6.001 SICP
Environment model

```
        ; 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
```


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 arbitrary 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

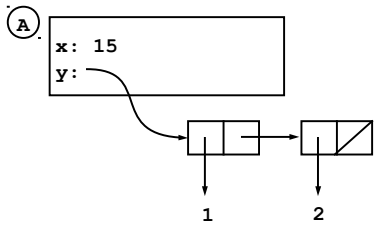
A shift in viewpoint

- 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

Frame: a table of bindings

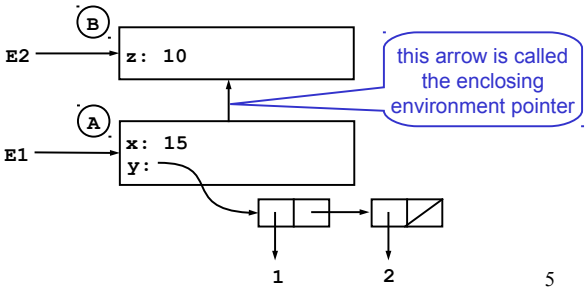
- Binding: a pairing of a name and a value

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



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



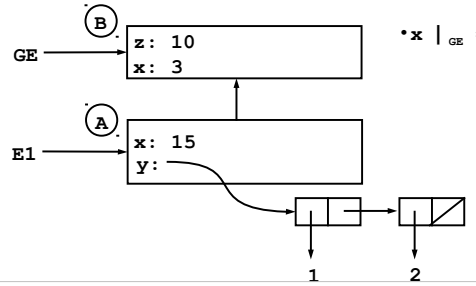
Evaluation in the environment model

- 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

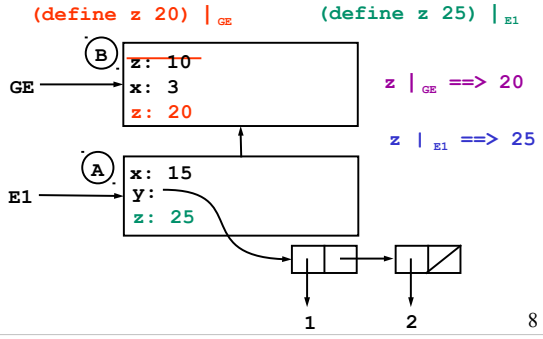
Name-rule

- A name X evaluated in environment E gives the value of X in the first frame of E where X is bound
  - `z |GE ==> 10` `z |E1 ==> 10` `x |E1 ==> 15`
- In E1, the binding of x in frame A shadows the binding of x in B



Define-rule

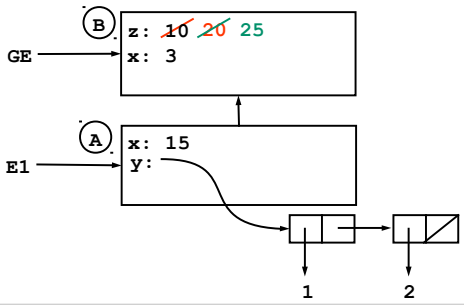
- A define special form evaluated in environment E creates or replaces a binding in the first frame of E



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`



Your turn: evaluate the following in order

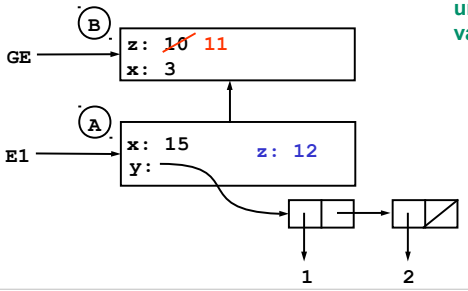
`(+ z 1) |E1` ==> 11

`(set! z (+ z 1)) |E1` (modify EM)

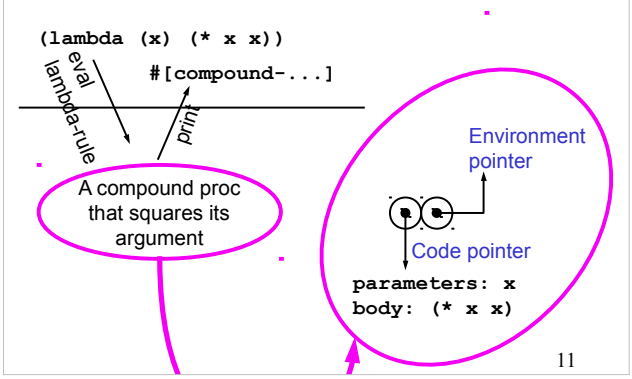
`(define z (+ z 1)) |E1` (modify EM)

