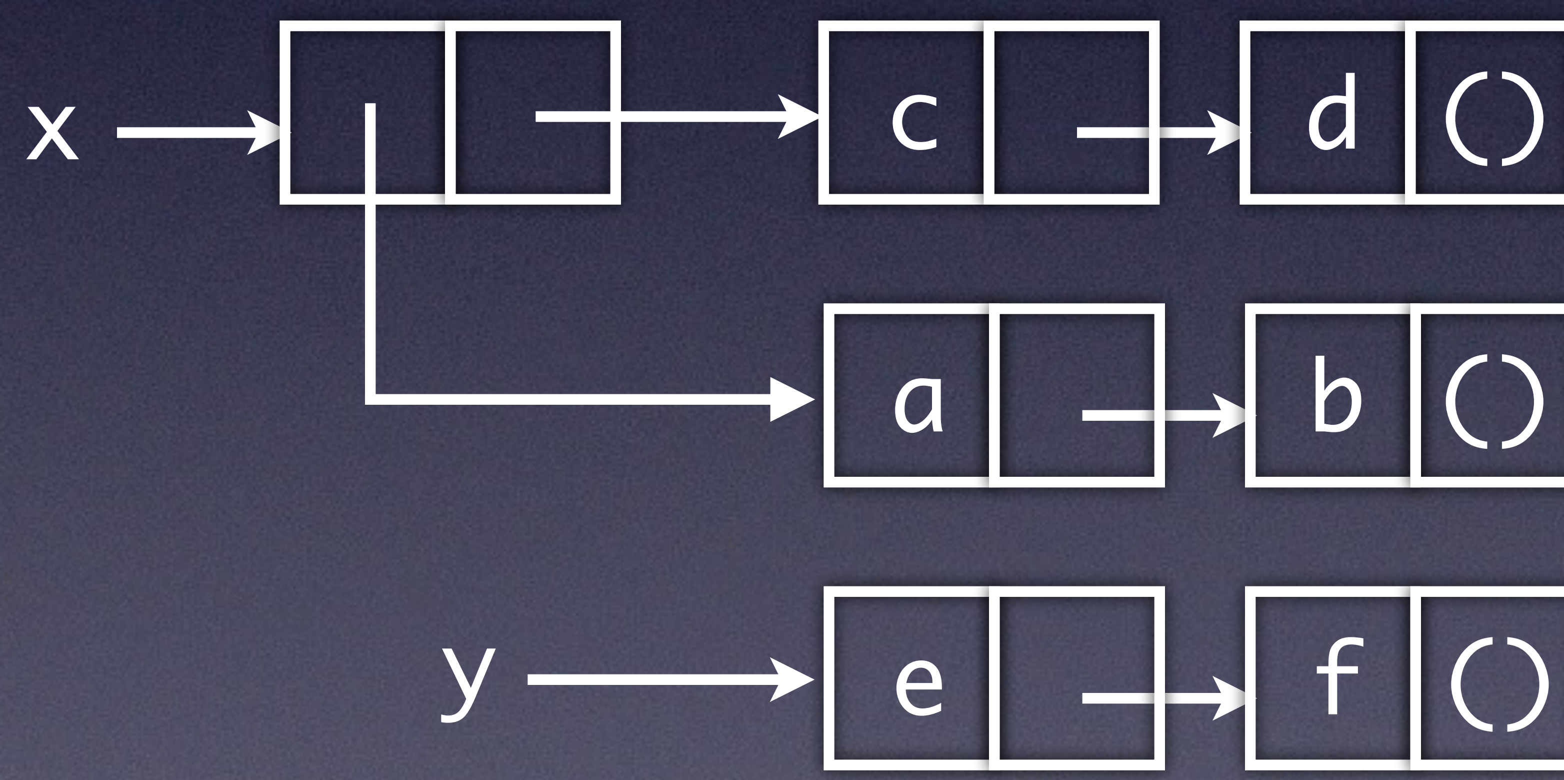
Add Two Primitives

Modify existing pairs

