Topic 17
Assignment, Local State, and the Environment Model of Evaluation

Section 3.1 & 3.2

Substitution Model and Why it Doesn't work

- You have just been introduced to an *assignment operator* in scheme – this is really the first time where symbols are viewed as variables whose values can be set (rather than as values themselves)
- Introducing assignment breaks us away from the functional model of programming

Today

1. We will explicitly see WHY the *substitution model* of evaluation no longer works and
2. We will be introduced to the *environment model* of evaluation that can better explain the behavior of these new programs

Functional Programming

- (Almost) everything we've seen to now has been "functional"
- **Functional** in the sense that it is based on a mathematical model of functions
- Each of our procedures take input and return a value

Functional

- a function, always returns the **same** value for the same inputs:
  - f(x) = 2x+3
  - f(3) = 9 … always
  - f(3) = 9
- I hope this seems obvious…

Functional

(fib 6)
→ 8

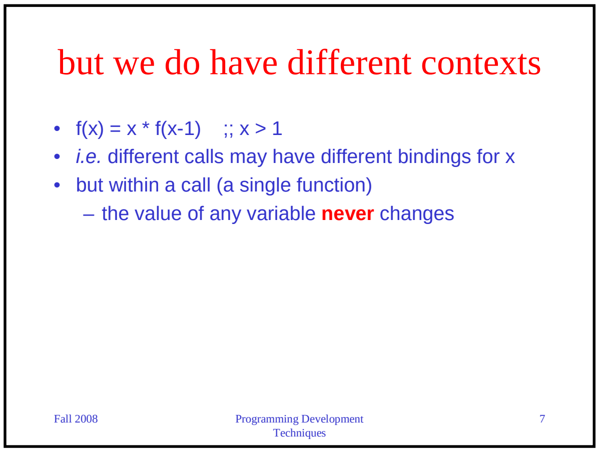
(fact 6)
→ 720

(fib 6)
→ 8

(fact 6)
→ 720

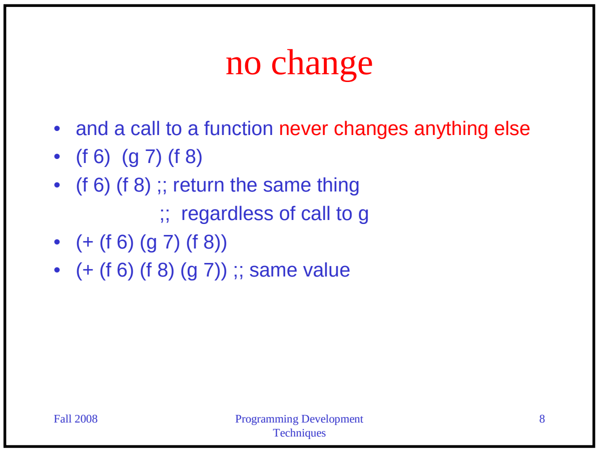
values never change

- once we assign a value
  - it is always the same
  - it **never** changes
- x = 6
  - then x always equals 6 in this context



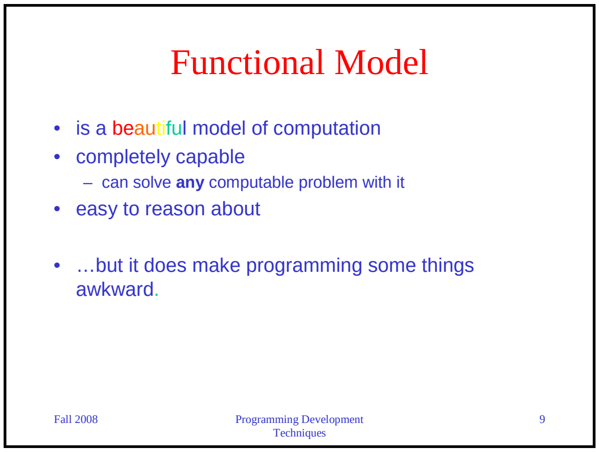
but we do have different contexts

- f(x) = x * f(x-1)   ;; x > 1
- *i.e.* different calls may have different bindings for x
- but within a call (a single function)
  – the value of any variable **never** changes



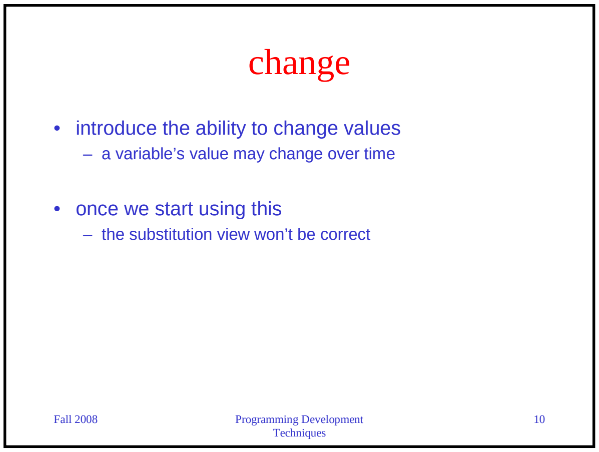
no change

- and a call to a function never changes anything else
- (f 6) (g 7) (f 8)
- (f 6) (f 8) ;; return the same thing
  ;; regardless of call to g
- (+ (f 6) (g 7) (f 8))
- (+ (f 6) (f 8) (g 7)) ;; same value



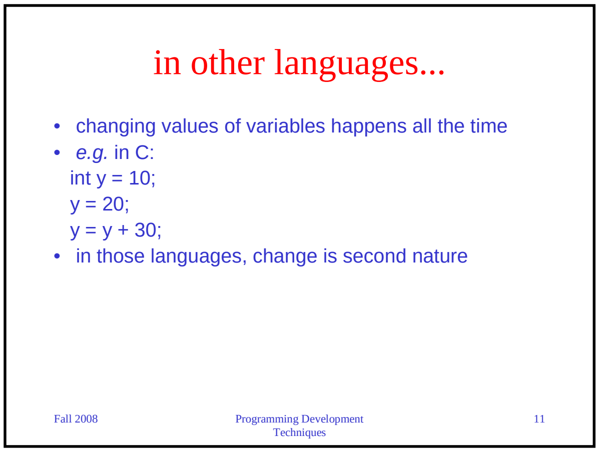
Functional Model

- is a beautiful model of computation
- completely capable
  – can solve **any** computable problem with it
- easy to reason about
- …but it does make programming some things awkward.



change

- introduce the ability to change values
  – a variable’s value may change over time
- once we start using this
  – the substitution view won’t be correct

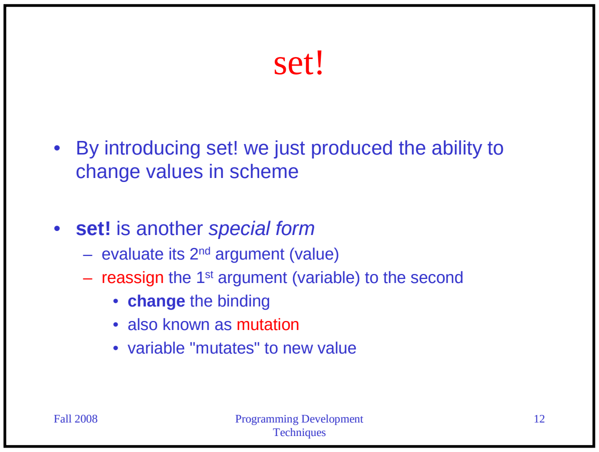


in other languages...

