

### 6.001 SICP

#### Environment model

- Models of computation

- Substitution model

- A way to figure out what happens during evaluation

- (define l '(a b c))
      - (car l) → a
      - (define m '(1 2 3))
      - (car l) → a

- Not really what happens in the computer

- (car l) → a
      - (set-car! l 'z)
      - (car l) → z

- The Environment Model



1

### Can you figure out why this code works?

```
(define make-counter  
  (lambda (n)  
    (lambda () (set! n (+ n 1))  
              n ))))
```

```
(define ca (make-counter 0))
```

```
(ca) ==> 1
```

```
(ca) ==> 2
```

```
(define cb (make-counter 0))
```

```
(cb) ==> 1
```

```
(ca) ==> 3 ; ca and cb are independent
```

2

### What the EM is:

- A precise, completely mechanical description of:

- name-rule            looking up the value of a variable
  - define-rule        creating a new definition of a var
  - set!-rule            changing the value of a variable
  - lambda-rule        creating a procedure
  - application        applying a procedure

- Enables analyzing more complex scheme code:

- Example: **make-counter**

- Basis for implementing a scheme interpreter

- for now: draw EM state with boxes and pointers
  - later on: implement with code

3

### A shift in viewpoint

- As we introduce the environment model, we are going to shift our viewpoint on computation

- Variable:

- OLD – name for value
  - NEW – place into which one can store things

- Procedure:

- OLD – functional description
  - NEW – object with inherited context

- Expressions

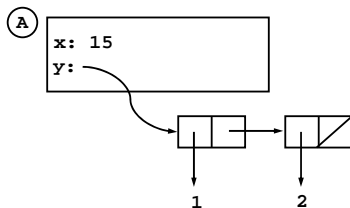
- Now **only** have meaning with respect to an environment

4

### Frame: a table of bindings

- Binding:** a pairing of a name and a value

Example: **x** is bound to 15 in frame A  
**y** is bound to (1 2) in frame A  
the value of the variable **x** in frame A is 15



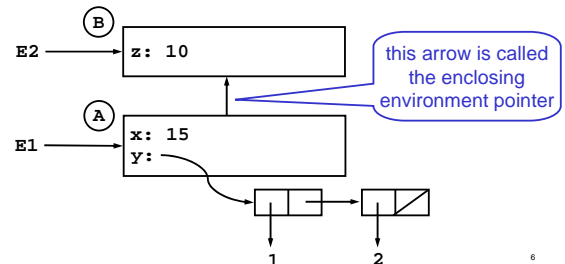
5

### Environment: a sequence of frames

- Environment E1 consists of frames A and B

- Environment E2 consists of frame B only

- A frame may be shared by multiple environments



6

### Evaluation in the environment model

- All evaluation occurs **in an environment**
  - The **current environment** changes when the interpreter applies a procedure
- The top environment is called the **global environment (GE)**
  - Only the GE has no enclosing environment
- To **evaluate** a combination
  - Evaluate the subexpressions *in the current environment*
  - Apply the value of the first to the values of the rest

7

### Name-rule

- A name X evaluated in environment E gives **the value of X in the first frame of E where X is bound**
- In E1, the binding of x in frame A **shadows** the binding of x in B

8

### Define-rule

- A define special form evaluated in environment E **creates or replaces a binding in the first frame of E**

```
(define z 20) |GE      (define z 25) |E1
```

10

### Set!-rule

- A set! of variable X evaluated in environment E **changes the binding of X in the first frame of E where X is bound**

```
(set! z 20) |GE      (set! z 25) |E1
```

11

### Your turn: evaluate the following in order

```
(+ z 1) |E1           ==>           11
(set! z (+ z 1)) |E1   (modify EM)
(define z (+ z 1)) |E1   (modify EM)
(set! y (+ z 1)) |GE   (modify EM)
```

13

### Double bubble: how to draw a procedure

```
(lambda (x) (* x x))
```

14

### Lambda-rule

- A lambda special form evaluated in environment E creates a procedure whose environment pointer is E

```
(define square (lambda (x) (* x x))) |E1
```

environment pointer points to frame A because the lambda was *evaluated in E1* and  $E1 \rightarrow A$

