Topic 18
Environment Model of Evaluation

Section 3.2

Acknowledgement: This lecture (and also much of the previous one) were taken from an old CalTech Course Page

bindings (review)

- a binding is an association between a name and a Scheme value
- names:
  - variable names, procedure names
  - formal parameters of procedures
  - in let statements: (let ((name value) ...) ...)
- values: any Scheme value

bindings (review)

- a binding is an association between a name and a Scheme value
- examples:
  - name: x value: 10
  - name: y value: #f
  - name: square value: (lambda (x) (* x x))

frames (review)

- a frame is a collection of bindings:

x: 10
y: #f
square: (lambda (x) (* x x))

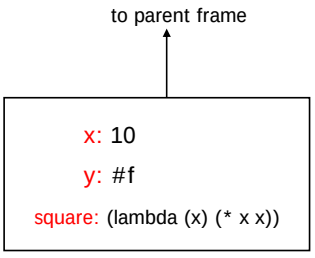
frames (review)

- frames are used to look up the value corresponding to a name
  - x = ?
  - y = ?
  - square = ?

x: 10
y: #f
square: (lambda (x) (* x x))

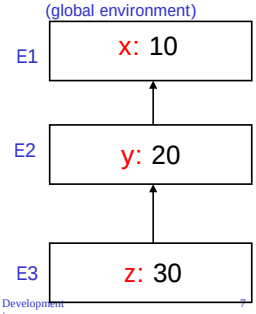
frames (review)

- frames have an enclosing environment
  - which will be another frame (parent frame)
  - if lookup fails, go to parent frame and try again
  - then to *its* parent frame etc.



environments (review)

- an environment is a linked chain of frames ending in the global environment
- every frame defines an environment starting from it



environments (review)

- the global environment is there when Scheme interpreter starts up
- evaluating code can create new frames
  - and thus new environments
- all code being evaluated does so in the context of an environment
  - because names must be looked up
  - called the current environment

rule 1: define

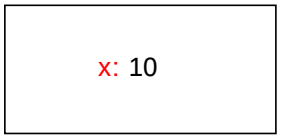
- define creates a new binding in the current environment (current frame)
- example:
- (define x 10)

(global environment)

rule 1: define

- define creates a new binding in the current environment (current frame)
- example:
- (define x 10)

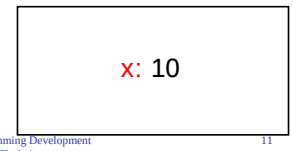
(global environment)



rule 2: set!

- set! changes an old binding in the current environment
- *NEVER* creates a new binding
- example:
- (set! x 'foo)

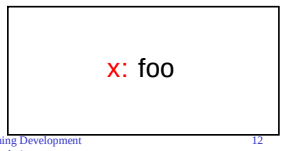
(global environment)



rule 2: set!

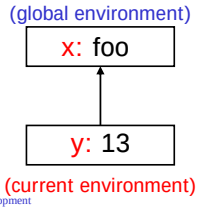
- set! changes an old binding in the current environment
- *NEVER* creates a new binding
- example:
- (set! x 'foo)

(global environment)



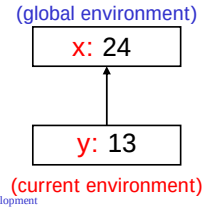
rule 2: set!

- set! can change a binding in a different frame (if no binding in current frame)
- example:
- (set! x 24)
  - no x in current env
  - so change in global env



rule 2: set!

- set! can change a binding in a different frame (if no binding in current frame)
- example:
- (set! x 24)
  - no x in current env
  - so change in global env

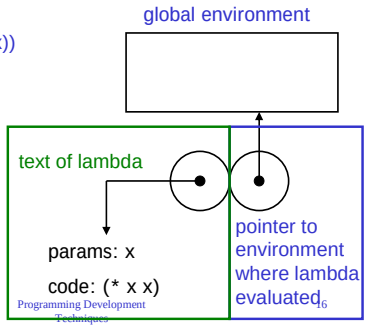


rule 3: lambda

- a lambda expression (procedure) is a pair:
  - the text of the lambda expression
  - a pointer to the environment in which the lambda expression was evaluated (created)

