

# Chapter 3b

## Objects

# 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

or

According to the **streams** of  
information that flow in the system

Raises the linguistic issue of  
“delayed evaluation”

# Objects: Here's What we Want

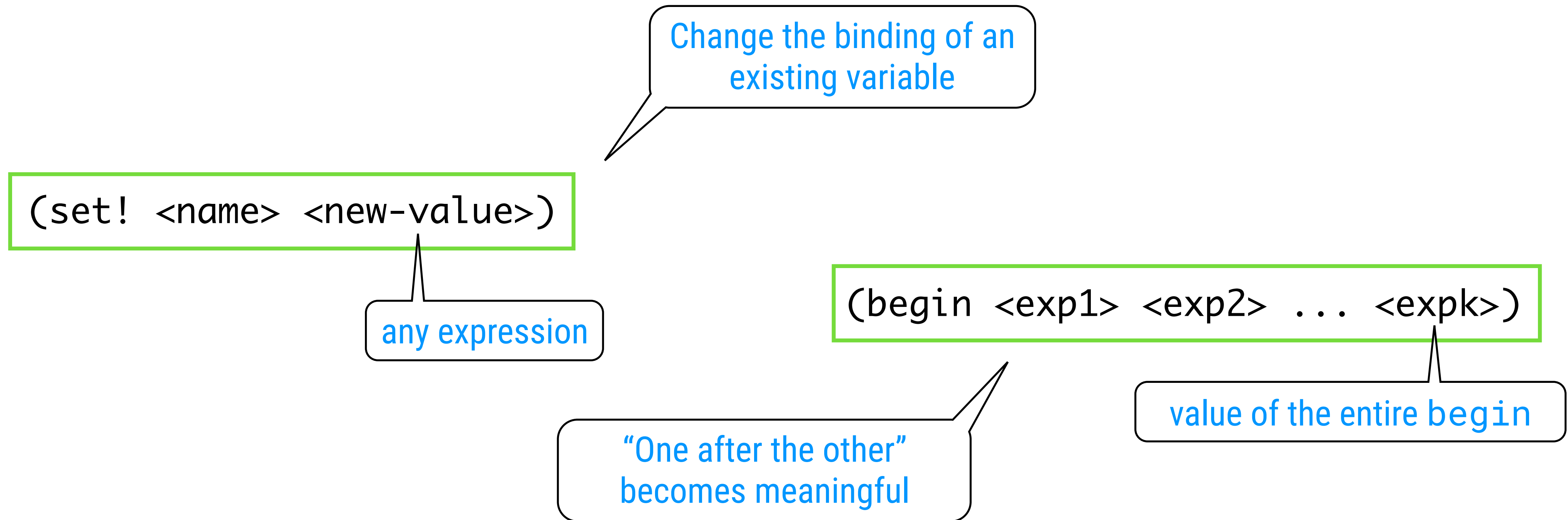
Not a mathematical  
function anymore!

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

It seems to  
"remember" stuff.

time-varying "local state"

# Two New Special Forms



From now on, we leave the realm of (pure) **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

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

local

but still only 1

```
> (new-withdraw 10)  
90  
> (set! balance 30000)
```

protection

⊕ set!: cannot set variable before its definition: balance

# Third Solution

parametrized

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

A class is a generator of objects

```
class account {
  int balance
  account(balance) {
    this.balance = balance };
  int withdraw(amount) {
    this.balance = balance - amount;
    return this.balance }
}
```

```
> (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)

```
(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)
```

This lambda “contains” the balance variable and 2 lambdas

make-account  
returns a lambda!

```
> (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))
```

messages



# 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 (- 25 20))
   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

Identifiers correspond to a place that can contain a value. Statements can change that value.

Scheme is NOT a FPL!

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.

# Subtleties 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

Even worse in concurrent programs

In imperative programming, we can no longer think of a function as a mathematical function:  $f(a) = f(a)$  is not always true.

Referential Transparency

Imperative variant

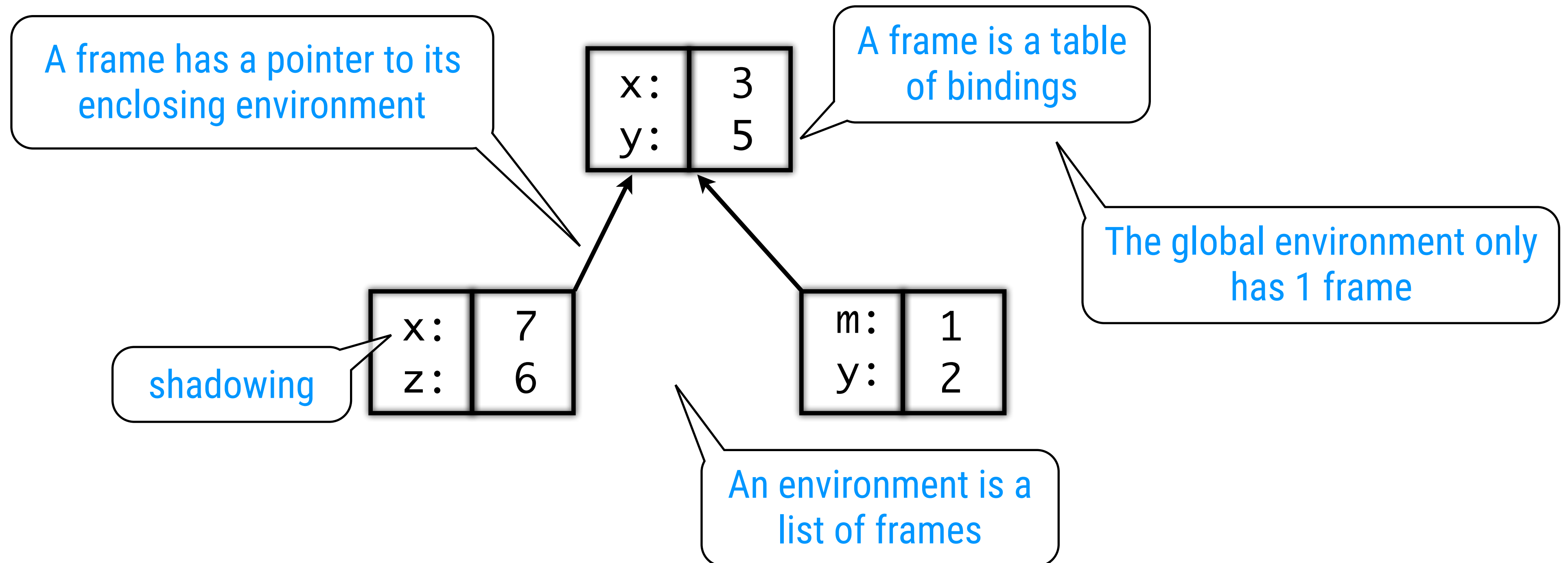
```
(define (factorial2 n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                  (set! counter (+ counter 1))
                  (iter))))
    (iter)))
```

The order becomes crucial: harder to reason about!

(iter)))  
||

# 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.

# Evaluation Rules: Version 2

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

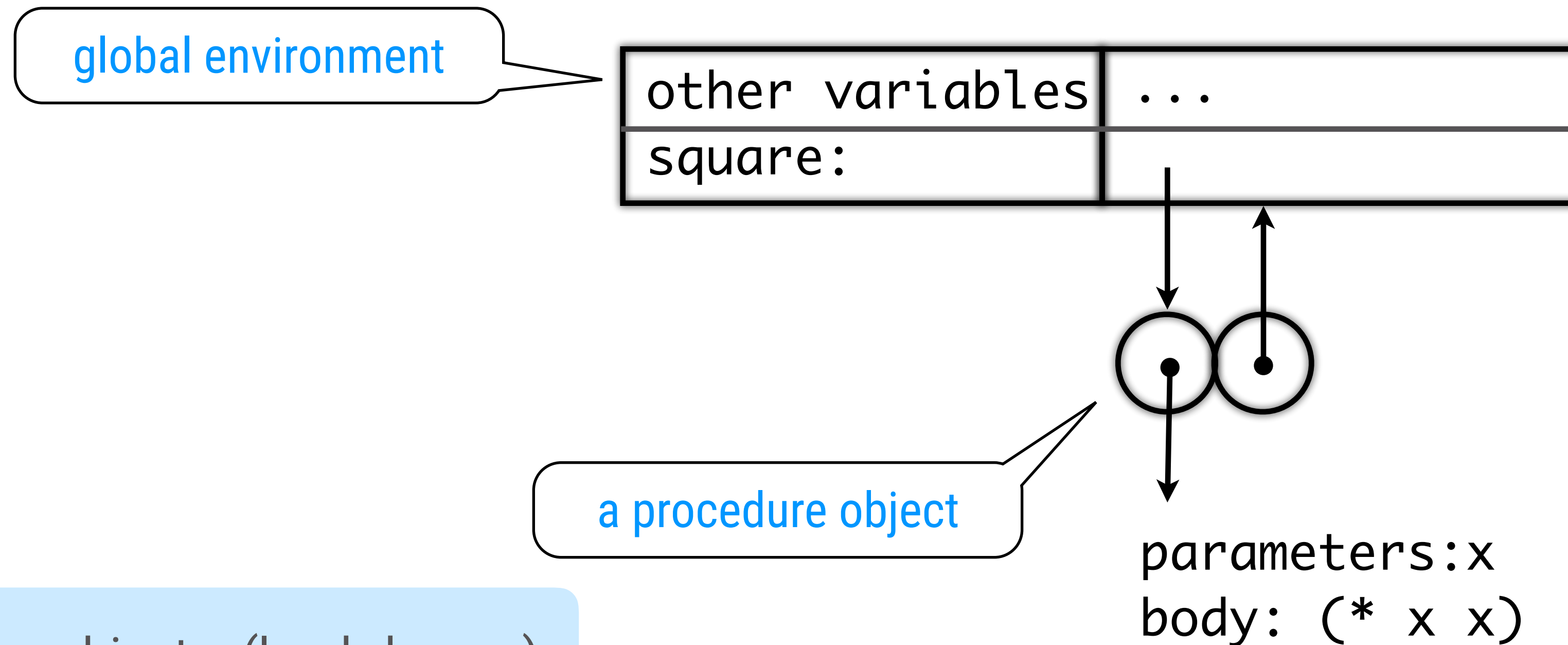
- **numerals** evaluate to numbers
  - **identifiers** evaluate to their value in the **environment**
  - **combinations**:
    - evaluate all the subexpressions in the combination in the environment
    - apply the procedure that is the value of the leftmost expression (= the operator) to the arguments 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**.

was: identifiers evaluate to the value of their binding

# Procedure Creation

**VERY IMPORTANT**

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



procedure object = (lambda, env)



# Procedure Application

**VERY IMPORTANT**

(square 5)

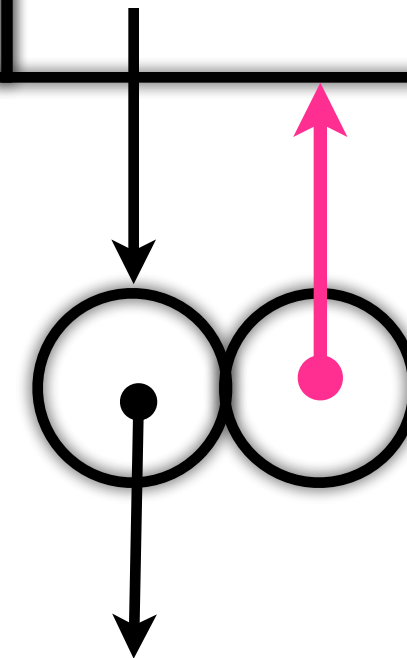
global environment

|                 |     |
|-----------------|-----|
| other variables | ... |
| square:         |     |

new environment

|    |   |
|----|---|
| x: | 5 |
|----|---|

(\* x x)



parameters: x  
body: (\* x x)

# Evaluation Rules: Version 2 (ctd)

- A **procedure** is **applied** to a set of arguments by **constructing an environment 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 frame** constructed. The new frame has as its **enclosing frame** the environment part of the procedure object that is applied.
- A **procedure** is **created** 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.

**VERY IMPORTANT**



# Example from Chapter 1: Creation