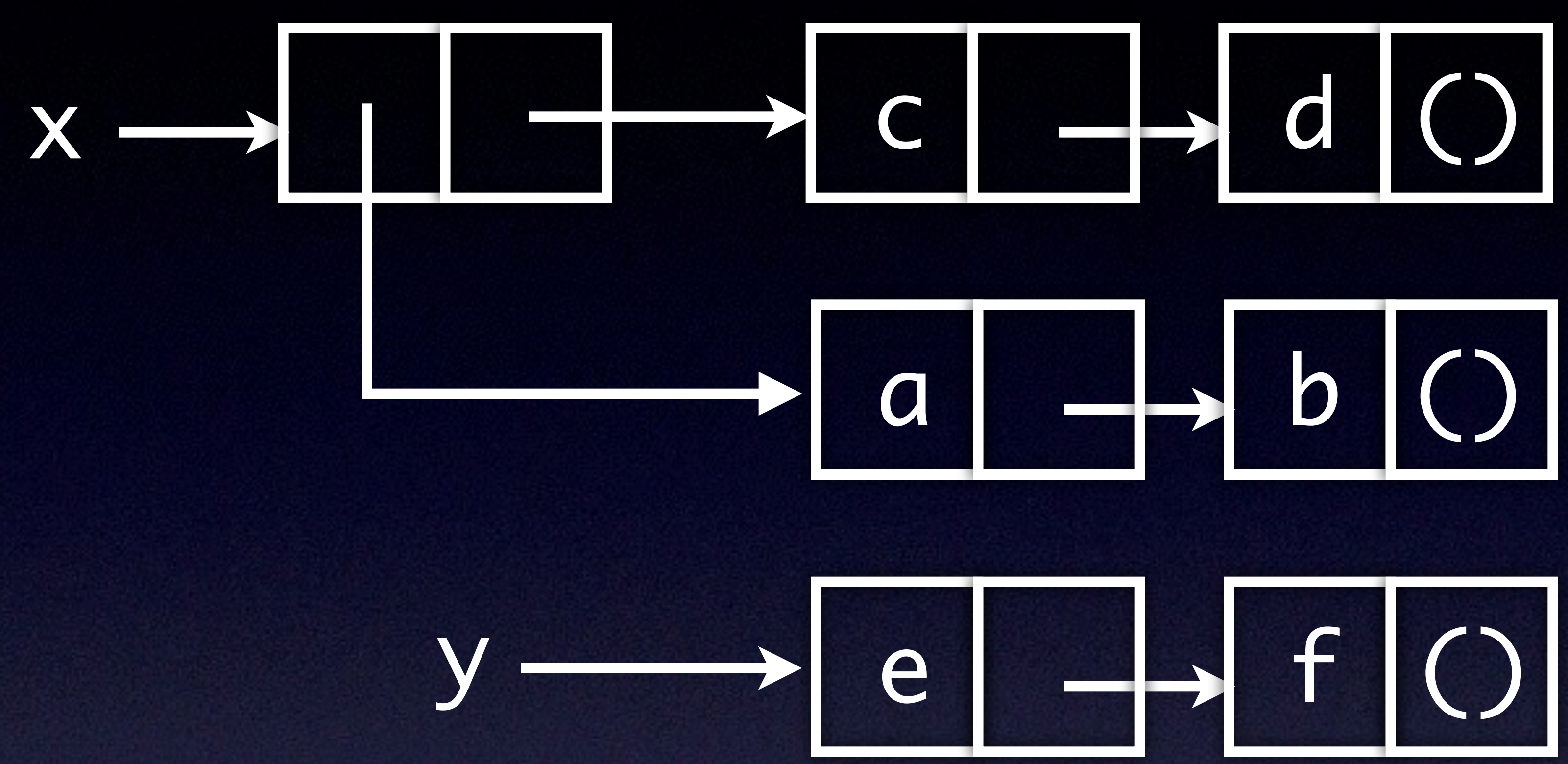
```
(set-car! <pair> <value>)
```

```
(set-cdr! <pair> <value>)
```