Evaluating a lambda actually returns a pointer to the procedure object

parameters: x  
body: (\* x x)

15

### To apply a compound procedure P to arguments:

- Create a new frame A
- Make A into an environment E: A's enclosing environment pointer goes to the same frame as the environment pointer of P
- In A, bind the parameters of P to the argument values
- Evaluate the body of P with E as the current environment

16

```
(square 4) |GE
```

parameters: x  
body: (\* x x)

square |<sub>GE</sub> ==> #[proc]

( \* x x ) |<sub>E1</sub> ==> 16

\* |<sub>E1</sub> ==> #[prim]

x |<sub>E1</sub> ==> 4

18

### Example: inc-square

inc-square: procedure object  
square: procedure object

p: x  
b: (\* x x)

p: y  
b: (+ 1 (square y))

```
(define square (lambda (x) (* x x))) |GE  
(define inc-square (lambda (y) (+ 1 (square y)))) |GE
```

19

### Example cont'd: (inc-square 4) |<sub>GE</sub>

inc-square: procedure object  
square: procedure object

p: x  
b: (\* x x)

p: y  
b: (+ 1 (square y))

y: 4

inc-square |<sub>GE</sub> ==> #[compound-proc ...]

20

### Example cont'd: (square y) |<sub>E1</sub>

inc-square: procedure object  
square: procedure object

p: x  
b: (\* x x)

p: y  
b: (+ 1 (square y))

y: 4

x: 4

square |<sub>E1</sub> ==> #[compound]    y |<sub>E1</sub> ==> 4

( \* x x ) |<sub>E2</sub> ==> 16    (+ 1 16) ==> 17

\* |<sub>E2</sub> ==> #[prim]    x |<sub>E2</sub> ==> 4

21

### Lessons from the inc-square example

- EM doesn't show the complete state of the interpreter
  - missing the stack of pending operations
- The GE contains all standard bindings (\*, cons, etc)
  - omitted from EM drawings
- Useful to link environment pointer of each frame to the procedure that created it

22

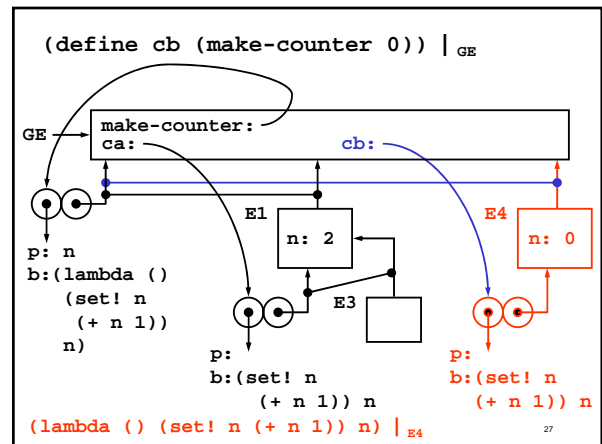
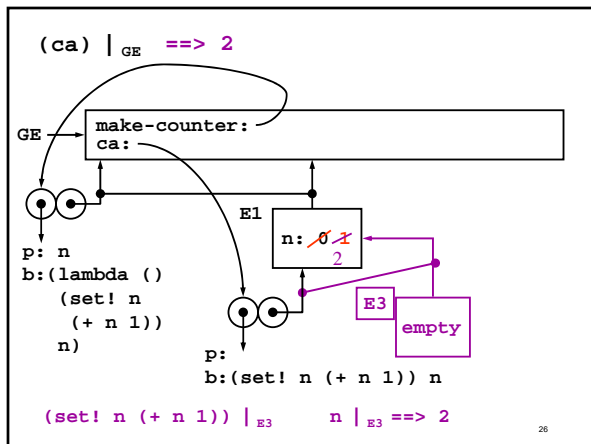
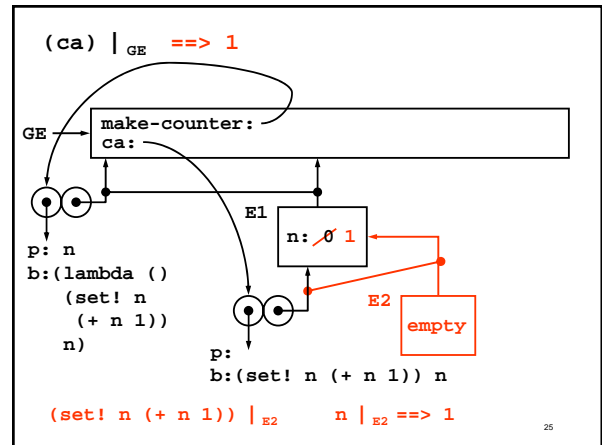
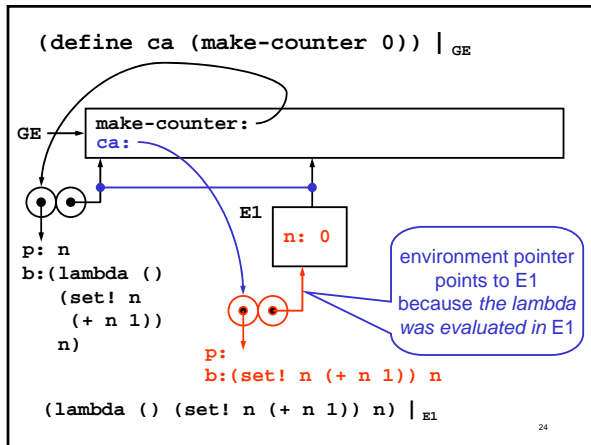
### Example: make-counter