```
> (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)))
```

c.f. Substitution Model

global  
environment

sum-of-squares:  
square:  
f:

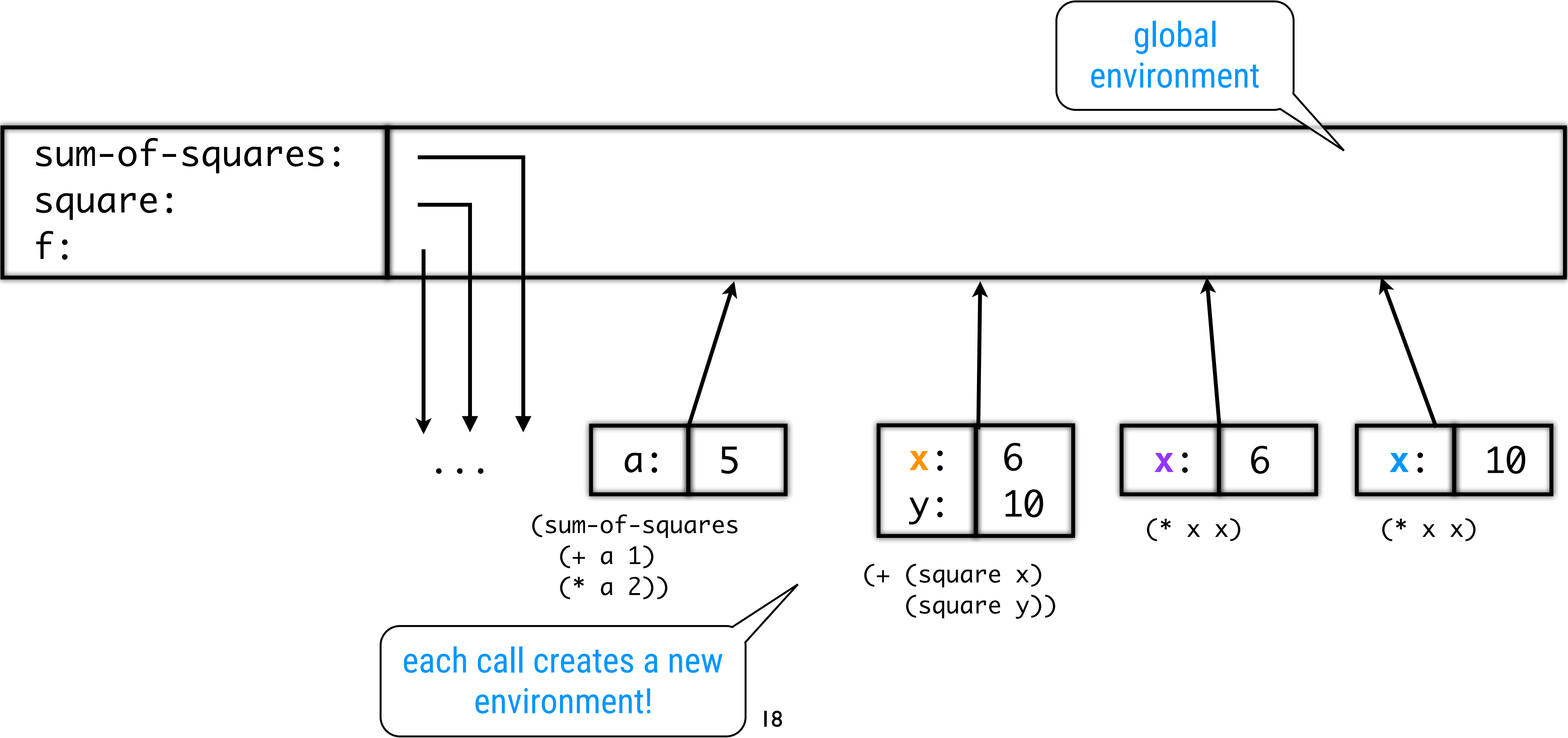
pars:a  
body: (sum-of-squares  
      (+ a 1)  
      (\* a 2))

pars:x  
body: (\* x x)

pars:x,y  
body: (+ (square x)  
         (square y))

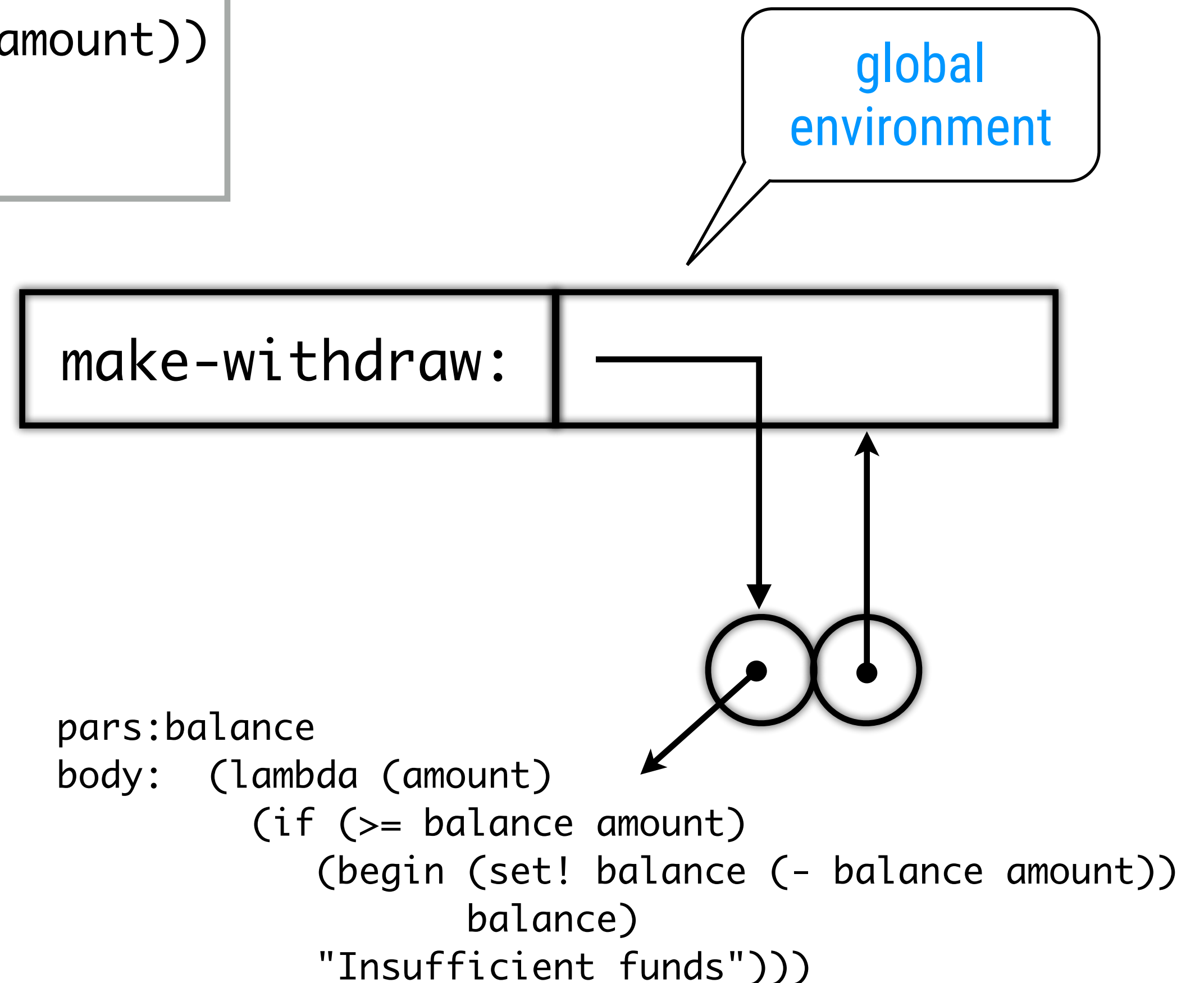
# Example from Chapter 1: Application