- changing values of variables happens all the time
- *e.g.* in C:

```
int y = 10;
y = 20;
y = y + 30;
```

- in those languages, change is second nature



set!

- By introducing set! we just produced the ability to change values in scheme
- **set!** is another *special form*
  – evaluate its 2nd argument (value)
  – reassign the 1st argument (variable) to the second
    - **change** the binding
    - also known as mutation
    - variable "mutates" to new value

consider:

- (**define** astate 0)
- (**define** (accum0! x)
  (**set!** astate
  (**+** astate x)))

- (accum0! 1)
- astate
- → 1
- (accum0! 1)
- astate
- → 2

value changes over time

- (define (accum0! x)
  (set! astate
  (+ astate x)))
- astate does not have a unique value here
  - initially has one value
  - has a different value after assignment

accumulator (revised)

- (**define** astate 0)
- (**define** (accum! x)
  (**begin**
  (**set!** astate (+ astate x))
  astate))
- Now, the set! expression changes the value of the final expression

accumulator (revised)

- (**define** astate 0)
- (**define** (accum! x)
  (**begin**
  (**set!** astate (+ astate x))
  astate))
  - that is:
    - (begin (set! astate (+ astate x)) astate)
    - is not the same as merely: astate

using accum!

- (accum! 1)

→ 1

- (accum! 1)

→ 2

- (accum! 1)

→ 3

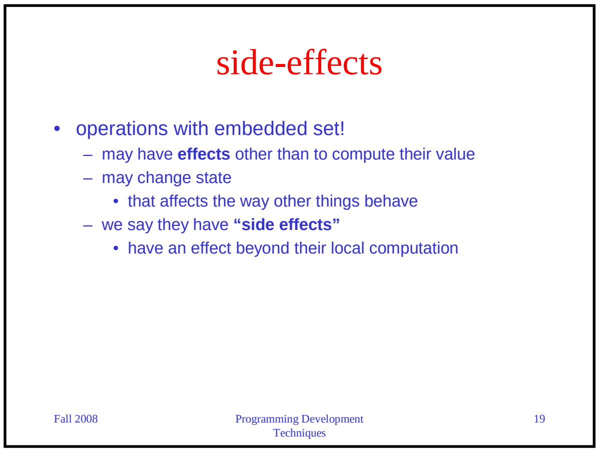
history starts to matter

- (define astate 0)
- (begin
  (accum! 1)
  (accum! 1)
  (accum! 1)
)
- → 3

not same as:

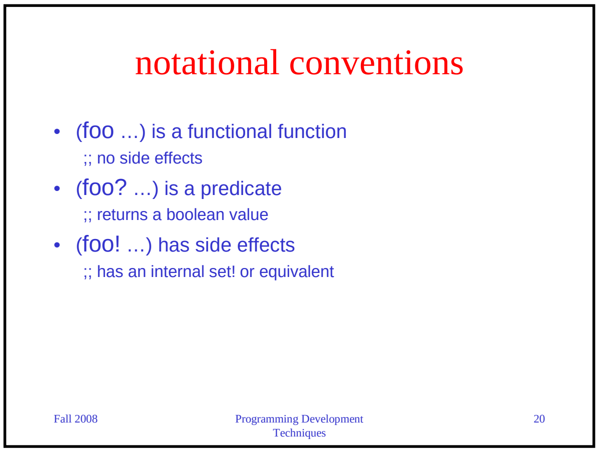
- (define astate 0)
- (accum! 1)
- → 1

- intervening accum!'s change the value of astate
- changes the value of the final (accum! 1)



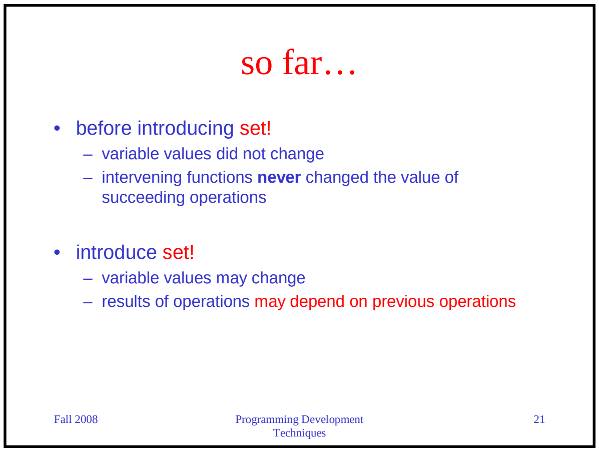
side-effects

- operations with embedded set!
  - may have **effects** other than to compute their value
  - may change state
    - that affects the way other things behave
  - we say they have **“side effects”**
    - have an effect beyond their local computation



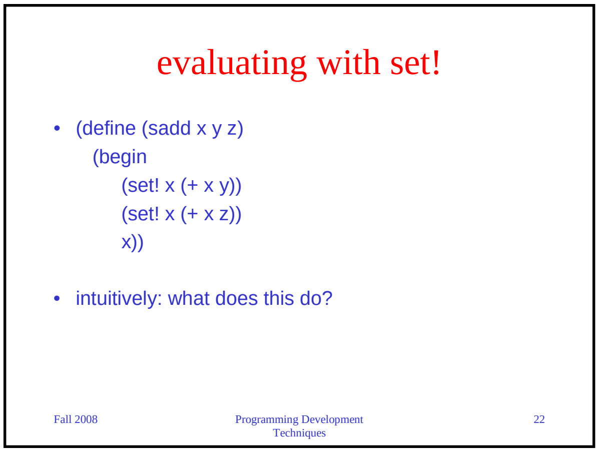
notational conventions

- (foo …) is a functional function
  ;; no side effects
- (foo? …) is a predicate
  ;; returns a boolean value
- (foo! …) has side effects
  ;; has an internal set! or equivalent



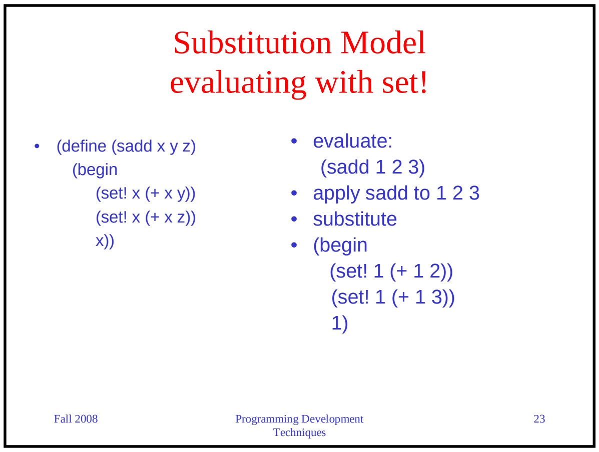
so far…

- before introducing set!
  - variable values did not change
  - intervening functions **never** changed the value of succeeding operations
- introduce set!
  - variable values may change
  - results of operations may depend on previous operations



evaluating with set!

-
  ```
  (define (sadd x y z)
    (begin
        (set! x (+ x y))
        (set! x (+ x z))
        x))
  ```
- intuitively: what does this do?