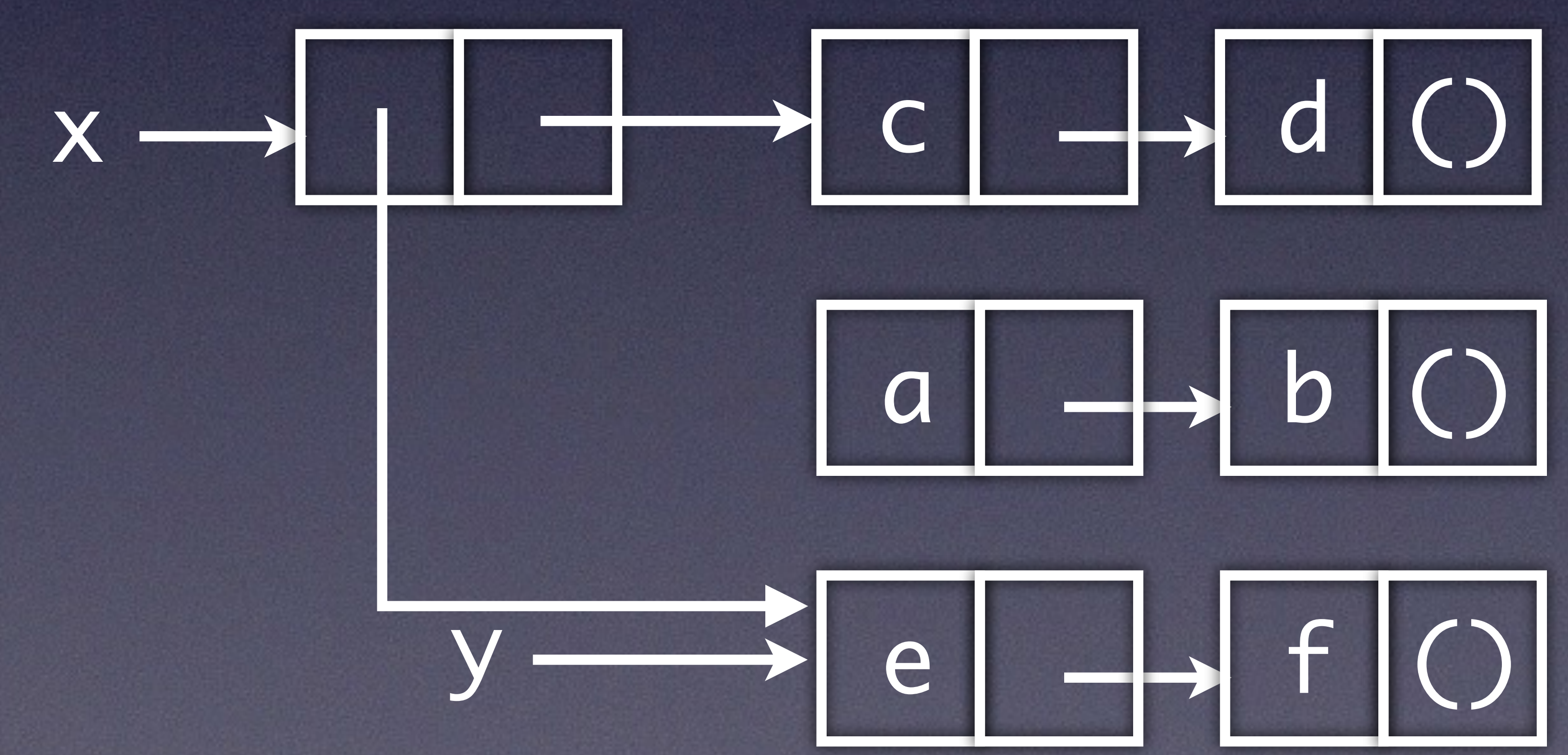
```
(define x ‘((a b) c d))

(define y ‘(e f))
```