```
> (f 5)
```



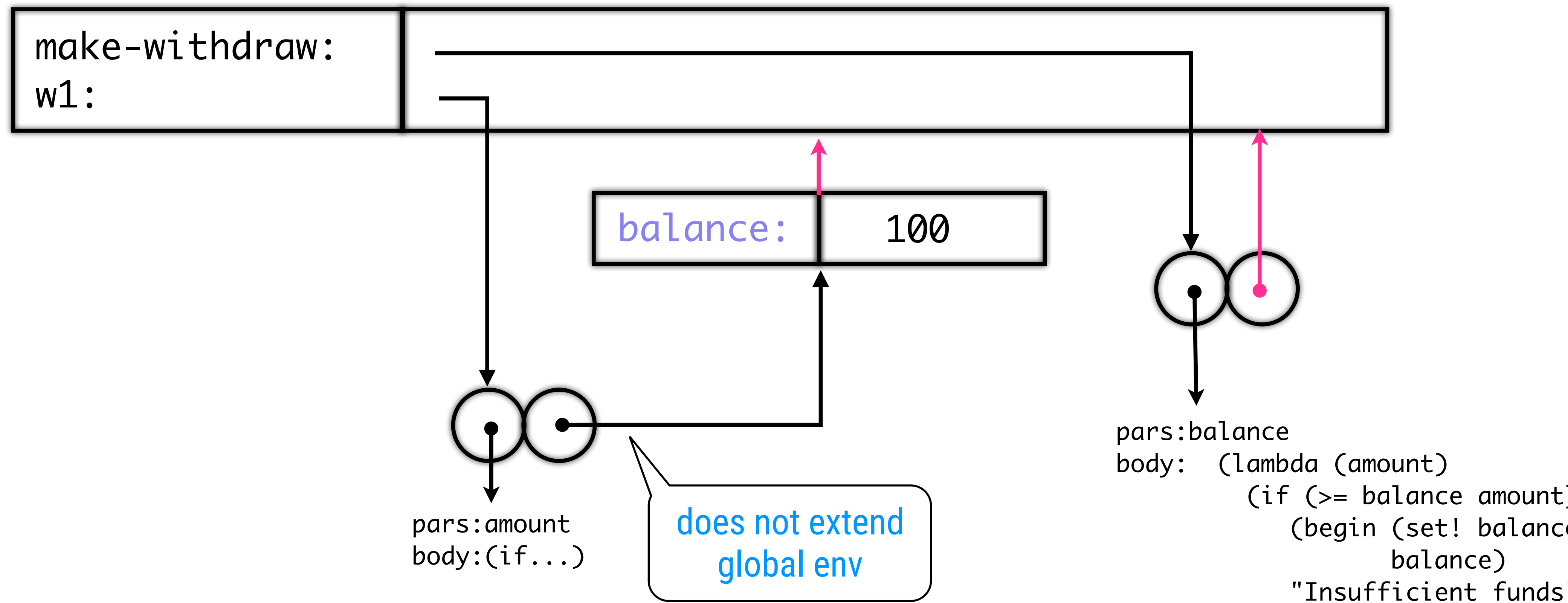
# Objects with Local State (1/4)

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



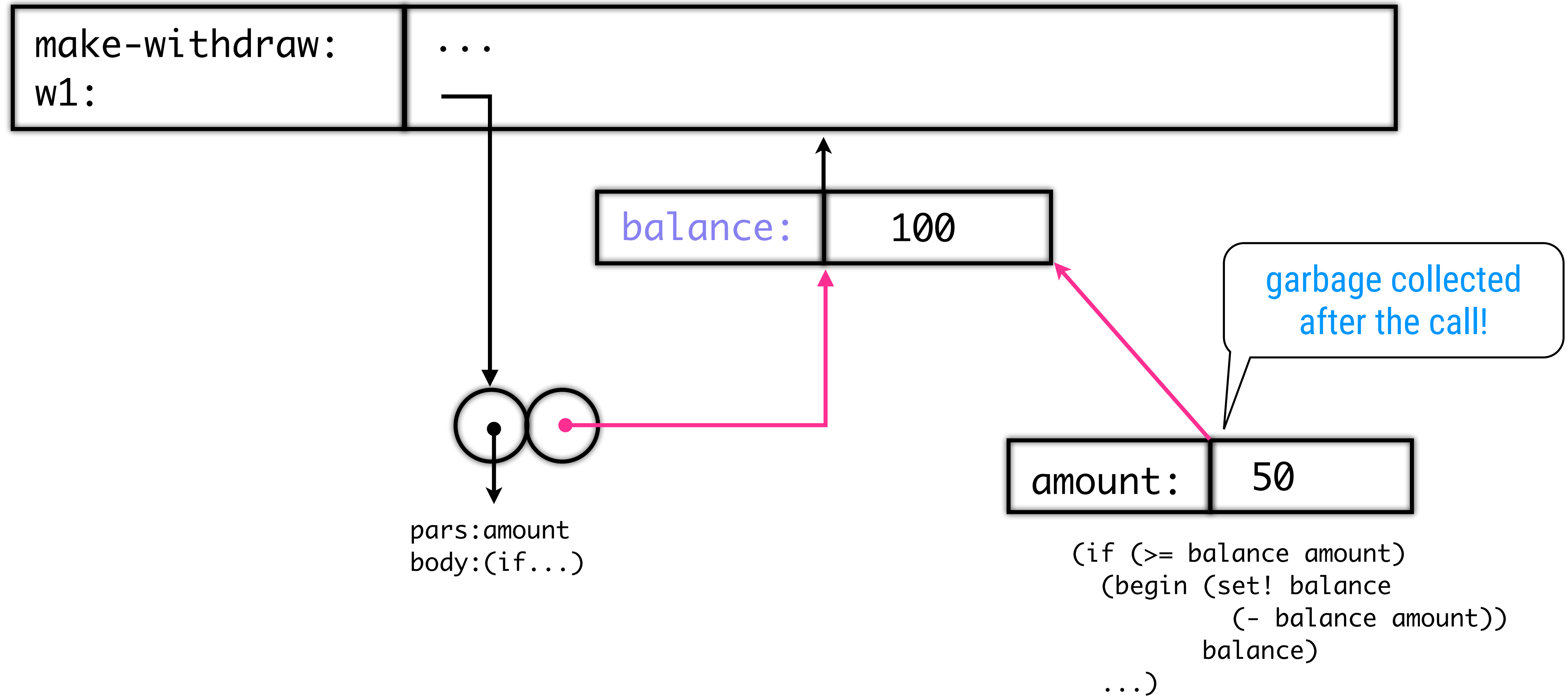
# Objects with Local State(2/4)

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



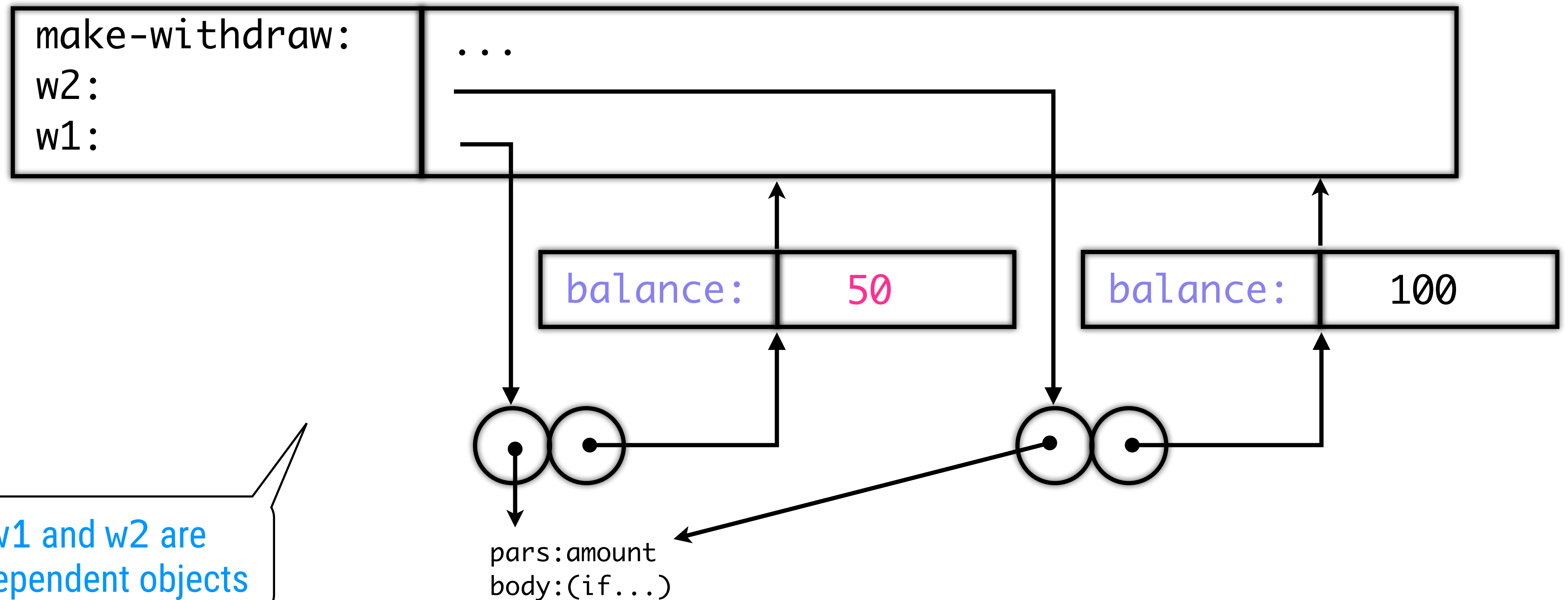
# Objects with Local State(3/4)

(w1 50)



# Objects with Local State(4/4)

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

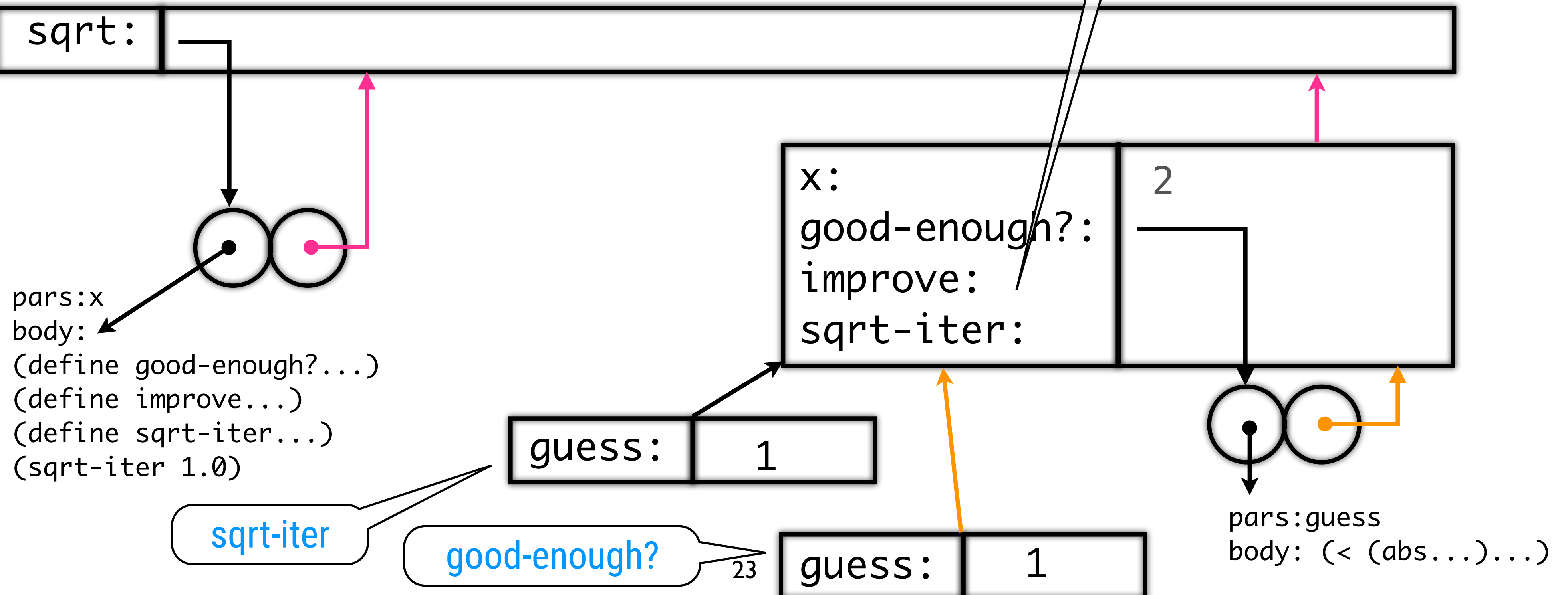


# Internal Definitions