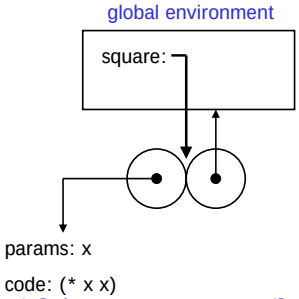
rule 3: lambda

- evaluate:
  (lambda (x) (* x x))



with define

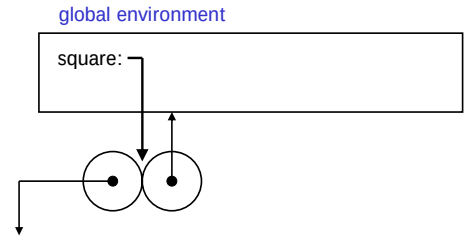
- (define square
  (lambda (x) (* x x)))



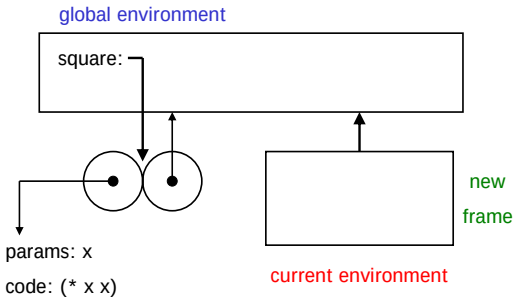
rule 4: evaluation

- changes the way we model **apply**
- To apply a procedure:
  1. construct a new frame
  2. bind the formal parameters of the procedure to the arguments of the procedure call
  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

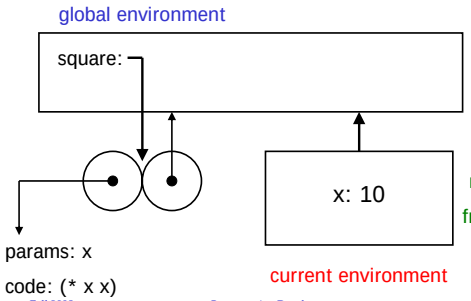
evaluating (square 10)



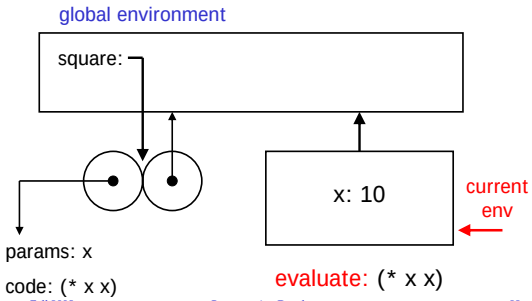
evaluating (square 10)



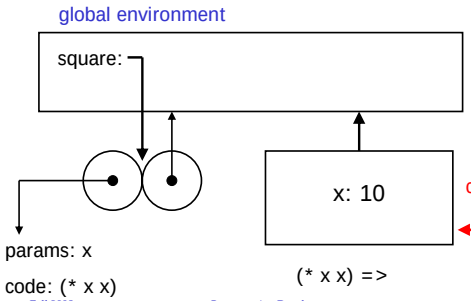
evaluating (square 10)



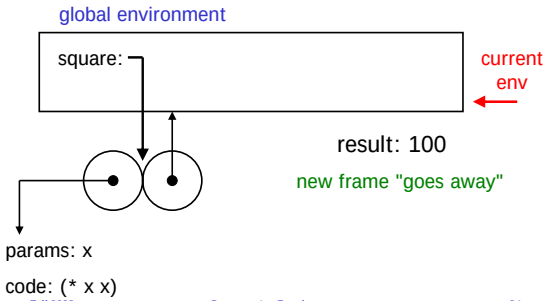
evaluating (square 10)



evaluating (square 10)



evaluating (square 10)



That was WAY too easy!

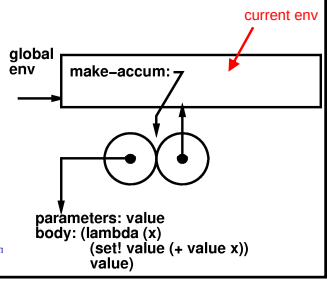
Let's try something harder...

accumulator (revisited)

- (**define** (make-accum value)
  (**lambda** (x)
  (**set!** value (**+** value x))
  value))
- *N.B.* body of **lambda**, **define**, **cond** clause is implicitly a **begin** block.