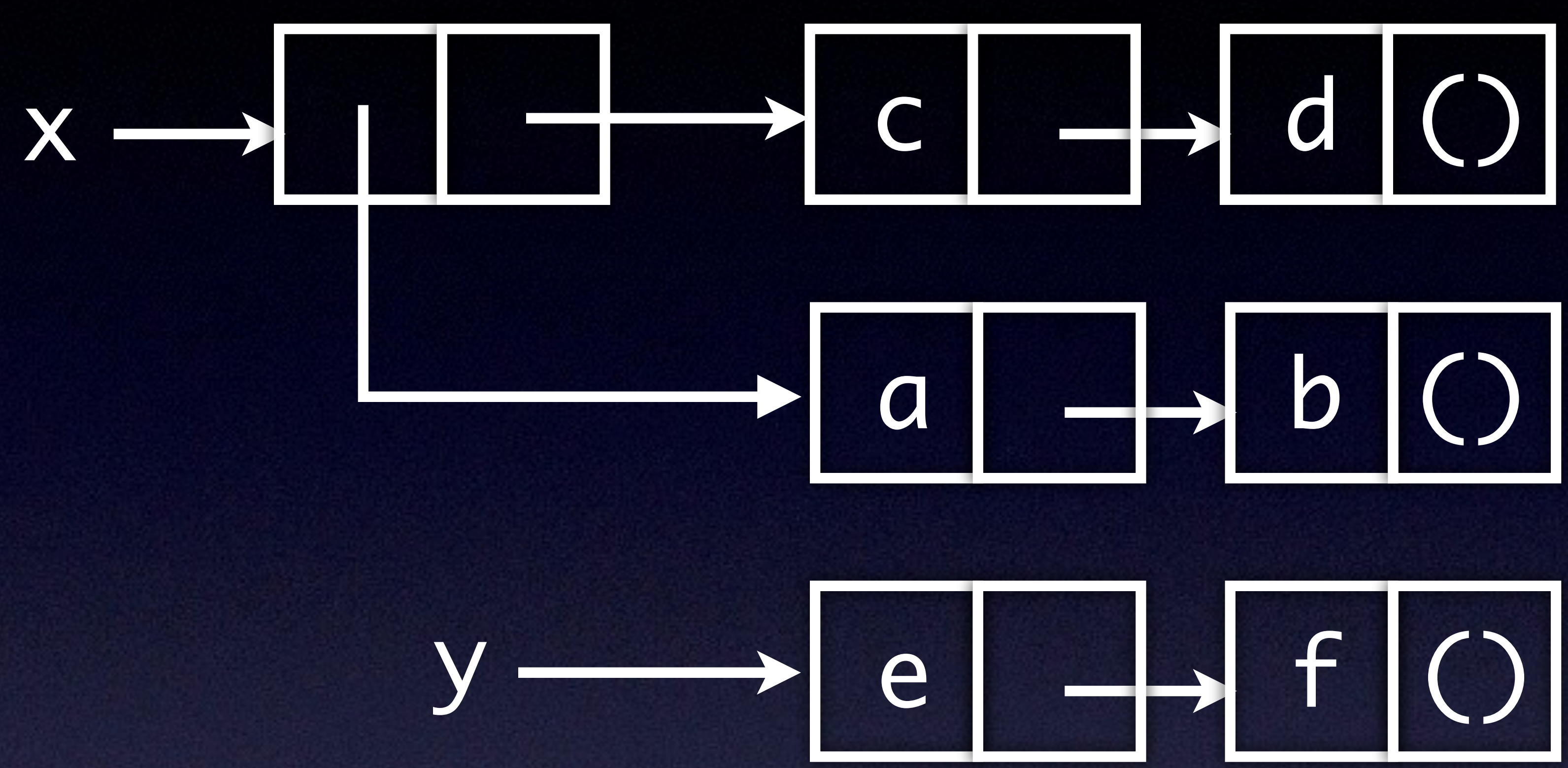
Example 1



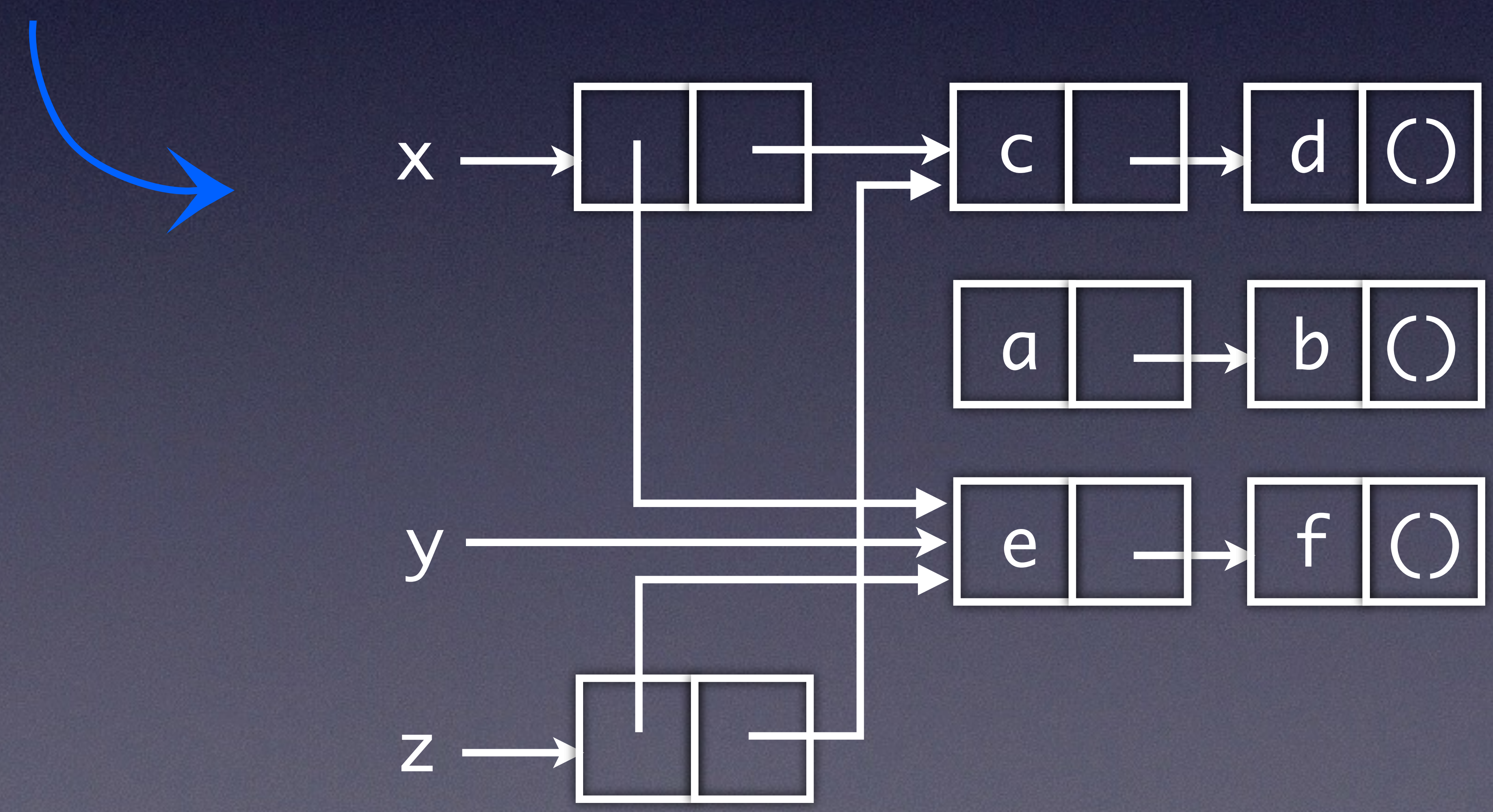
`(set-car! x y)`



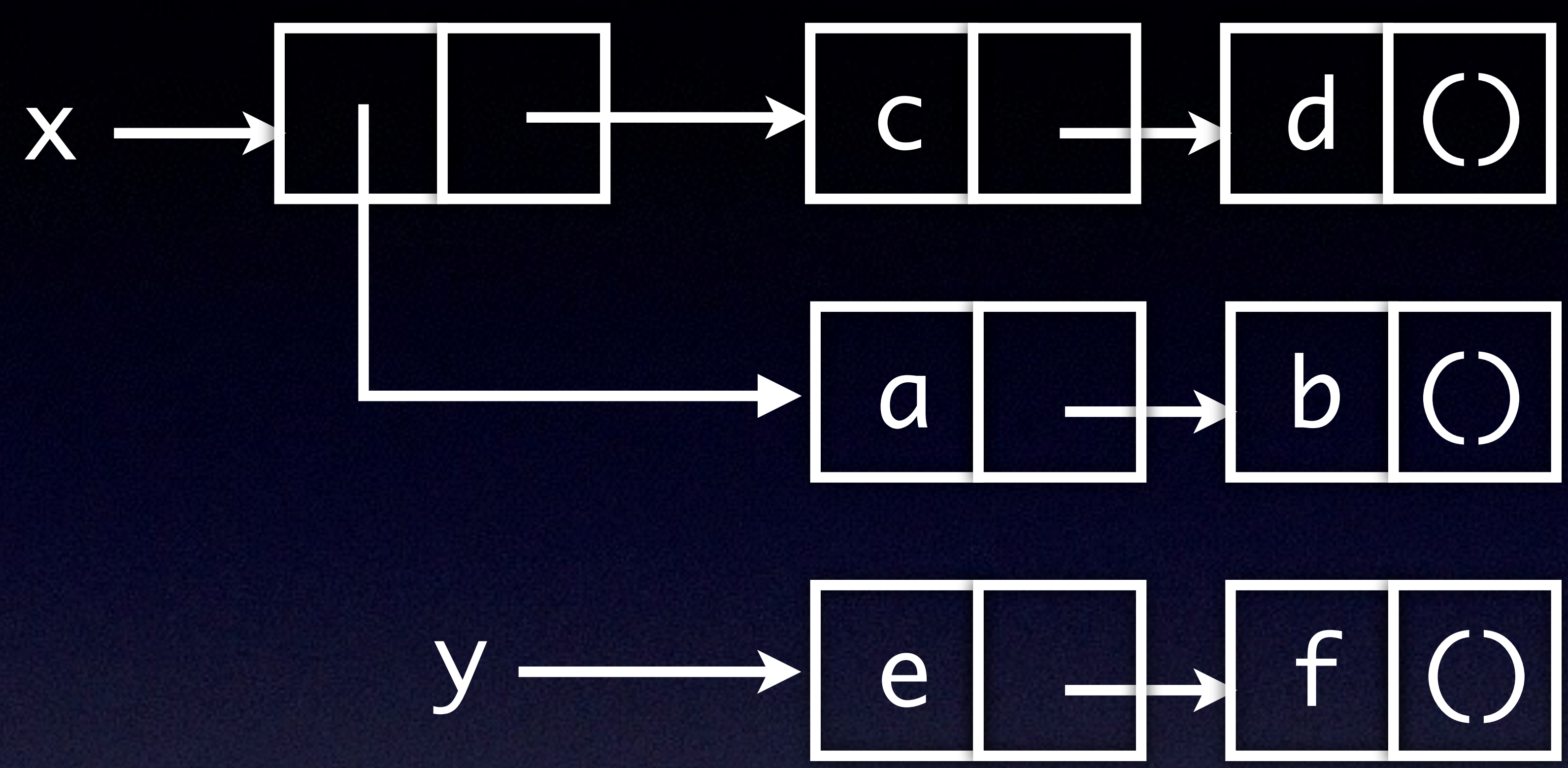
Example 2