`(set! y (+ z 1)) |GE` (modify EM)

Error: unbound variable: y



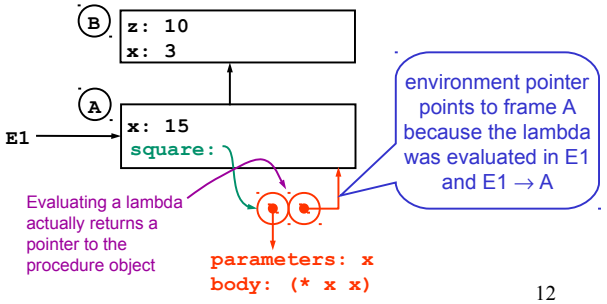
Double bubble: how to draw a procedure



Lambda-rule

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

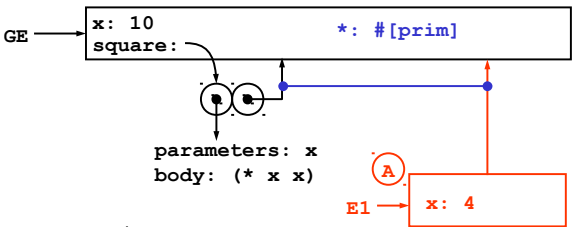
`(define square (lambda (x) (* x x))) |E1`



To apply a compound procedure P to arguments:

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

`(square 4)` | GE

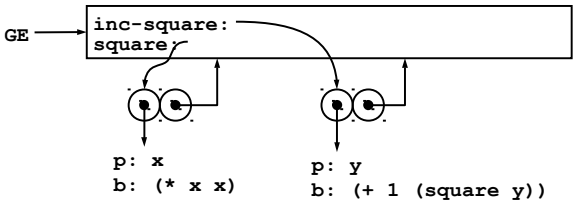


`square` | GE `==> #[proc]`

`(* x x)` | E1 `==> 16`

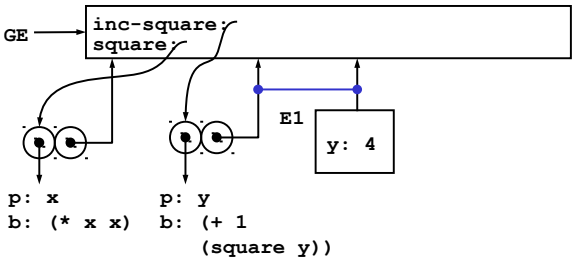
`*` | E1 `==> #[prim]` `x` | E1 `==> 4`

Example: inc-square




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


Example cont'd: `(inc-square 4)` | GE

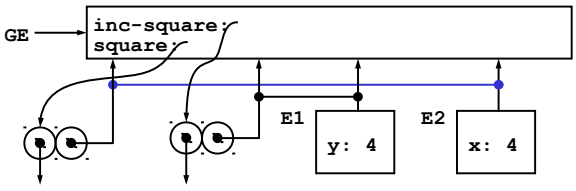


`inc-square` | GE `==> #[compound-proc ...]`

`(+ 1 (square y))` | E1

`+` | E1 `==> #[prim]` `(square y)` | E1

Example cont'd: `(square y)` | E1



`square` | E1 ==> `#[compound]` `y` | E1 ==> `4`

`(* x x)` | E2 ==> `16` `(+ 1 16)` ==> `17`

`*` | E2 ==> `#[prim]` `x` | E2 ==> `4`

Lessons from the `inc-square` example

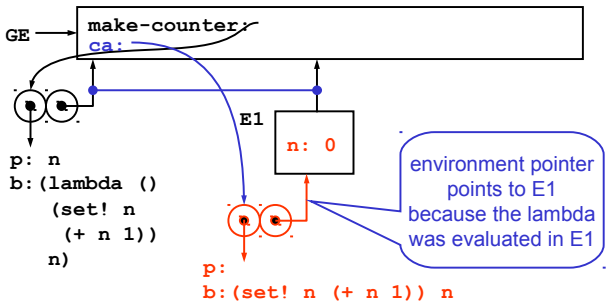
- EM doesn't show the complete state of the interpreter
  - missing the stack of pending operations
- The GE contains all standard bindings (`*`, `cons`, etc)
  - omitted from EM drawings
- Useful to link environment pointer of each frame to the procedure that created it