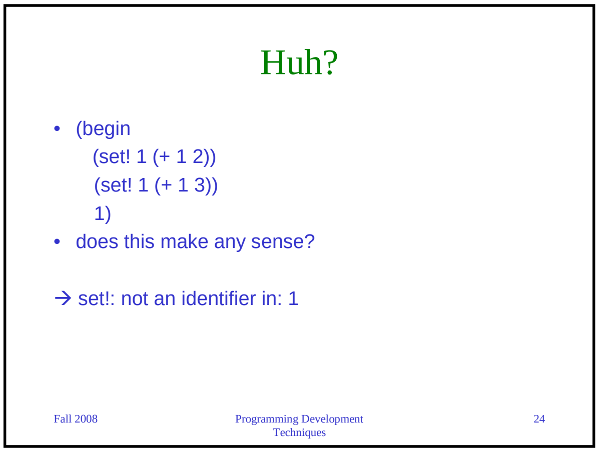


Substitution Model evaluating with set!

-
  ```
  (define (sadd x y z)
    (begin
       (set! x (+ x y))
       (set! x (+ x z))
       x))
  ```

- evaluate:
  (sadd 1 2 3)
- apply sadd to 1 2 3
- substitute

-
  ```
  (begin
    (set! 1 (+ 1 2))
    (set! 1 (+ 1 3))
    1)
  ```



Huh?

-
  ```
  (begin
    (set! 1 (+ 1 2))
    (set! 1 (+ 1 3))
    1)
  ```
- does this make any sense?

→ set!: not an identifier in: 1

problem

```
• (define (sadd x y z)
    (begin
      (set! x (+ x y))
      (set! x (+ x z))
      x))
```

```
• substitute
• (begin
    (set! 1 (+ 1 2))
    (set! 1 (+ 1 3))
    1)
```

our substitution model does not admit the possibility that a variable’s value might change

problem

```
• (define (sadd x y z)
    (begin
      (set! x (+ x y))
      (set! x (+ x z))
      x))
```

```
• substitute
• (begin
    (set! 1 (+ 1 2))
    (set! 1 (+ 1 3))
    1)
```

our substitution model does not distinguish between a value and a variable

the bottom line

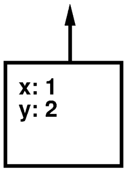
- the substitution model
  - breaks down in the presence of side-effects
  - cannot handle change of variable's value
- *we need a better model...*

the environment model

- new model
  - need to reason about variables as **locations**
- recall that we said **define** created an "association"
  - a mapping between a variable and a value
- now need to bring that to the forefront of our model

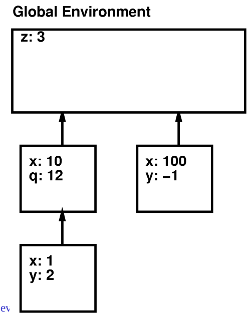
frames

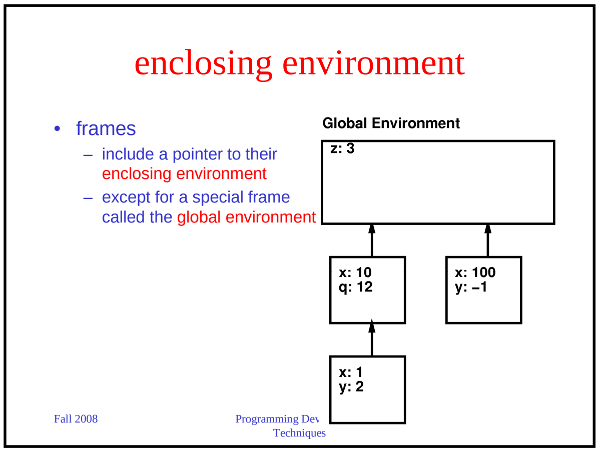
- we call the association table a frame
- a frame contains bindings
  - mapping from a variable name
  - to a value
- e.g.
  - x is currently 1
  - y is currently 2



environment

- An environment
  - is a collection of linked frames





enclosing environment

- frames
  - include a pointer to their enclosing environment
  - except for a special frame called the global environment

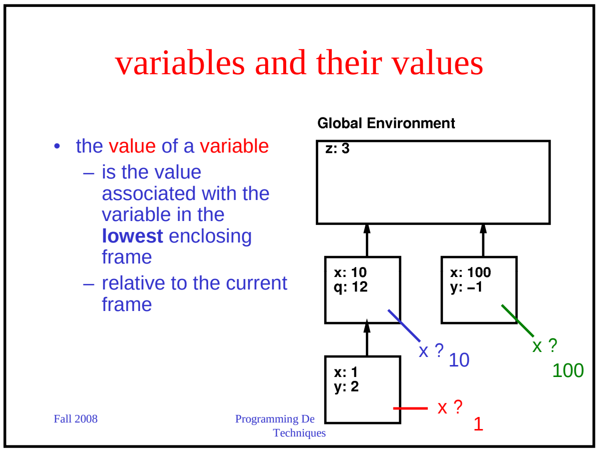
Global Environment

z: 3

x: 10
q: 12

x: 100
y: –1

x: 1
y: 2



variables and their values

- the value of a variable
  - is the value associated with the variable in the **lowest** enclosing frame
  - relative to the current frame

Global Environment

z: 3

x: 10
q: 12

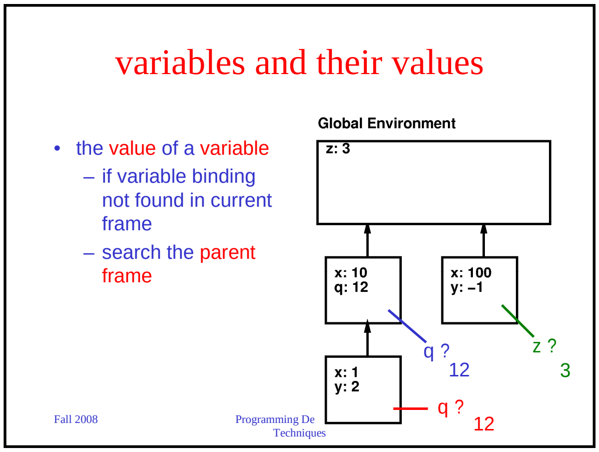
x: 100
y: –1

x: 1
y: 2

x ? 10

x ? 100

x ? 1



variables and their values

- the value of a variable
  - if variable binding not found in current frame
  - search the parent frame