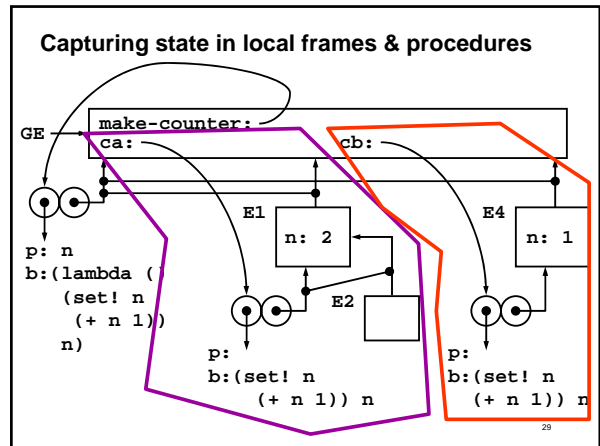
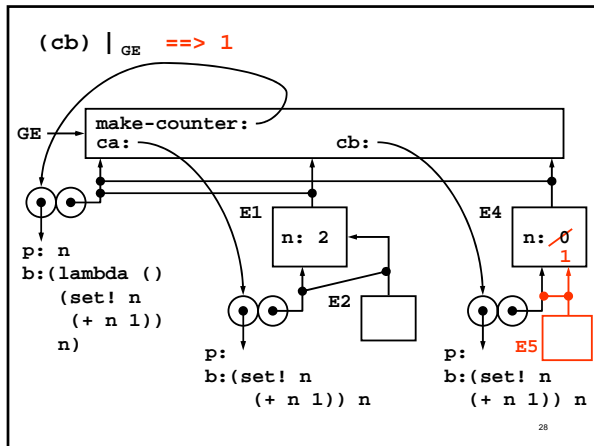
- Counter: something which counts up from a number

```
(define make-counter
  (lambda (n)
    (lambda () (set! n (+ n 1))
              n)
  ))
```

```
(define ca (make-counter 0))
(ca) ==> 1
(ca) ==> 2
(define cb (make-counter 0))
(cb) ==> 1
(ca) ==> 3
(cb) ==> 2 ; ca and cb are independent
```

23





### Lessons from the make-counter example

- Environment diagrams get complicated very quickly
  - Rules are meant for the computer to follow, not to help humans
- A lambda inside a procedure body captures the frame that was active when the lambda was evaluated
  - this effect can be used to store **local state**

GE

make-counter:  
ca:                      cb:

p: n  
b: (lambda (n)  
     (set! n  
       (+ n 1))  
     n)

E1    n: 2

E2

E4    n: 1

p: (set! n  
     (+ n 1)) n

p: (set! n  
     (+ n 1)) n

30