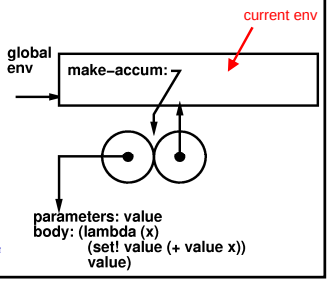
accumulator

- (**define** (make-accum value)
  (**lambda** (x)
  (**set!** value (**+** value x))
  value))



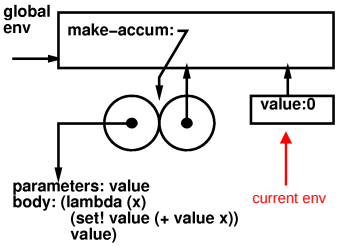
accumulator (another look)

- (**define** make-accum
  (**lambda** (value)
  (**lambda** (x)
  (**set!** value
  (**+** value x))
  value))



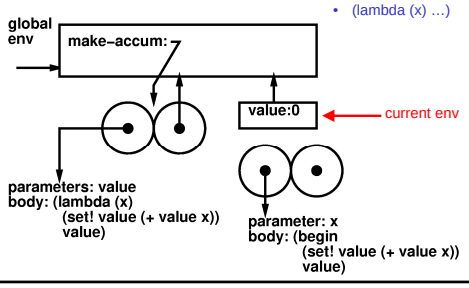
accumulator

- (**define** a! (make-accum 0))
  - evaluate (make-accum 0)
    - create frame
    - bind arguments



accumulator

- evaluate (make-accum 0)
  - create, bind
  - evaluate body
    - (lambda (x) …)

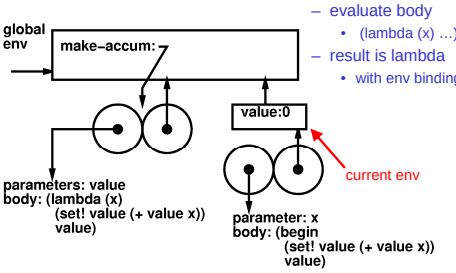


rule 3:

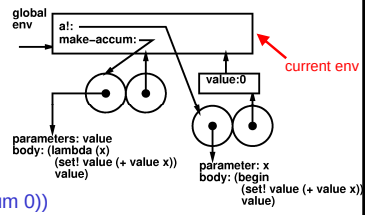
- when we create a procedure (evaluate a lambda expression)
  - its environment is the environment in which the lambda expression is evaluated

accumulator

- evaluate (make-accum 0)
  - create, bind
  - evaluate body
    - (lambda (x) …)
  - result is lambda
    - with env binding

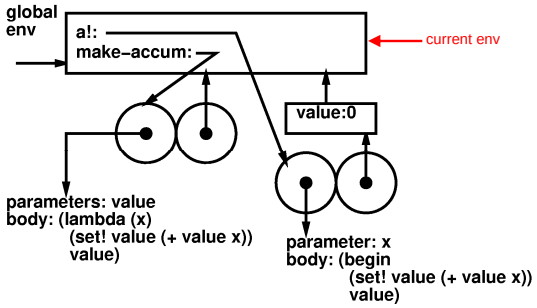


accumulator

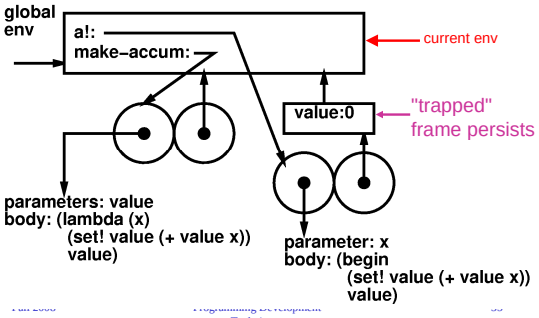


- (define a! (make-accum 0))
  - evaluate (make-accum 0)
    - result is (lambda (x) …) with env ptr
  - make binding for a!
    - in its environment

after (define a!…)

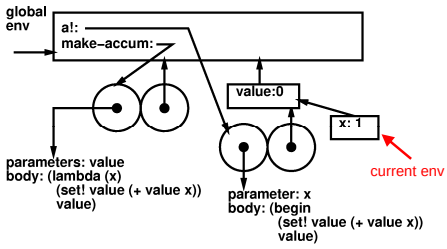


after (define a!…)



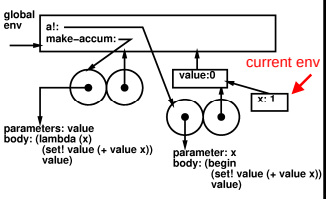
using a!