Global Environment

z: 3

x: 10
q: 12

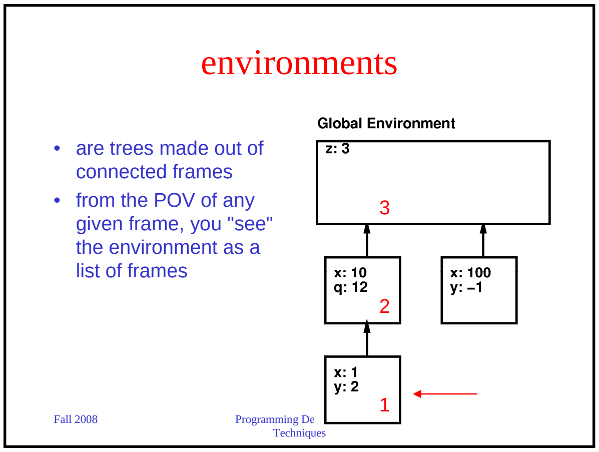
x: 100
y: –1

x: 1
y: 2

q ? 12

z ? 3

q ? 12



environments

- are trees made out of connected frames
- from the POV of any given frame, you "see" the environment as a list of frames

Global Environment

z: 3

3

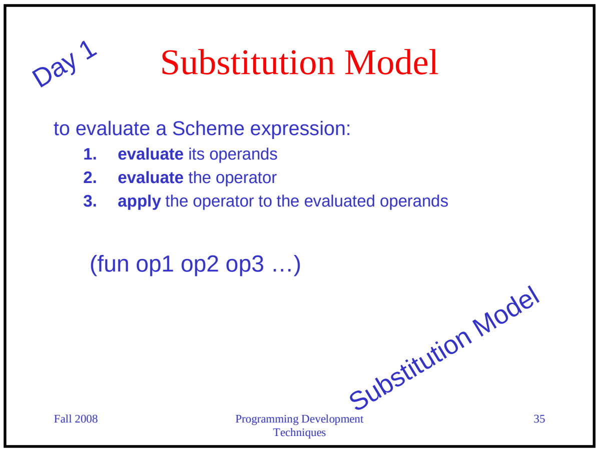
x: 10
q: 12

2

x: 100
y: –1

x: 1
y: 2

1



Day 1

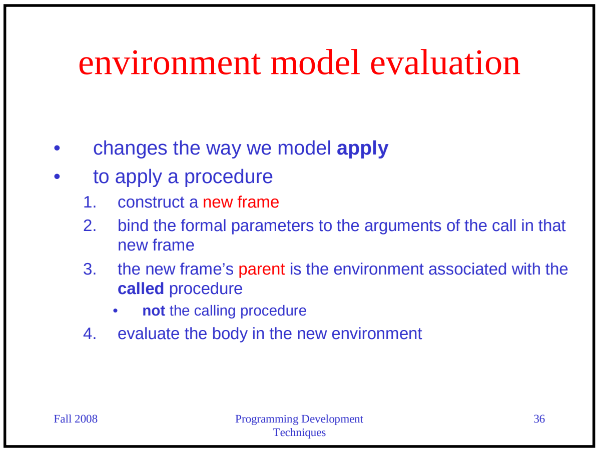
Substitution Model

to evaluate a Scheme expression:

1. **evaluate** its operands
2. **evaluate** the operator
3. **apply** the operator to the evaluated operands

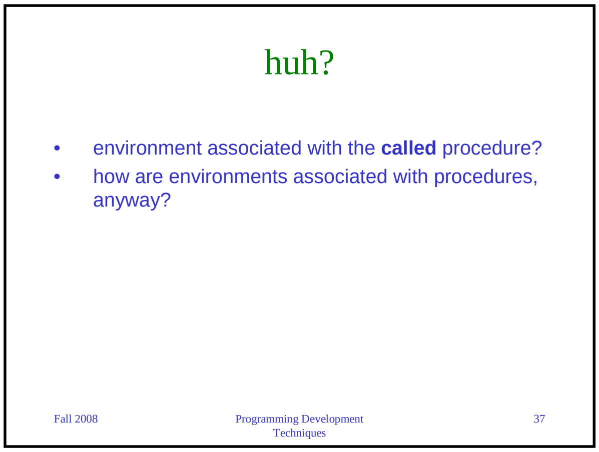
(fun op1 op2 op3 …)

Substitution Model



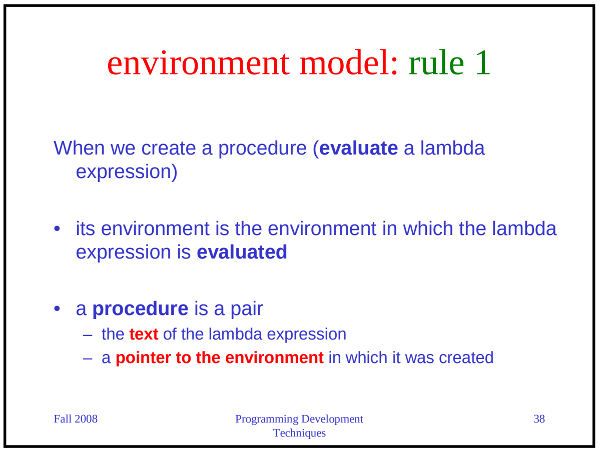
environment model evaluation

- changes the way we model **apply**
- to apply a procedure
  1. construct a new frame
  2. bind the formal parameters to the arguments of the call in that new frame
  3. the new frame's parent is the environment associated with the **called** procedure
     - **not** the calling procedure
  4. evaluate the body in the new environment



huh?

- environment associated with the **called** procedure?
- how are environments associated with procedures, anyway?

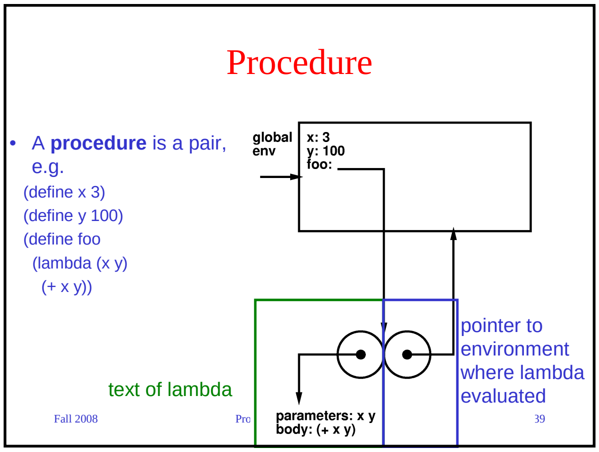


environment model: rule 1

When we create a procedure (**evaluate** a lambda expression)

- its environment is the environment in which the lambda expression is **evaluated**

- a **procedure** is a pair
  - the **text** of the lambda expression
  - a **pointer to the environment** in which it was created



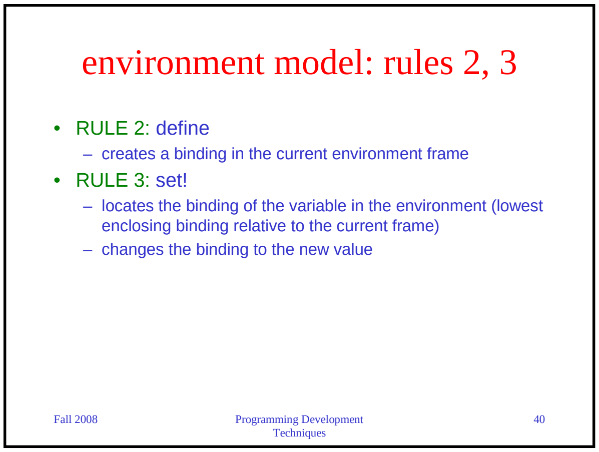
Procedure

- A **procedure** is a pair, e.g.