```
(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))
```

(sqrt 2)

not shown for conciseness



# 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 linked to the evaluation environment for the enclosing procedure.



# Adding Another Dimension

|              | data | procedures |
|--------------|------|------------|
| primitive    | X    | X          |
| combinations | X    | X          |
| abstraction  | X    | X          |

so far: constructors, selectors  
now: mutators

Let's now investigate the interaction  
with mutable state.

# Add Two Primitives

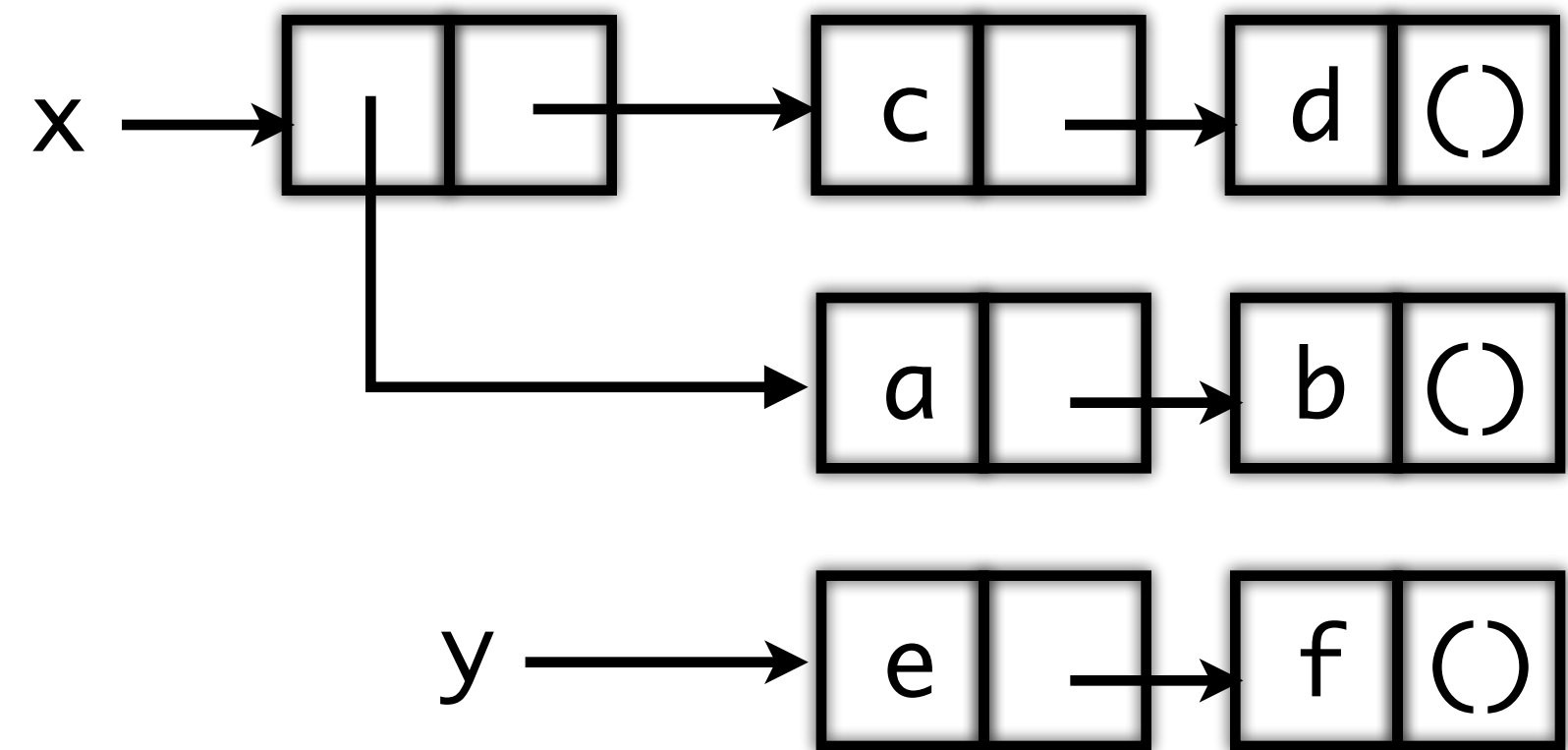
Modify existing pairs

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

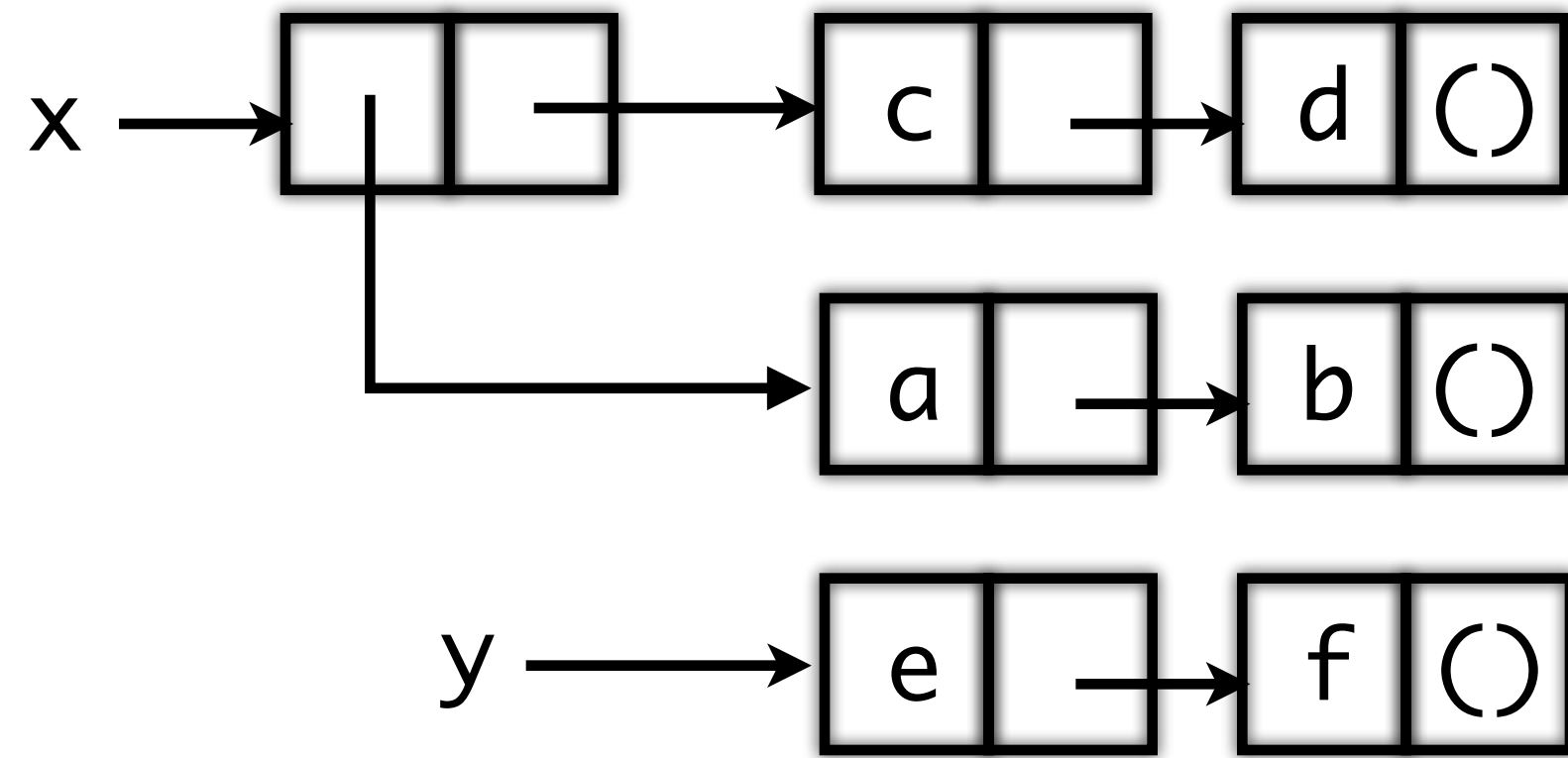
```
(set-cdr! <pair> <value>)
```