- (a! 1)
  - create frame
  - bind values



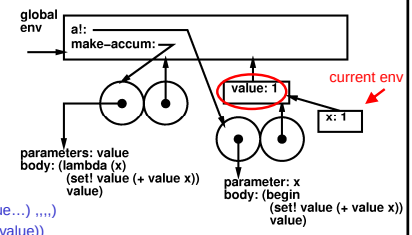
using a!

- (a! 1)
  - create frame
  - bind values
  - evaluate body
    - (begin (set! value…) …)
      - (set! value (+ x value))
      - (set! value (+ 1 0))
      - (set! value 1)

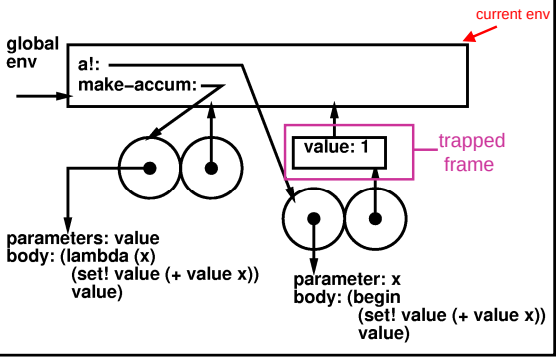


using a!

- (a! 1)
  - Create frame
  - Bind values
  - Evaluate body
    - (begin (set! value…) ,,,,)
    - (set! value (+ x value))
    - (set! value (+ 1 0))
    - (set! value 1)



environment after (a! 1)



more usage

- (a! 1)

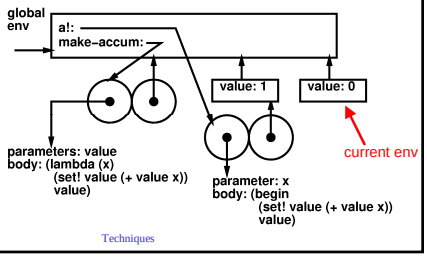
→ 2

- (a! 1)

→ 3

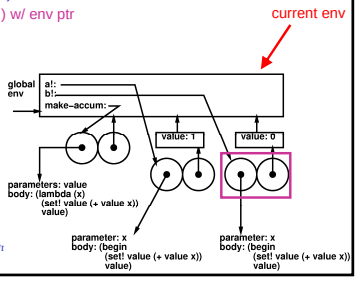
accumulator 2

- (define b! (make-accum 0))
  - evaluate (make-accum 0)
    - create
    - bind…



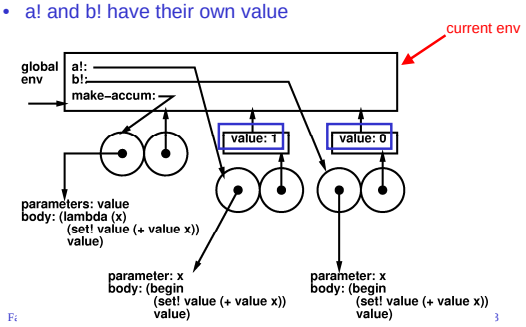
accumulator 2

- (define b! (make-accum 0))
  - evaluate (make-accum 0)
    - result is (lambda (x) …) w/ env ptr
  - make binding for b!



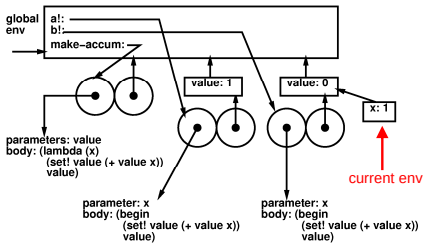
resulting environment

- a! and b! have their own value



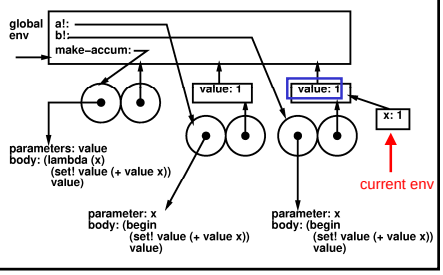
using b!