(define x 3)
(define y 100)
(define foo
(lambda (x y)
(+ x y))

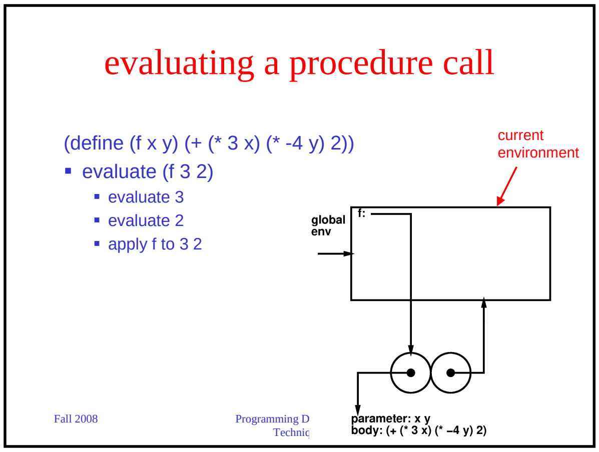
text of lambda

pointer to environment where lambda evaluated



environment model: rules 2, 3

- RULE 2: define
  - creates a binding in the current environment frame
- RULE 3: set!
  - locates the binding of the variable in the environment (lowest enclosing binding relative to the current frame)
  - changes the binding to the new value

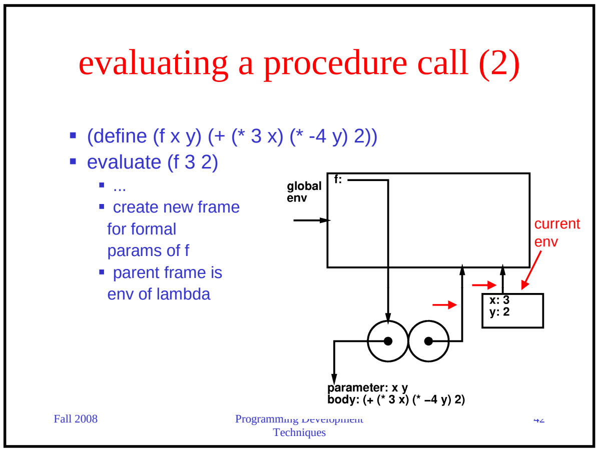


evaluating a procedure call

(define (f x y) (+ (* 3 x) (* -4 y) 2))

- evaluate (f 3 2)
  - evaluate 3
  - evaluate 2
  - apply f to 3 2

current environment



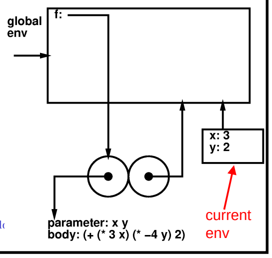
evaluating a procedure call (2)

- (define (f x y) (+ (* 3 x) (* -4 y) 2))
- evaluate (f 3 2)
  - ...
  - create new frame for formal params of f
  - parent frame is env of lambda

current env

evaluating a procedure call (3)

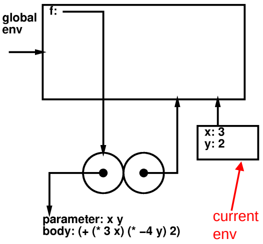
- (define (f x y) (+ (* 3 x) (* -4 y) 2))
- evaluate (f 3 2)
  - ...
  - evaluate body in new frame
  - (+ (* 3 x) (* -4 y) 2)


global env
f:
x: 3
y: 2
parameter: x y
body: (+ (* 3 x) (* –4 y) 2)
current env

evaluating a procedure call (4)

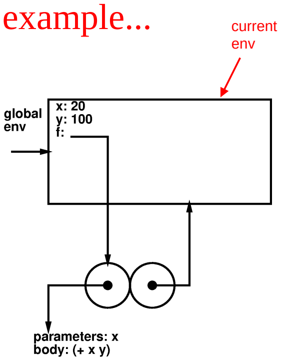
- (define (f x y) (+ (* 3 x) (* -4 y) 2))
- evaluate (f 3 2)
  - ...
  - (+ (* 3 x) (* -4 y) 2)
  - (+ (* 3 3) (* -4 2) 2)
  - (+ 9 -8 2)
  - 3

lookup not substitution


global env
f:
x: 3
y: 2
parameter: x y
body: (+ (* 3 x) (* –4 y) 2)
current env

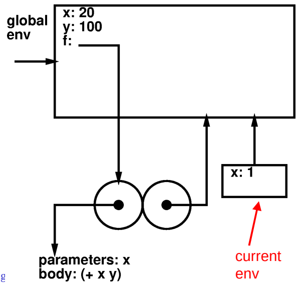
another example...

- (define x 20)
- (define y 100)
- (define (f x) (+ x y))
- (f 1)


current env
global env
x: 20
y: 100
f:
parameters: x
body: (+ x y)

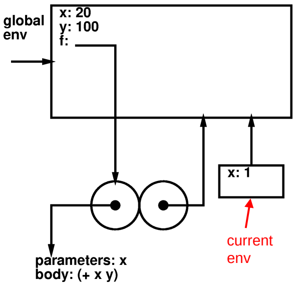
cont'd... (2)

- (define x 20)
- (define y 100)
- (define (f x) (+ x y))
- (f 1)
- create new frame for f
  - and bind call arguments to formal params


global env
x: 20
y: 100
f:
x: 1
parameters: x
body: (+ x y)
current env

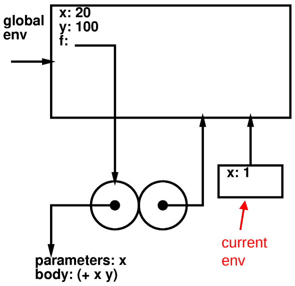
cont'd... (3)

- (define x 20)
- (define y 100)
- (define (f x) (+ x y))
- (f 1)
- create new frame for f
  - and bind call arguments to formal params
- evaluate body: (+ x y)


global env
x: 20
y: 100
f:
x: 1
parameters: x
body: (+ x y)
current env

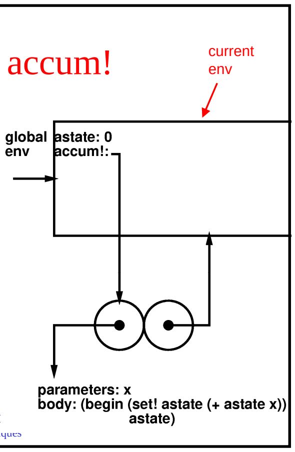
cont'd... (4)

- evaluate: (+ x y)
  - evaluate + → +
  - evaluate x → 1
  - evaluate y → 100
- (+ 1 100)
- 101


global env
x: 20
y: 100
f:
x: 1
parameters: x
body: (+ x y)
current env

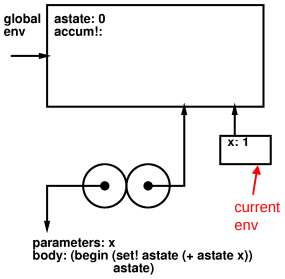
evaluate accum!