```
(define x '((a b) c d))
```

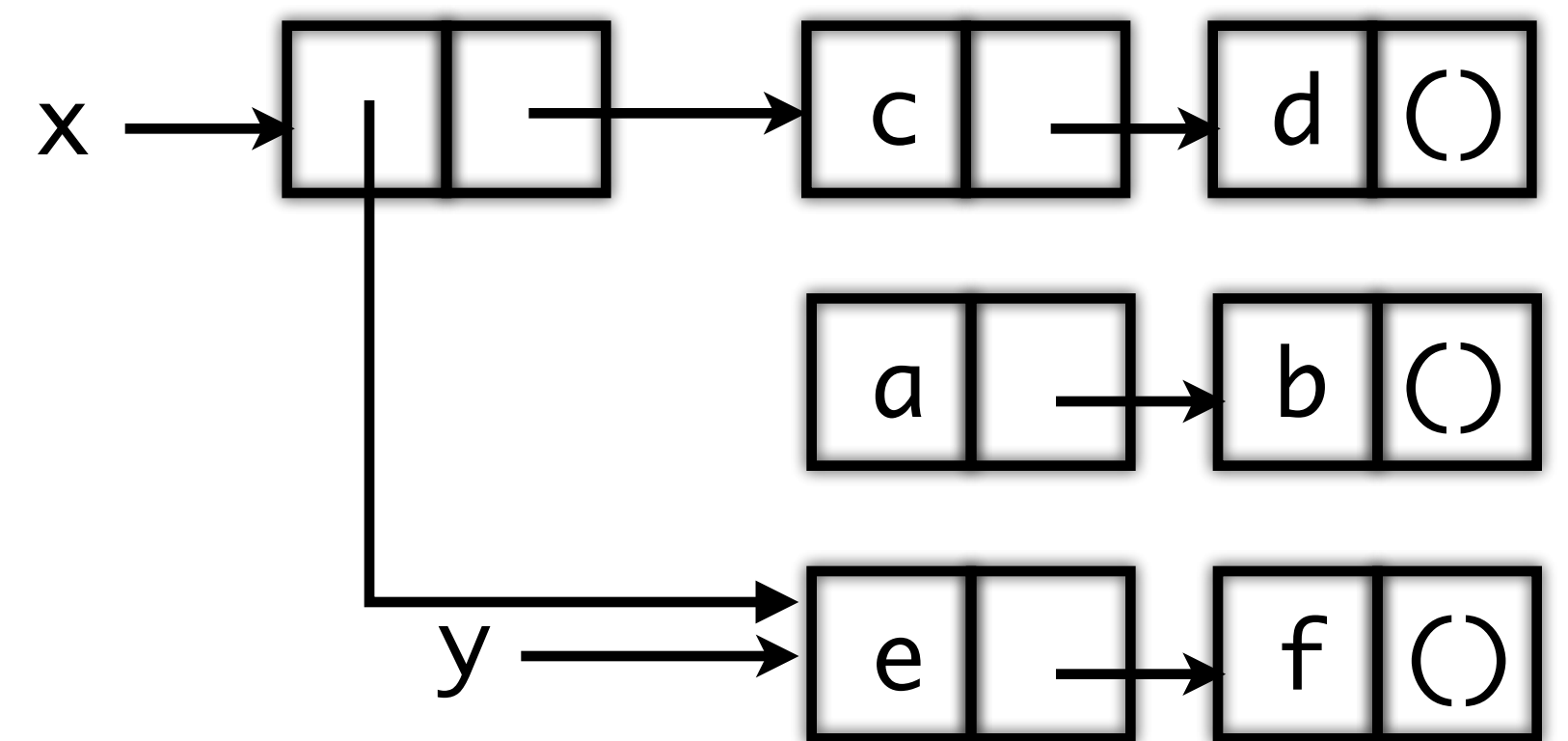
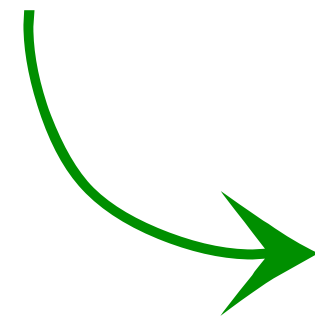
```
(define y '(e f))
```



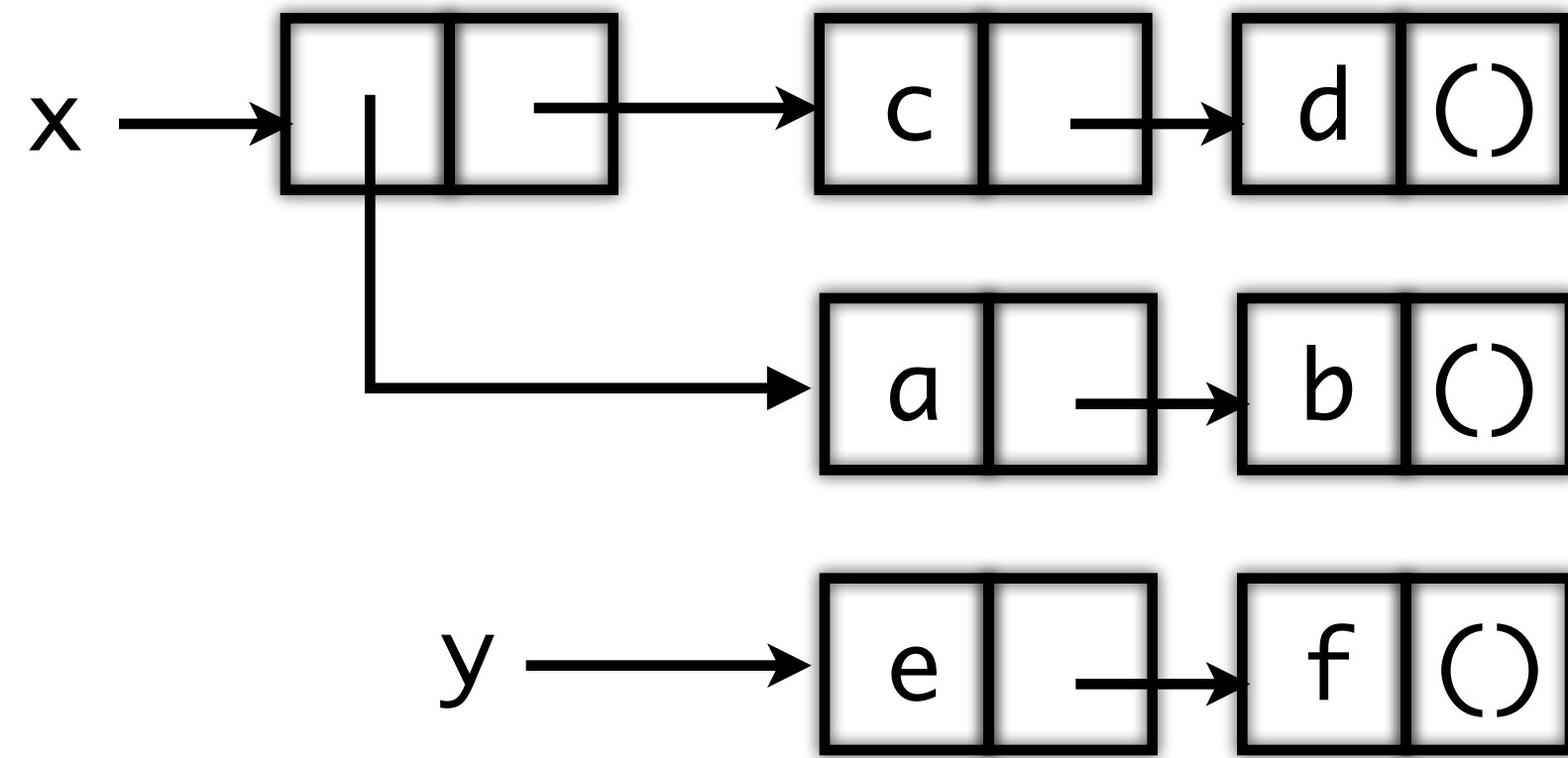
# Example 1



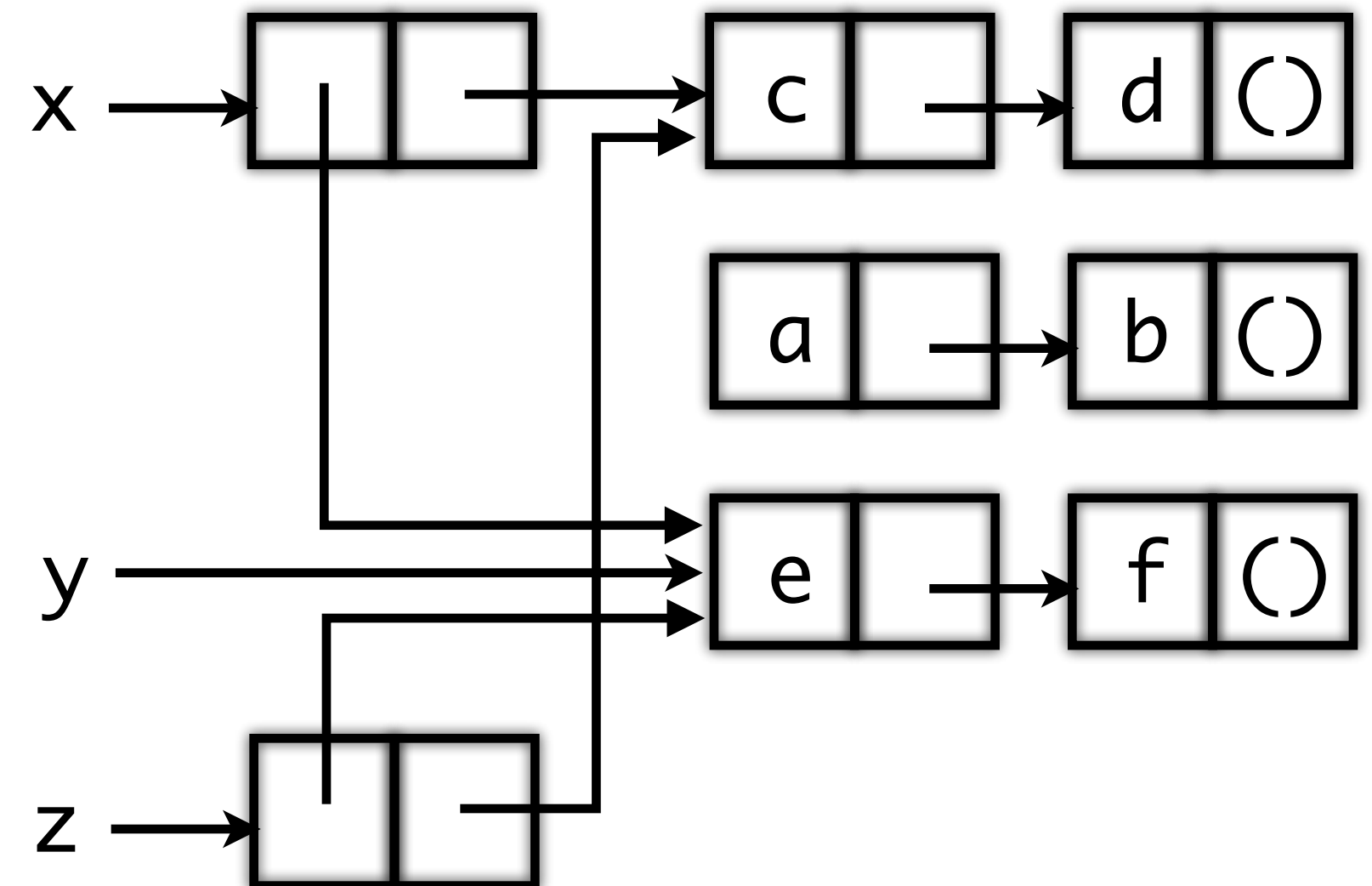
(set-car! x y)



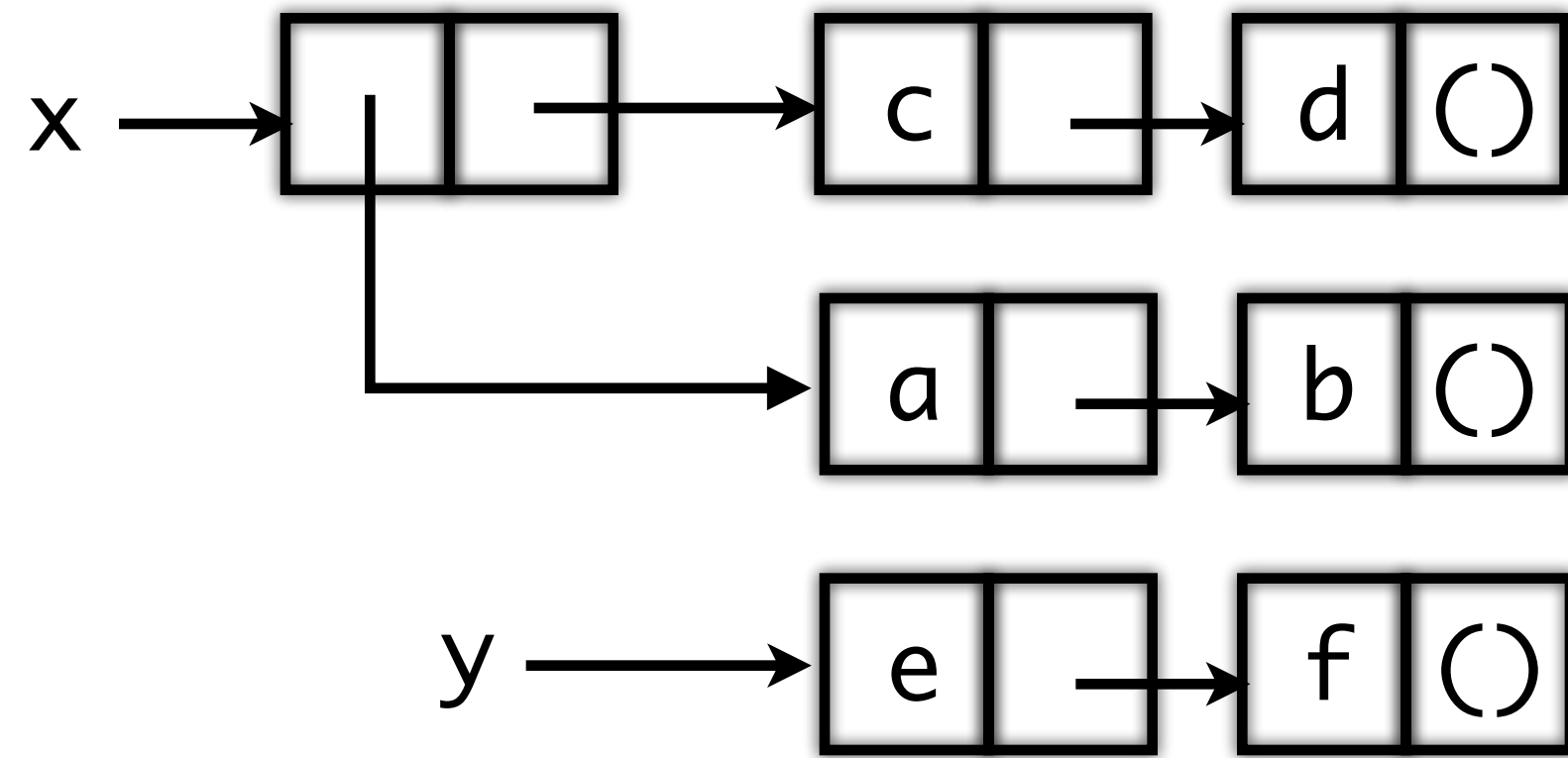
# Example 2



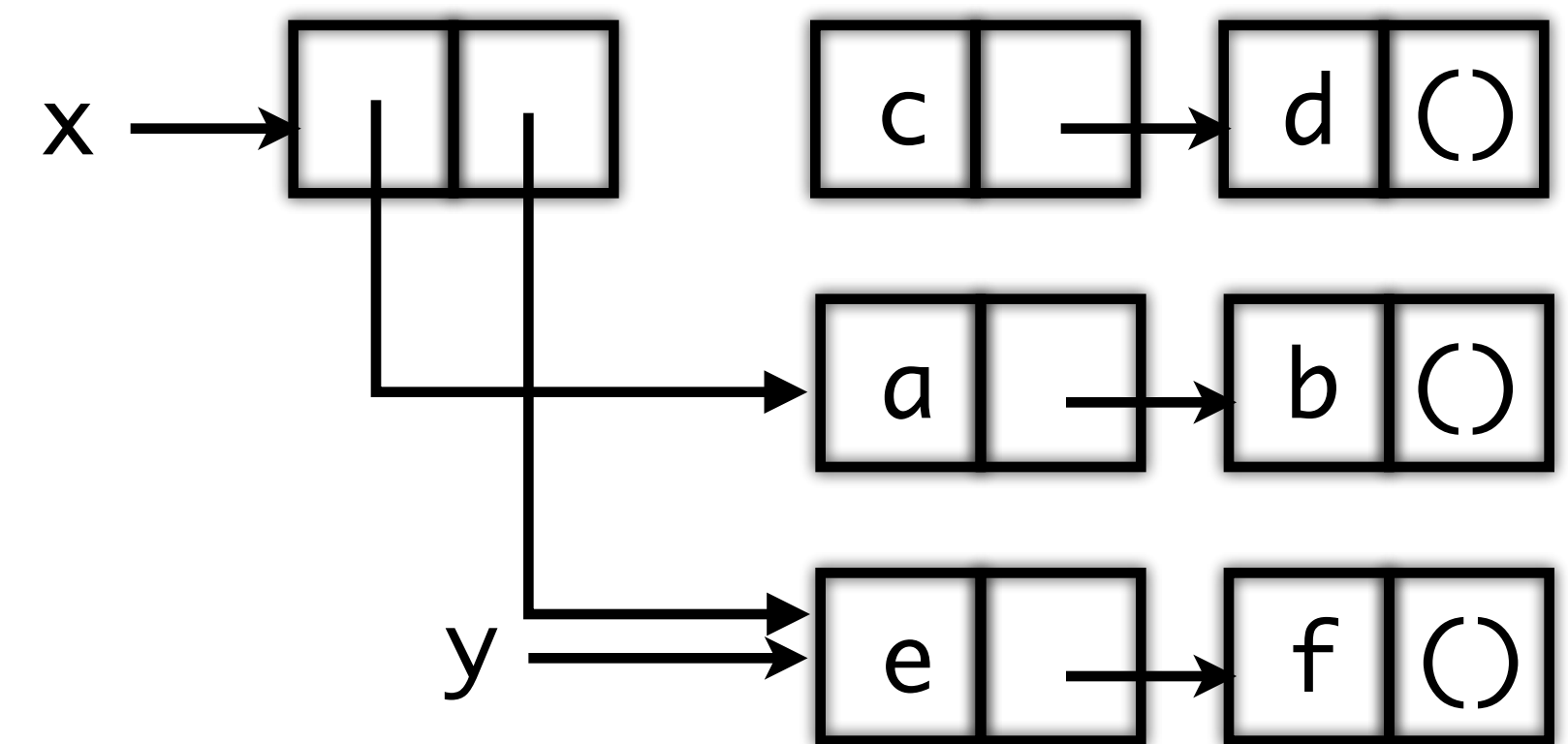
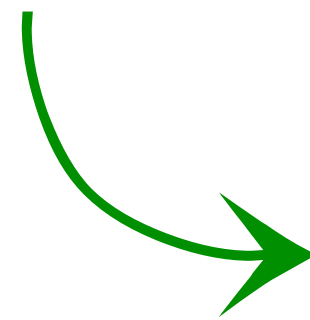