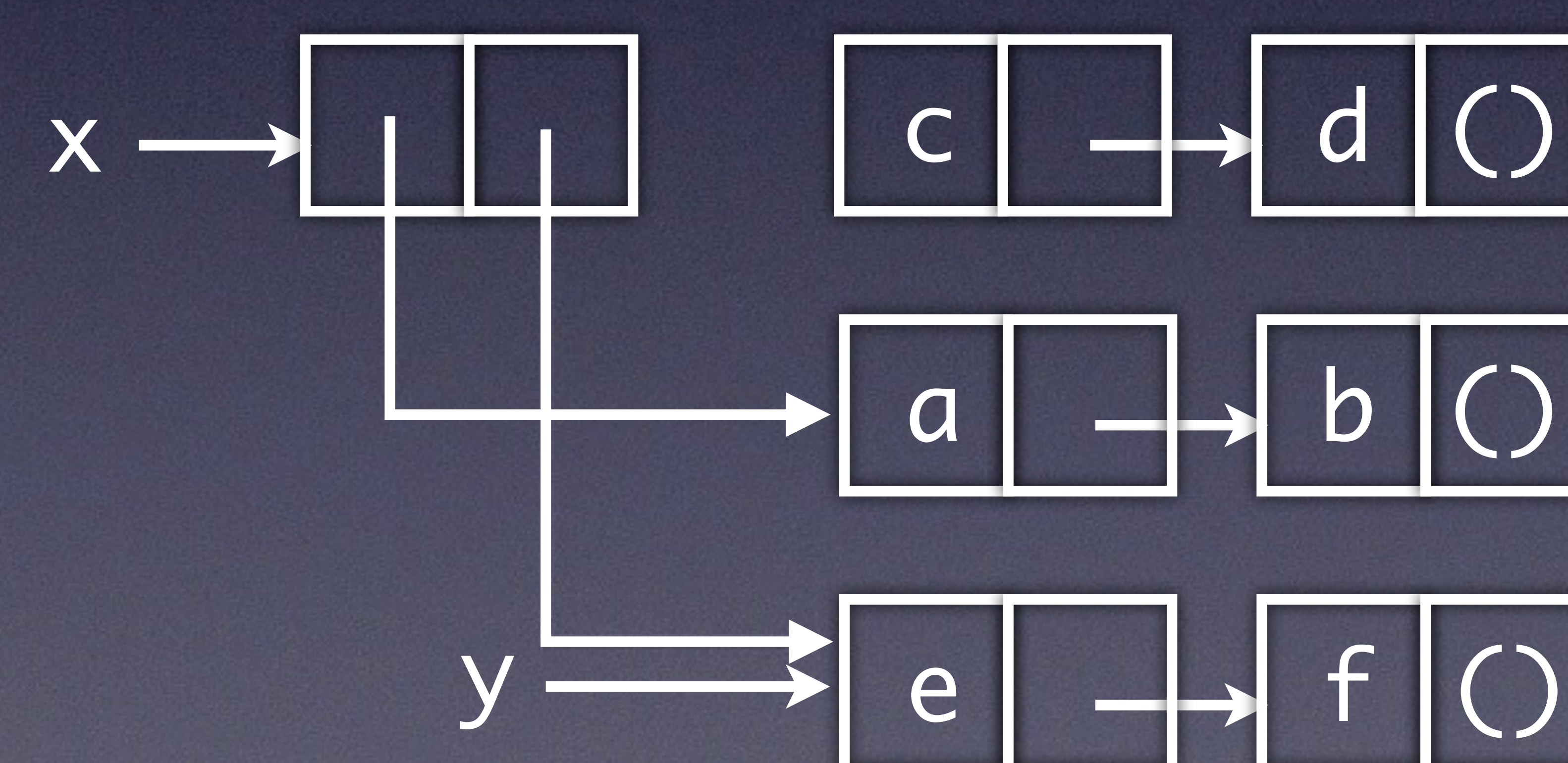

```
(define z (cons y (cdr x)))
```



Example 3




```
(set-cdr! x y)
```



Case Study: Representing Queues

FIFO

Operation	Resulting Queue
`(define q (make-queue))`	
`(insert-queue! q 'a)`	a
`(insert-queue! q 'b)`	a b
`(delete-queue! q)`	b
`(insert-queue! q 'c)`	b c
`(insert-queue! q 'd)`	b c d
`(delete-queue! q)`	c d