- (define astate 0)
- (define (accum! x)
  (begin
  (set! astate (+ astate x))
  astate))
- (accum! 1)
- (accum! 1) ; again!


current env
global env
astate: 0
accum!:
parameters: x
body: (begin (set! astate (+ astate x))
astate)

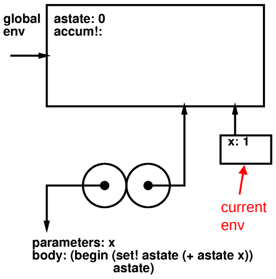
evaluate accum! (2)

- (define (accum! x) …)
- (accum! 1)
  - create call frame
  - bind formal params


global env
astate: 0
accum!:
x: 1
current env
parameters: x
body: (begin (set! astate (+ astate x))
astate)

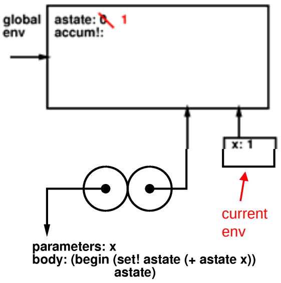
evaluate accum! (3)

- evaluate:
  - (begin (set! astate (+ astate x))
    astate)
  - evaluate first expression
    - (set! astate
      (+ astate x))
    - evaluate 2nd arg.
      - (+ astate x)
      - (+ 0 1)
      - 1


global env
astate: 0
accum!:
x: 1
current env
parameters: x
body: (begin (set! astate (+ astate x))
astate)

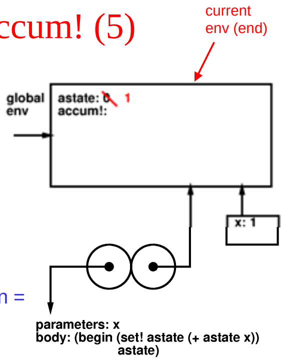
evaluate accum! (4)

- evaluate:
  - (begin (set! astate (+ astate x))
    astate)
  - evaluate first expression
    - (set! astate …)
    - evaluate 2nd arg.
    - → 1
    - update binding of first argument (variable astate)

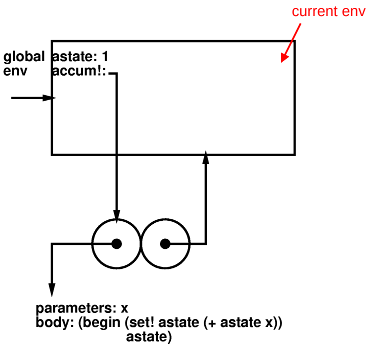

global env
astate: ~~0~~ 1
accum!:
x: 1
current env
parameters: x
body: (begin (set! astate (+ astate x))
astate)

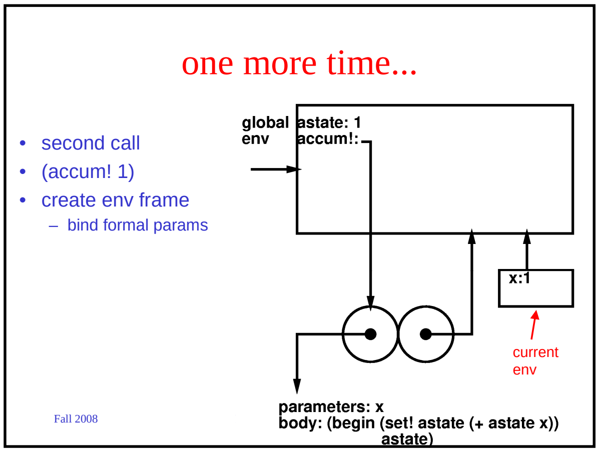
evaluate accum! (5)

- evaluate:
  - (begin … astate)
  - evaluate first expr
  - evaluate second expr
    - astate
    - 1
- return value of final expression = 1


current env (end)
global env
astate: ~~0~~ 1
accum!:
x: 1
parameters: x
body: (begin (set! astate (+ astate x))
astate)

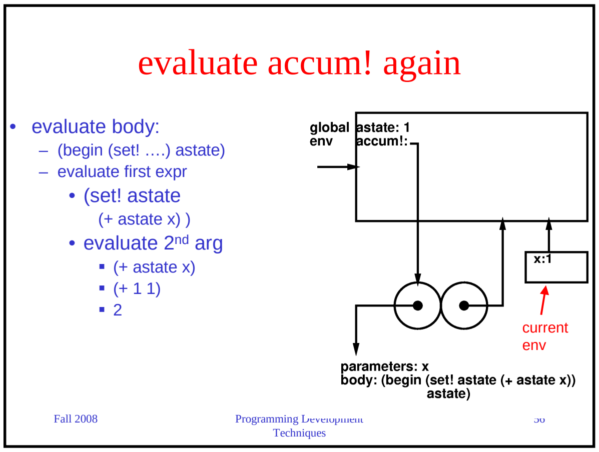
after first (accum! 1)


current env
global env
astate: 1
accum!:
parameters: x
body: (begin (set! astate (+ astate x))
astate)



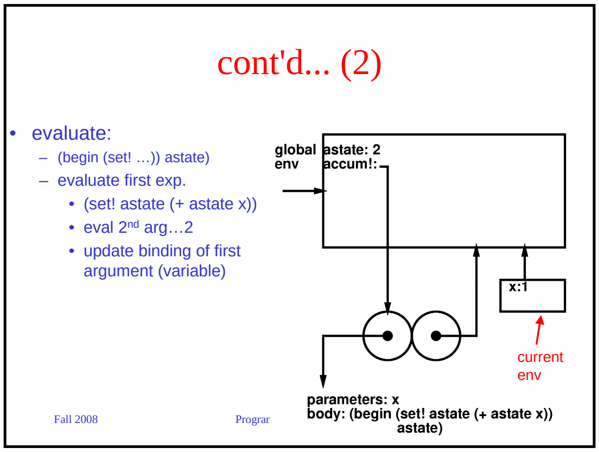
one more time...