- (b! 1)
  - …frame



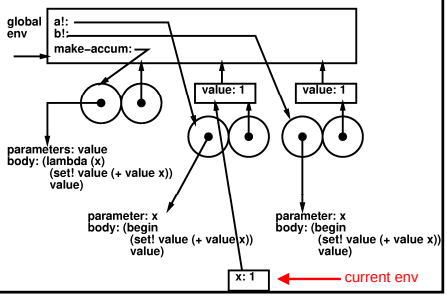
using b!

- (b! 1)
  - …frame
  - …set!



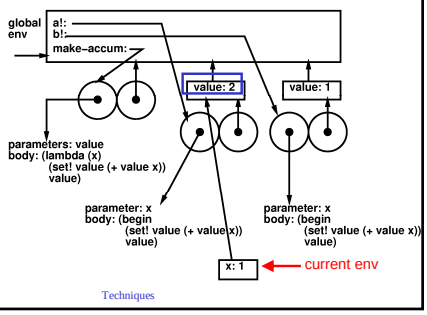
using a! again

- (a! 1)
  - …frame



using a! again

- (a! 1)
  - …frame
  - …set!

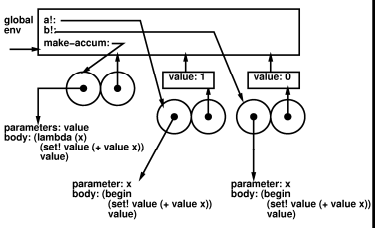


using b!

- (b! 1)
- → 2
- (a! 1)
- → 3
- (a! 1)
- → 4
- (b! 1)
- → 3

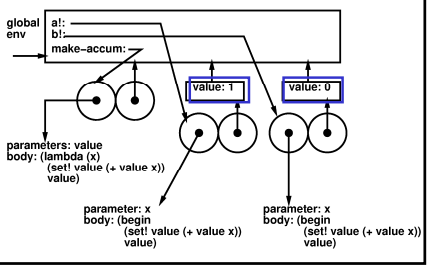
procedure environment

- `make-accum` creates a *unique* frame for each call
- the lambda **traps** the frame and keeps a pointer to it
- **only** that lambda has access to that frame



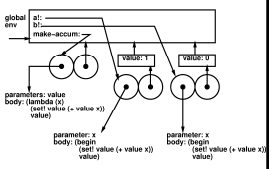
procedure environment

- **value** is a local variable that is owned (and accessible) exclusively by the resulting lambda



a! and b!

- a! and b! have unique procedure frames
- have unique local variables: value
- don't affect each other
- remember own state
- important form of encapsulation
  - limits **unwanted** interactions



contrast

- (define astate 0)
- (define (accum! x)
  (set! astate (+ x astate))
  astate))

- (define (toggle!)
  (if (= astate 0)
  (set! astate 1)
  (set! astate 0)))

interaction

- (accum! 1)
- → 1
- (toggle!) ;; astate: 0
- (accum! 1)
- → 1
- (accum! 1)
- → 2

- (toggle!) ;; astate: 0
- (accum! 1)
- → 1

- toggle! and accum! share a variable
- not independent

encapsulation

- generally a **bad idea** to put state in the global environment
  - pollutes namespace
  - gives other routines access
    - even ones that don't need it
  - can get bad interactions
    - even accidental ones

disciplined use of side-effects

- moral of the story:
- there are many things we **can** do with side-effects
- …that we really **should not** do.
- keep your state private!
  - nobody else has to know

controlling access

- What if I want multiple procedures to access the same state?
- want specificity
  - give multiple procedures access
  - but not all procedures

code and data

- encapsulate data
  - with associated (limited) code to manipulate
  - code mediates kinds of access
- familiar…
  - message-passing associated data and code

message-passing

```
; returns a procedure that will act as an accumulator
; or will return its value or reset the value
(define (mp-make-accum value)
  (lambda (op)
    (cond ((eq? op 'accum)
           (lambda (x) (set! value
                             (+ value x))))
          ((eq? op 'value) value)
          ((eq? op 'reset) (set! value 0))
          (else (error "unknown op: " op)))))
```


message-passing use

- (define a2 (mp-make-accum 0))
- (a2 'value)