front

rear

The Queue ADT

constructo

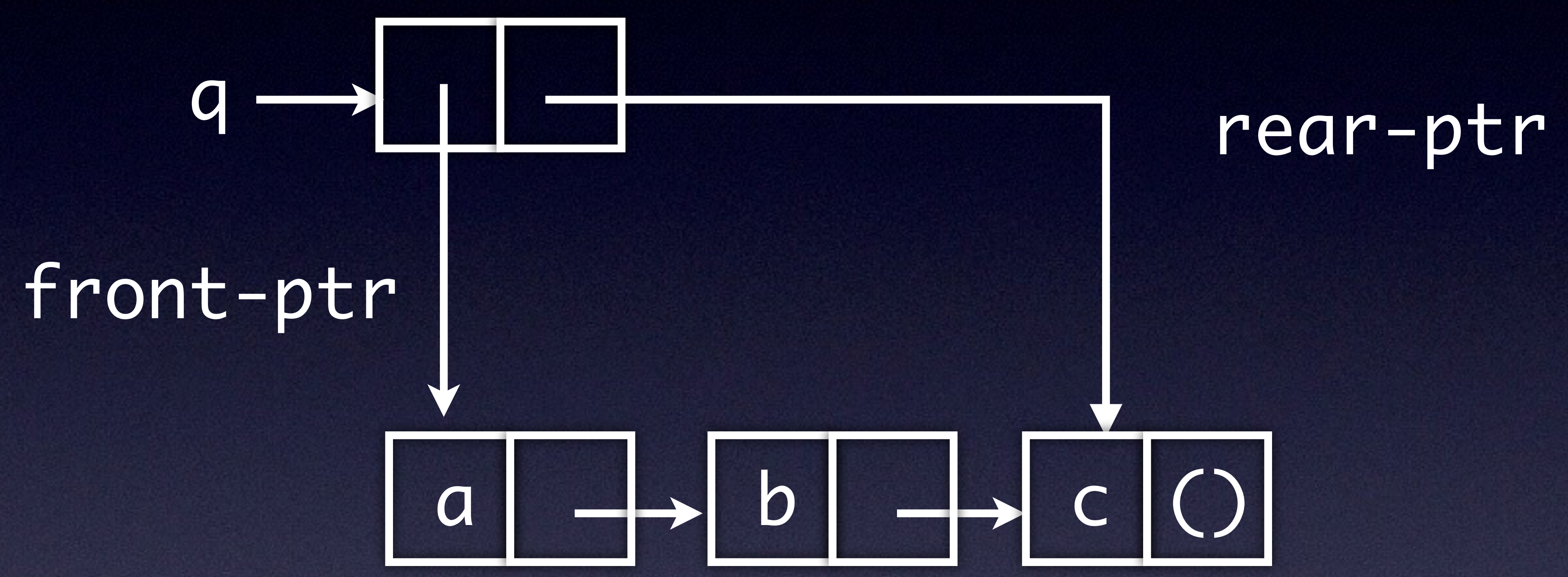
selectors

mutators

```
(make-queue)

(empty-queue? <queue>)

(front-queue <queue>)

(insert-queue! <queue> <item>)

(delete-queue! <queue> <queue>)
```

Representation & Implementation

```
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))
(define (front-ptr queue) (car queue))
(define (rear-ptr queue) (cdr queue))
```




```
(define (empty-queue? queue) (null? (front-ptr queue)))

(define (make-queue) (cons '() '()))

(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))
```

Implementation (ctd.)

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))

(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))
```

Chapter 3: Forms of Modularity

Organize a system
in a modular way

Raises the linguistic issue of "state"

According to the objects that live in the system

According to the streams of information that flow in the system

Raises the linguistic issue of "delayed evaluation"

Remember Lists as Standard Interfaces

Compute sum of all prime numbers in an interval

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

Standard iterative style (efficient)

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime? (enumerate-interval a b))))
```

Using list operations (elegant but painfully inefficient)

And a second list is created!

All integers are actually stored

Computations can be Outrageous

Find the second prime in the interval [10.000, 1.000.000]

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

Almost a million integers are stored. Most of them ignored

The inefficiency becomes painfully apparent

Streams to the Rescue

Streams are lazy lists. They are a clever idea that allows one to use sequence manipulations without incurring the costs of manipulating sequences as lists

```
(stream-car (cons-stream x y)) = x
(stream-cdr (cons-stream x y)) = y
```

∃

```
the-empty-stream
stream-null?
```

The difference is the time at which the elements are evaluated. With ordinary lists, both the car and the cdr are evaluated at construction time. With streams, the cdr is evaluated at selection time.

Very similar to lists

```scheme
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))
(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))
(define (stream-for-each proc s)
  (if (stream-null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for-each proc (stream-cdr s)))))
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter pred
                                     (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream)))))
```

Delayed Objects in Scheme

special form

```
(delay <exp>)          (force <exp>)
```

syntactic sugar

```
(cons-stream <a> <b>)   ⇔   (cons <a> (delay <b>))
(stream-car <exp>)      ⇔   (car <exp>)
(stream-cdr <exp>)      ⇔   (force (cdr <exp>))
```

Back to the Example

Find the second prime in the interval [10.000, 1.000.000]

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
       low
       (stream-enumerate-interval (+ low 1) high))))
```

only when needed

```
(stream-car
 (stream-cdr
  (stream-filter prime?
                 (stream-enumerate-interval 10000 1000000))))
```

Implementing Delay & Force

syntactic sugar

```
(delay <exp>)
```

⇔

```
(lambda () <exp>)
```

delay evaluation +
capture environment!