`(define z (cons y (cdr x)))`



# 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

selectors

(make-queue)

constructor

(empty-queue? <queue>)

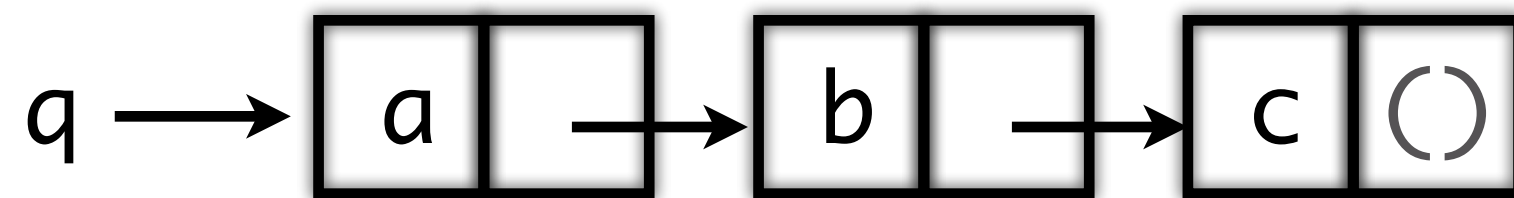
(front-queue <queue>)

mutators

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

(delete-queue! <queue>)

# Representation & Implementation I



represent as an ordinary list