- second call
- (accum! 1)
- create env frame
  - bind formal params



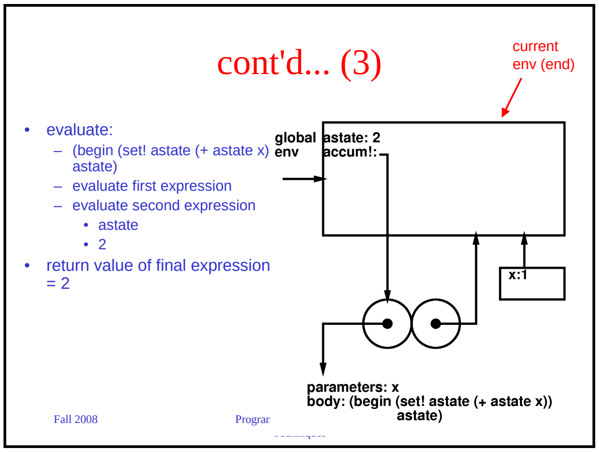
evaluate accum! again

- evaluate body:
  - (begin (set! ….) astate)
  - evaluate first expr
    - (set! astate
      (+ astate x))
    - evaluate 2nd arg
      - (+ astate x)
      - (+ 1 1)
      - 2



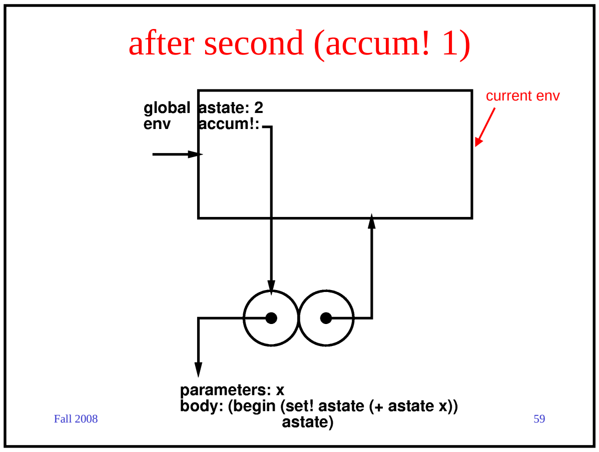
cont'd... (2)

- evaluate:
  - (begin (set! …)) astate)
  - evaluate first exp.
    - (set! astate (+ astate x))
    - eval 2nd arg…2
    - update binding of first argument (variable)

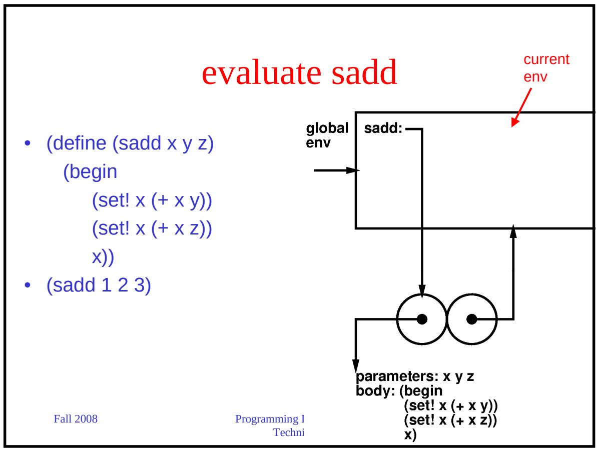


cont'd... (3)

- evaluate:
  - (begin (set! astate (+ astate x) astate)
  - evaluate first expression
  - evaluate second expression
    - astate
    - 2
- return value of final expression = 2

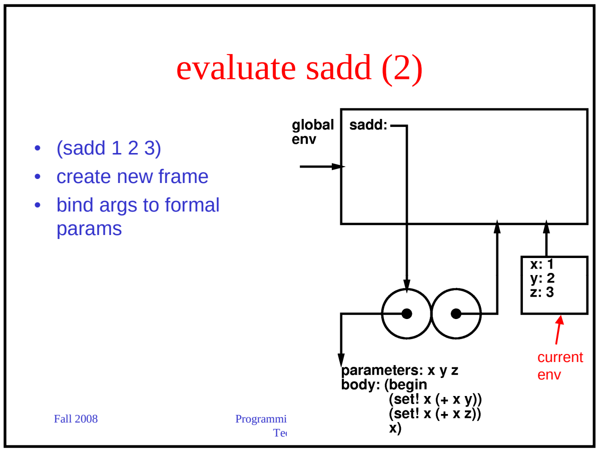


after second (accum! 1)



evaluate sadd

- (define (sadd x y z)
  (begin
  (set! x (+ x y))
  (set! x (+ x z))
  x))
- (sadd 1 2 3)



evaluate sadd (2)

- (sadd 1 2 3)
- create new frame
- bind args to formal params

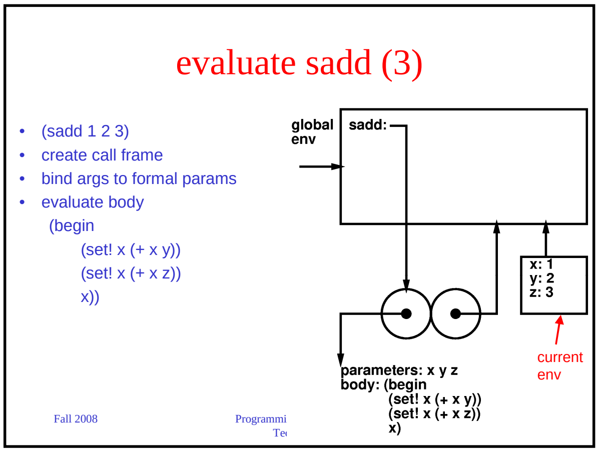
global env

sadd:

x: 1
y: 2
z: 3

current env

parameters: x y z
body: (begin
(set! x (+ x y))
(set! x (+ x z))
x)



evaluate sadd (3)

- (sadd 1 2 3)
- create call frame
- bind args to formal params
- evaluate body

```
(begin
    (set! x (+ x y))
    (set! x (+ x z))
    x))
```

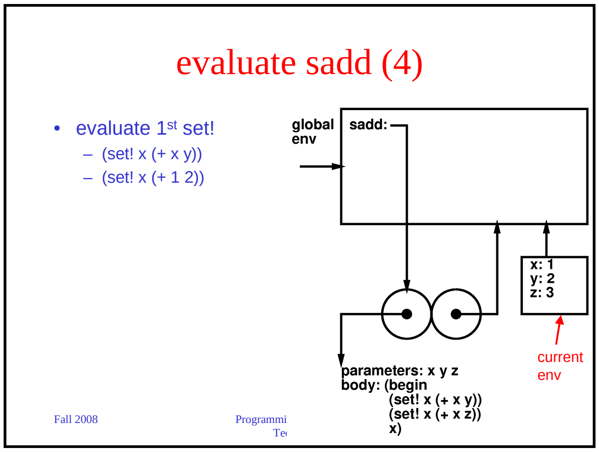
global env

sadd:

x: 1
y: 2
z: 3

current env

parameters: x y z
body: (begin
(set! x (+ x y))
(set! x (+ x z))
x)



evaluate sadd (4)

- evaluate 1st set!
  - (set! x (+ x y))
  - (set! x (+ 1 2))