```
(force <exp>)
```

⇔

```
(<exp>)
```

eval in correct
environment!

Infinite Streams

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define integers (integers-starting-from 1))

(define (divisible? x y) (= (remainder x y) 0))

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
                 integers))

(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))
```

Defining Streams Implicitly

```scheme
(define ones (cons-stream 1 ones))

(define (add-streams s1 s2)
  (stream-map + s1 s2))

(define integers (cons-stream 1 (add-streams ones integers)))


(define fibs
  (cons-stream 0
               (cons-stream 1
                            (add-streams (stream-cdr fibs)
                                         fibs))))
```

Exploiting the Stream Paradigm

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))

(define (sqrt-stream x)
  (define guesses
    (cons-stream 1.0
                 (stream-map (lambda (guess)
                               (sqrt-improve guess x))
                             guesses)))
  guesses)
```

```
> (display-stream (sqrt-stream 2))
1.
1.5
1.4166666666666665
1.4142156862745097
1.414213562374689
...
```

Stream vs Object Paradigm

Revisiting the bank account example

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

vs

```
(define (stream-withdraw balance amount-stream)
  (cons-stream
   balance
   (stream-withdraw (- balance (stream-car amount-stream))
                    (stream-cdr amount-stream))))
```

Chapters 1 - 2 - 3

	data	procedures
primitive	X	X
combinations	X	X
abstraction	X	X

But this is not sufficient for organizing large systems. We studied modularity.

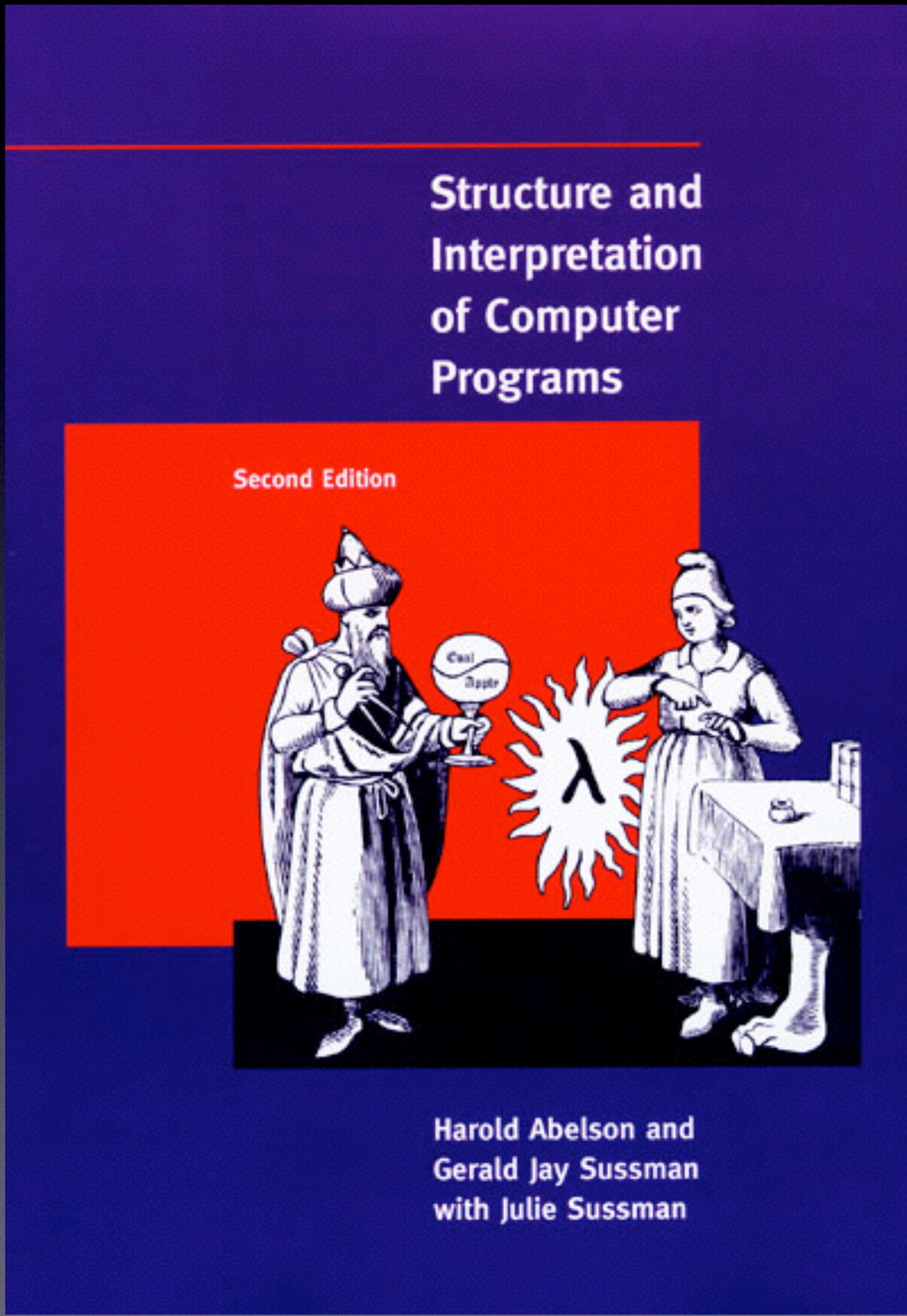
According to the objects that live in the system

According to the streams of information that flow in the system

Chapter 4

Metalinguistic Abstraction (Part I)

What if Scheme is not enough?



Metalinguistic abstraction -- establishing new languages -- plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators. An evaluator (or interpreter) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.

The interpreter is just another program