```
(define q (make-queue))
```

```
(insert-queue! q 'a)
```

```
(insert-queue! q 'b)
```

```
(insert-queue! q 'c)
```

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

```
(define (make-queue) '())
```

...

```
(define (insert-queue! queue item)
  ; scan until end of list and append new element
)
```

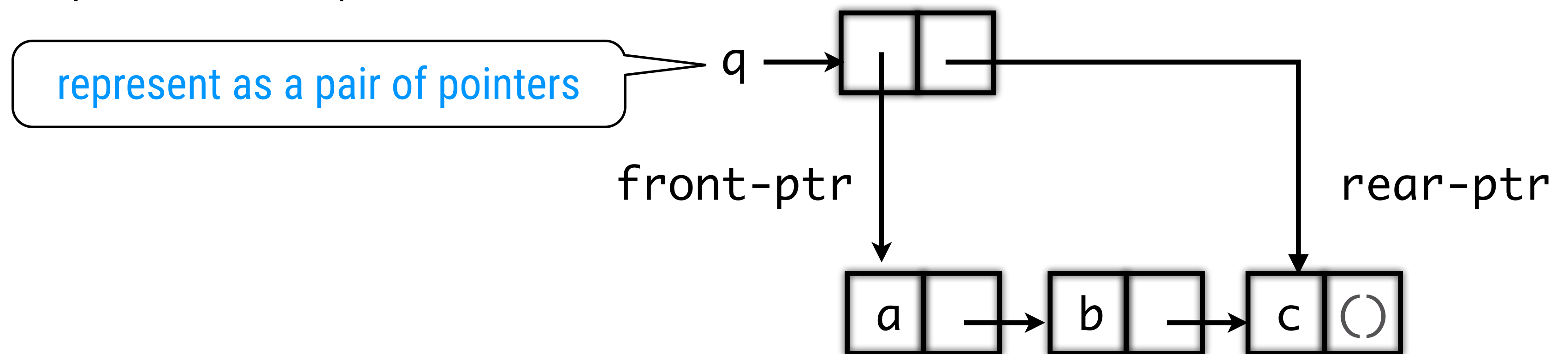
requires  $O(n)$  steps  
for list of  $n$  items

insertion inefficient!



# Representation & Implementation II

```
(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))
```



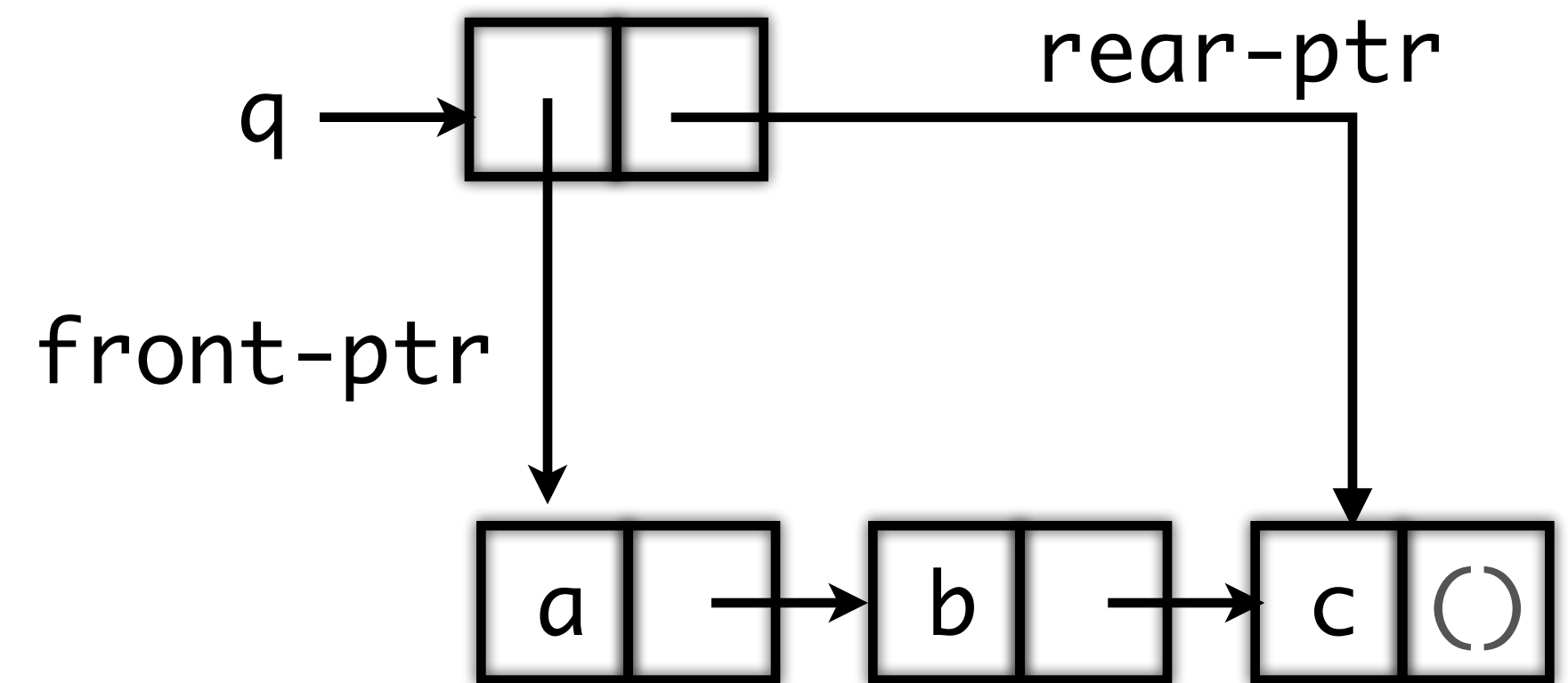
```
(define (empty-queue? queue) (null? (front-ptr queue)))
```

```
(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) 0(1)  
  (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))))
```

# Vectors in Scheme

“arrays”

```
(define (quicksort vector <<?)
  (define (swap i j)
    (let ((temp (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j temp)))
  ...
  (define (partition pivot i j)
    ...)
  (define (quicksort-main l r)
    (if (< l r)
        (begin
          (if (<<? (vector-ref vector r)
                    (vector-ref vector l))
              (swap l r))
          (let ((m (partition (vector-ref vector l) (+ l 1) r)))
            (swap l m)
            (quicksort-main l (- m 1))
            (quicksort-main (+ m 1) r))))))
  (quicksort-main 0 (- (vector-length vector) 1)))
```

```
(make-vector dim [val])
(vector-ref v idx)
(vector-set! v idx val)
```

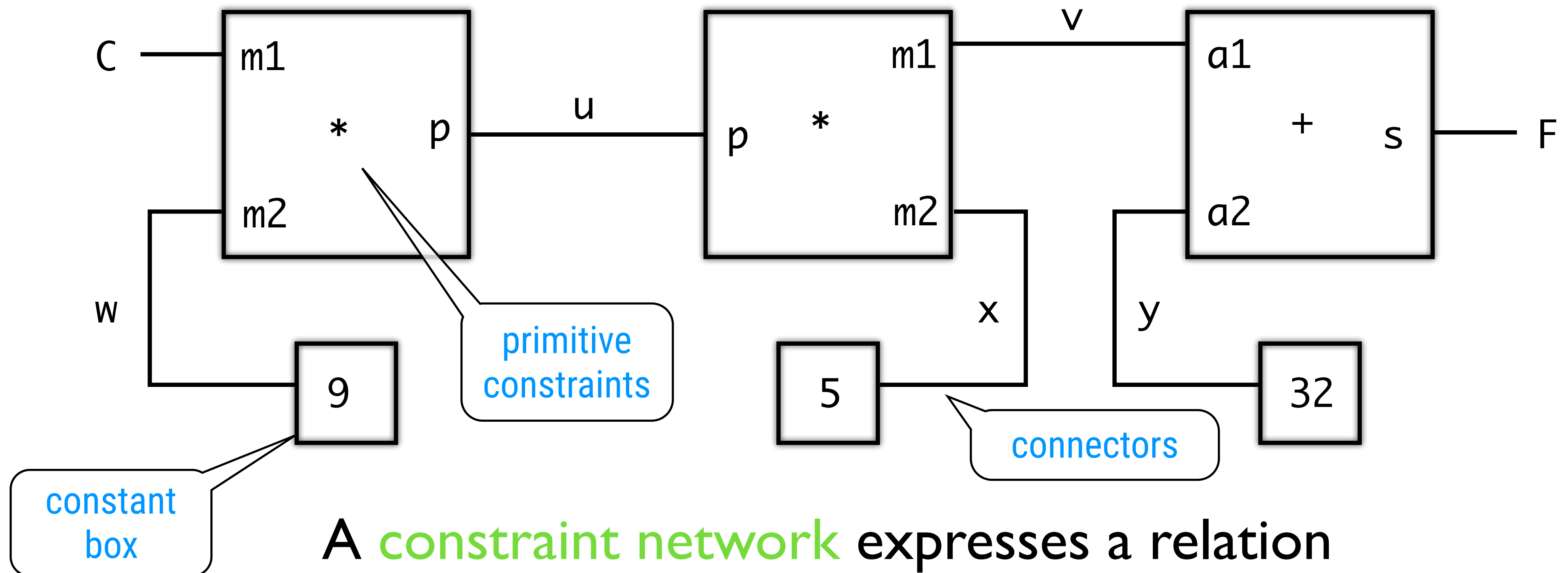
# Vectors (ctd.)