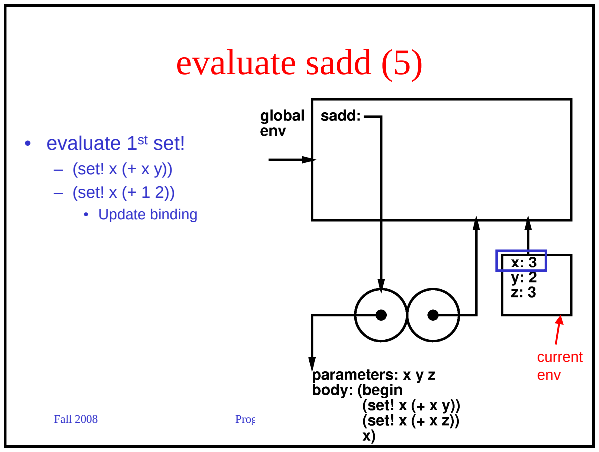
global env

sadd:

x: 1
y: 2
z: 3

current env

parameters: x y z
body: (begin
(set! x (+ x y))
(set! x (+ x z))
x)



evaluate sadd (5)

- evaluate 1st set!
  - (set! x (+ x y))
  - (set! x (+ 1 2))
    - Update binding

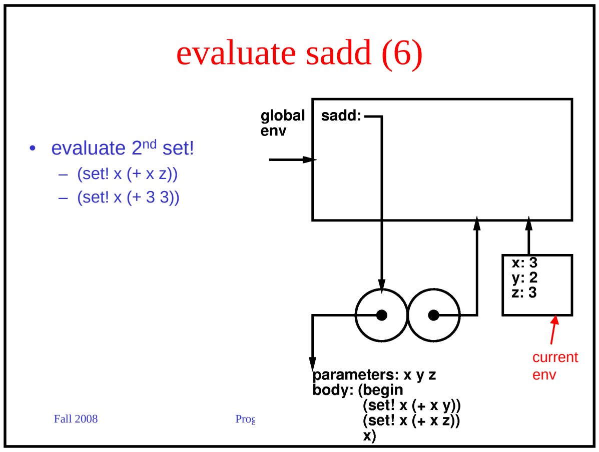
global env

sadd:

x: 3
y: 2
z: 3

current env

parameters: x y z
body: (begin
(set! x (+ x y))
(set! x (+ x z))
x)



evaluate sadd (6)

- evaluate 2nd set!
  - (set! x (+ x z))
  - (set! x (+ 3 3))

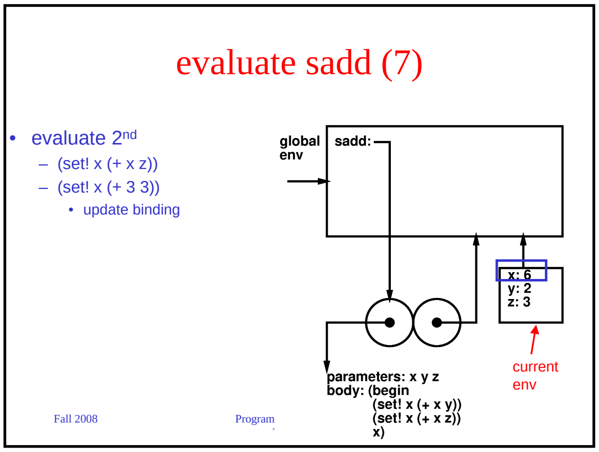
global env

sadd:

x: 3
y: 2
z: 3

current env

parameters: x y z
body: (begin
(set! x (+ x y))
(set! x (+ x z))
x)



evaluate sadd (7)

- evaluate 2nd
  - (set! x (+ x z))
  - (set! x (+ 3 3))
    - update binding

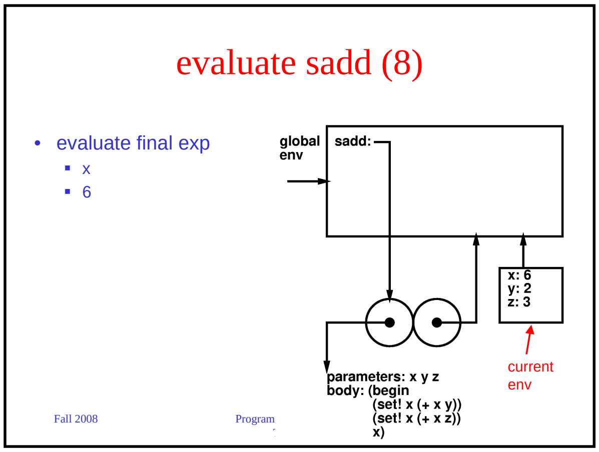
global env

sadd:

x: 6
y: 2
z: 3

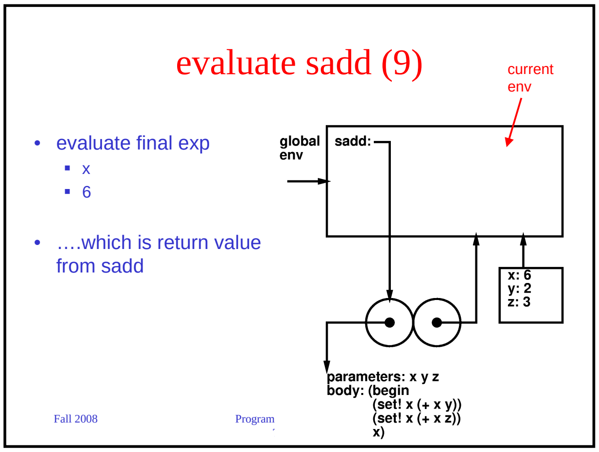
current env

parameters: x y z
body: (begin
(set! x (+ x y))
(set! x (+ x z))
x)



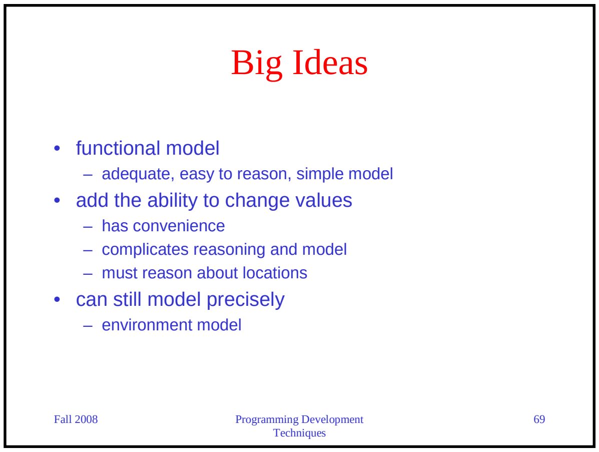
evaluate sadd (8)

- evaluate final exp
  - x
  - 6



evaluate sadd (9)

- evaluate final exp
  - x
  - 6
- ….which is return value from sadd



Big Ideas

- functional model
  - adequate, easy to reason, simple model
- add the ability to change values
  - has convenience
  - complicates reasoning and model
  - must reason about locations
- can still model precisely
  - environment model