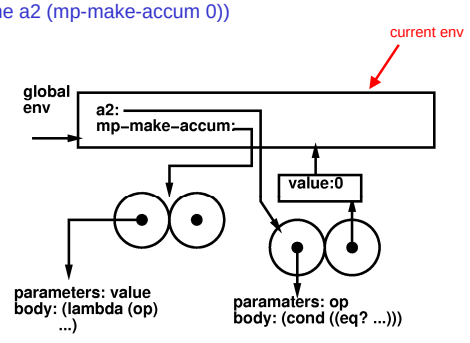
→ 0

- ((a2 'accum) 1)
- (a2 'value)

→ 1

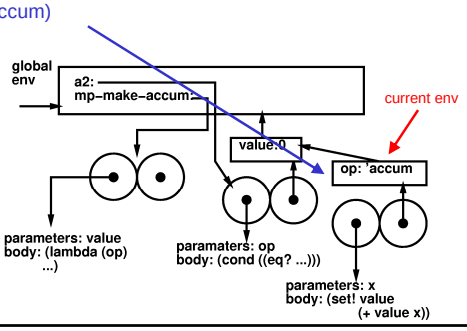
environment

- (define a2 (mp-make-accum 0))



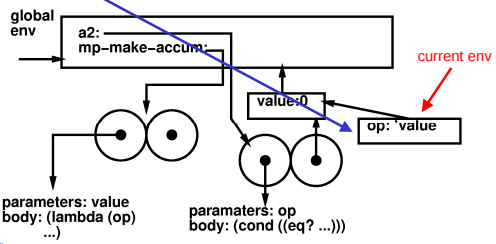
environment

- (define a2 (mp-make-accum 0))
- (a2 'accum)



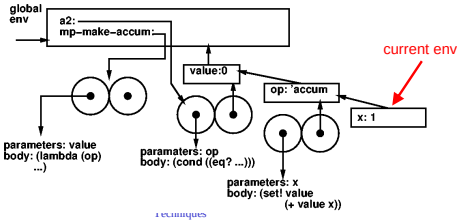
environment

- (define a2 (mp-make-accum 0))
- (a2 'value)



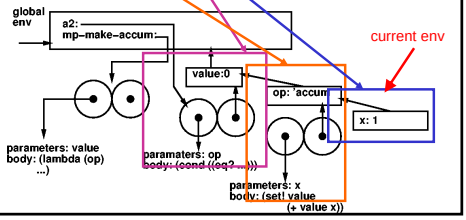
usage

- (define a2 (mp-make-accum 0))
- during: ((a2 'accum) 1)



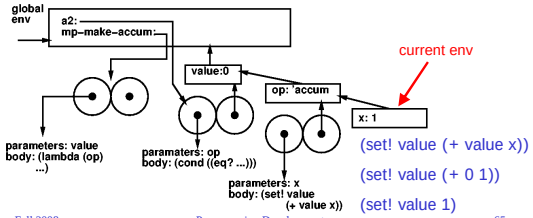
usage

- (define a2 (mp-make-accum 0))
- during: ((a2 'accum) 1)



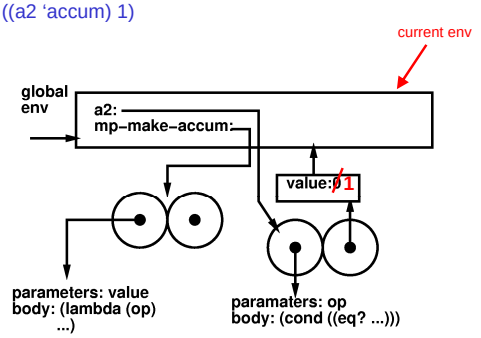
usage

- (define a2 (mp-make-accum 0))
- during: ((a2 'accum) 1)



usage

- after: ((a2 'accum) 1)



Big Ideas

- procedures point to environment at the place where they're **evaluated**
- procedures can have local state
- allows us to create procedure-specific (object-specific) state
- allows us to encapsulate data
  - avoiding conflict
  - control usage