```
(define (quicksort vector <<?)
  ...
  (define (shift-to-right i x)
    (if (<<? (vector-ref vector i) x)
        (shift-to-right (+ i 1) x)
        i))
  (define (shift-to-left j x)
    (if (<<? x (vector-ref vector j))
        (shift-to-left (- j 1) x)
        j))
  (define (partition pivot i j)
    (let ((shifted-i (shift-to-right i pivot))
          (shifted-j (shift-to-left j pivot)))
      (cond ((< shifted-i shifted-j)
              (swap shifted-i shifted-j)
              (partition pivot shifted-i (- shifted-j 1)))
            ((>= shifted-i shifted-j)
              shifted-j))))
  (define (quicksort-main l r)
    ...)
  (quicksort-main 0 (- (vector-length vector) 1)))
```

# Case Study: Constraint Programming

$$9C = 5(F-32)$$

bidirectional  
relationship



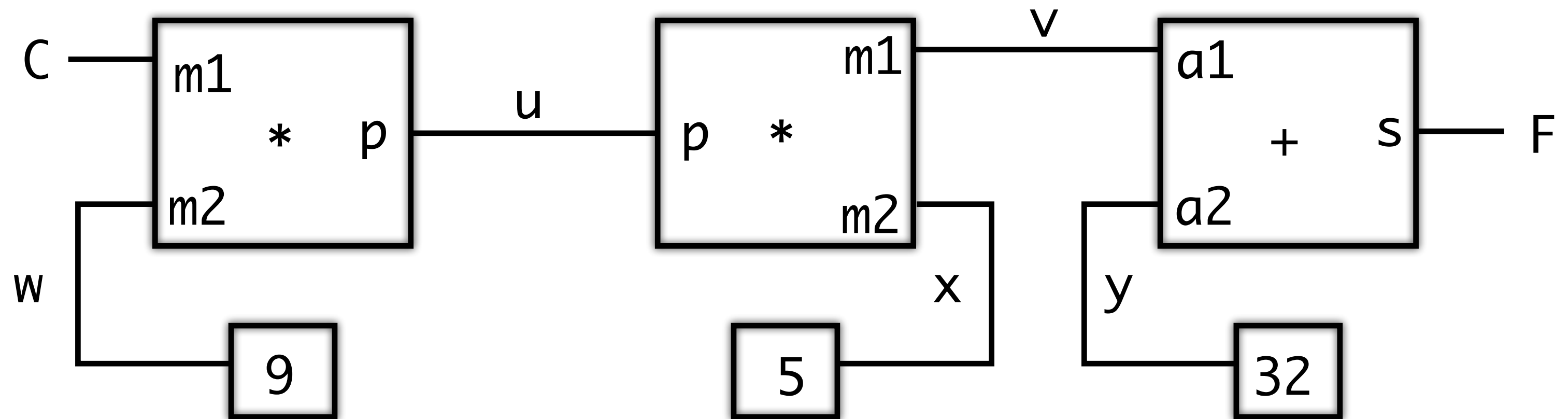
# Case Study: Constraint Programming

```
(define (celsius-fahrenheit-converter c f)
  (let ((u (make-connector))
        (v (make-connector))
        (w (make-connector))
        (x (make-connector))
        (y (make-connector)))
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)
    'ok))
```

internal  
connectors

primitive  
constraints

```
(define C (make-connector))
(define F (make-connector))
(celsius-fahrenheit-converter C F)
```



# Architecture

**Connectors** have a value. A connector is implemented as an **object with state**: its value, its informant (i.e. who set the value) and its connected constraints. **When setting the value**, all connected constraints (except for the informant) are given a tick so that they can recalculate themselves.

**Constraints** have a type (adder, multiplier,...) and a number of connectors. They are also implemented as an **object with state**. **When given a tick**, they query two connectors having a value and recalculate the value of the “third” connector.



# Connectors

turn on  
"tracing"

user sets C's  
value

F's value after  
propagation

ADT

(**has-value?** <con>)

(**get-value** <con>)

(**set-value!** <con> <val> <informant>)

(**forget-value!** <con> <val> <retractor>)

(**connect** <con> <new-constraint>)

```
> (probe "Celsius temp" C)
> (probe "Fahrenheit temp" F)
> (set-value! C 25 'user)
Probe: Celsius temp = 25
Probe: Fahrenheit temp = 77
done
> (get-value F)
77
> (set-value! F 212 'user)
Error! Contradiction (77 212)
> (forget-value! C 'user)
Probe: Celsius temp = ?
Probe: Fahrenheit temp = ?
done
> (set-value! F 212 'user)
Probe: Fahrenheit temp = 212
Probe: Celsius temp = 100
done
```



# Procedural layer atop OOP layer

```
(define (has-value? connector)  
  (connector 'has-value?))
```

```
(define (get-value connector)  
  (connector 'value))
```

```
(define (set-value! connector new-value informant)  
  ((connector 'set-value!) new-value informant))
```

```
(define (forget-value! connector retractor)  
  ((connector 'forget) retractor))
```

```
(define (connect connector new-constraint)  
  ((connector 'connect) new-constraint))
```

# Giving all Constraints a “tick”

```
(define (inform-about-value constraint)
  (constraint 'I-have-a-value))
(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))
```

ticks given by connector

constraints

workhorse that gives it

```
(define (for-each-except exception procedure list)
  (define (loop items)
    (cond ((null? items) 'done)
          ((eq? (car items) exception) (loop (cdr items)))
          (else (procedure (car items))
                  (loop (cdr items)))))
  (loop list))
```

# An Adder Constraint

```
(define (adder a1 a2 sum)
  (define (process-new-value)
    (cond ((and (has-value? a1) (has-value? a2))
      (set-value! sum
        (+ (get-value a1) (get-value a2))
        me))
      ((and (has-value? a1) (has-value? sum))
        (set-value! a2
          (- (get-value sum) (get-value a1))
          me))
      ((and (has-value? a2) (has-value? sum))
        (set-value! a1
          (- (get-value sum) (get-value a2))
          me))))))

(define (process-forget-value)
  (forget-value! sum me)
  (forget-value! a1 me)
  (forget-value! a2 me)
  (process-new-value))

(define (me request)
  (cond ((eq? request 'I-have-a-value)
    (process-new-value))
    ((eq? request 'I-lost-my-value)
    (process-forget-value))
    (else
    (error "Unknown request -- ADDER" request))))

(connect a1 me)
(connect a2 me)
(connect sum me)
me)
```

recalculate "the third one"

clear all connectors

```
(define (has-value? connector)
  (connector 'has-value?))

...
```

```
(define (connect connector new-constraint)
  ((connector 'connect) new-constraint))
```

# An Multiplier Constraint

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1) (= (get-value m1) 0))
              (and (has-value? m2) (= (get-value m2) 0)))
          (set-value! product 0 me))
          ((and (has-value? m1) (has-value? m2))
           (set-value! product
                        (* (get-value m1) (get-value m2))
                        me))
          ((and (has-value? product) (has-value? m1))
           (set-value! m2
                        (/ (get-value product) (get-value m1))
                        me))
          ((and (has-value? product) (has-value? m2))
           (set-value! m1
                        (/ (get-value product) (get-value m2))
                        me))))
  (define (process-forget-value)
    (forget-value! product me)
    (forget-value! m1 me)
    (forget-value! m2 me)
    (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
          (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else
           (error "Unknown request -- MULTIPLIER" request))))
  (connect m1 me)
  (connect m2 me)
  (connect product me)
  me)
```

recalculate "the  
third one"

# Primitive Constraints

```
(define (constant value connector)
  (define (me request)
    (error "Unknown request -- CONSTANT" request))
  (connect connector me)
  (set-value! connector value me)
  me)
```

cannot change a  
constant

```
(define (probe name connector)
  (define (print-probe value)
    (newline)
    (display "Probe: ")
    (display name)
    (display " = ")
    (display value))
  (define (process-new-value)
    (print-probe (get-value connector)))
  (define (process-forget-value)
    (print-probe "?"))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else
           (error "Unknown request -- PROBE" request))))
  (connect connector me)
  me)
```

just printout the values of  
the connected connector

# Finally

```
(define (make-connector)
  (let ((value #f) (informant #f) (constraints '()))
    (define (set-my-value newval setter)
      ...)
    (define (forget-my-value retractor)
      ...)
    (define (connect new-constraint)
      ...)
    (define (me request)
      (cond ((eq? request 'has-value?)
             (if informant #t #f))
            ((eq? request 'value) value)
            ((eq? request 'set-value!) set-my-value)
            ((eq? request 'forget) forget-my-value)
            ((eq? request 'connect) connect)
            (else (error "Unknown operation -- CONNECTOR"
                          request))))
    me)))
```

```
(define (inform-about-value constraint)
  (constraint 'I-have-a-value))
(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))
```

```
(define (set-my-value newval setter)
  (cond ((not (has-value? me))
        (set! value newval)
        (set! informant setter)
        (for-each-except setter
                          inform-about-value
                          constraints)))
        ((not (= value newval))
         (error "Contradiction" (list value newval)))
        (else 'ignored)))

(define (forget-my-value retractor)
  (if (eq? retractor informant)
      (begin (set! informant #f)
              (for-each-except retractor
                              inform-about-no-value
                              constraints))
      'ignored))

(define (connect new-constraint)
  (if (not (memq new-constraint constraints))
      (set! constraints
              (cons new-constraint constraints)))
  (if (has-value? me)
      (inform-about-value new-constraint))
  'done)
```



# 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 3