Example: make-counter

- Counter: something which counts up from a number

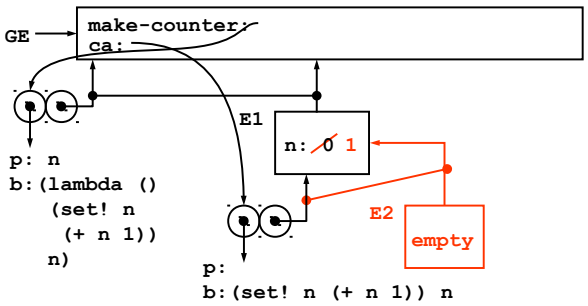
```
(define make-counter
  (lambda (n)
    (lambda () (set! n (+ n 1))
               n
    )))

(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
```


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



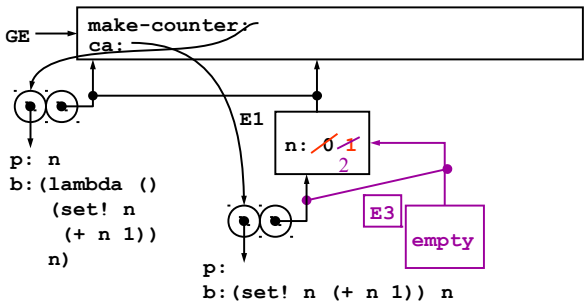
`(lambda () (set! n (+ n 1)) n)` | E1

(ca) | GE ==> 1



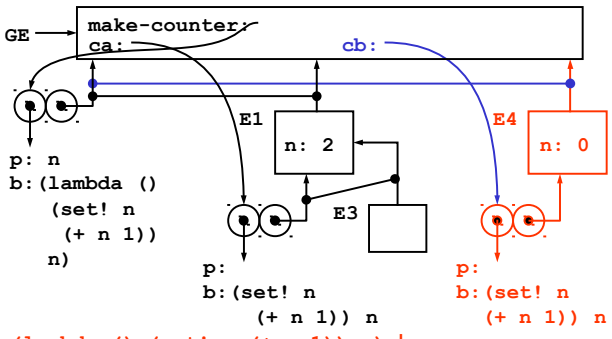
(set! n (+ n 1)) | E2 n | E2 ==> 1

(ca) | GE ==> 2



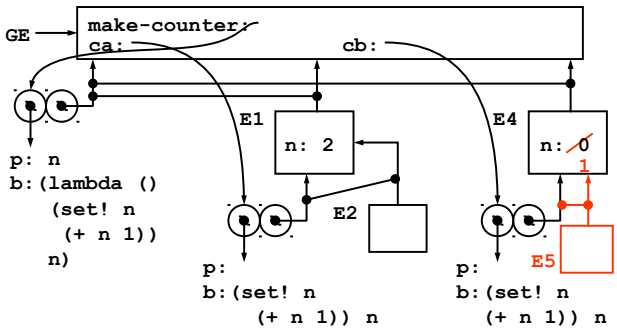
(set! n (+ n 1)) | E3 n | E3 ==> 2

(define cb (make-counter 0)) | GE

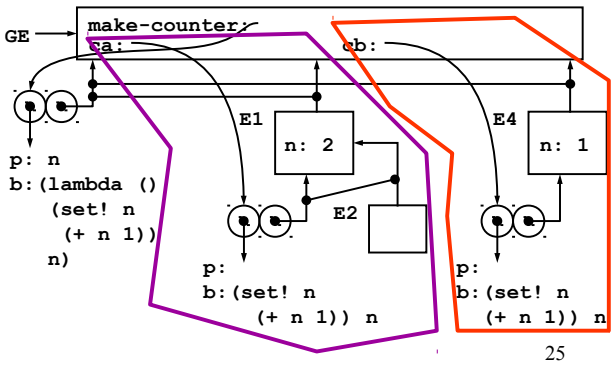


(lambda () (set! n (+ n 1)) n) | E4

(cb) | GE ==> 1



Capturing state in local frames & procedures



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