6.001 SICP
Environment Model

- A model for computation consistent with mutation
  - tells us where variable bindings live
  - tells us where bindings are changed
- A graphical model for how Scheme works
  - shows how lexical scoping (or block structure) is achieved
- A means to create and manipulate procedures with local state

Need for a New Model of Computation

- **Functional Programming** (up to now)
  - Every expression (almost) has a value
  - Procedures capture a *mapping* from values to values

    ```
    (define (square x) (* x x)) ; number -> number
    ```

  - **Substitution Model** – expansions (by way of procedure applications) and reductions of expressions

    ```
     (square 5)
    ==> (* 5 5)
    ==> 25
    ```

Need for a New Model of Computation

- **Functional Programming** (up to now)
  - Every expression (almost) has a value
  - Procedures capture a *mapping* from values to values

```
(define (square x) (* x x)) ; number -> number
```

  - **Substitution Model** – expansions (by way of procedure applications) and reductions of expressions

```
(square 5)
==> (* 5 5)
==> 25
```

- **Imperative Programming** (with mutation)
  - Expressions can "do" something – have side effects

```
(define x 10)
x ==> 10
(set! x 20)  – changes something…
x ==> 20     – different values, depends on WHEN evaluated!
```

Environment Model

What the environment model 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 analysis of procedures with local/mutable state:
    - Example: **make-counter**

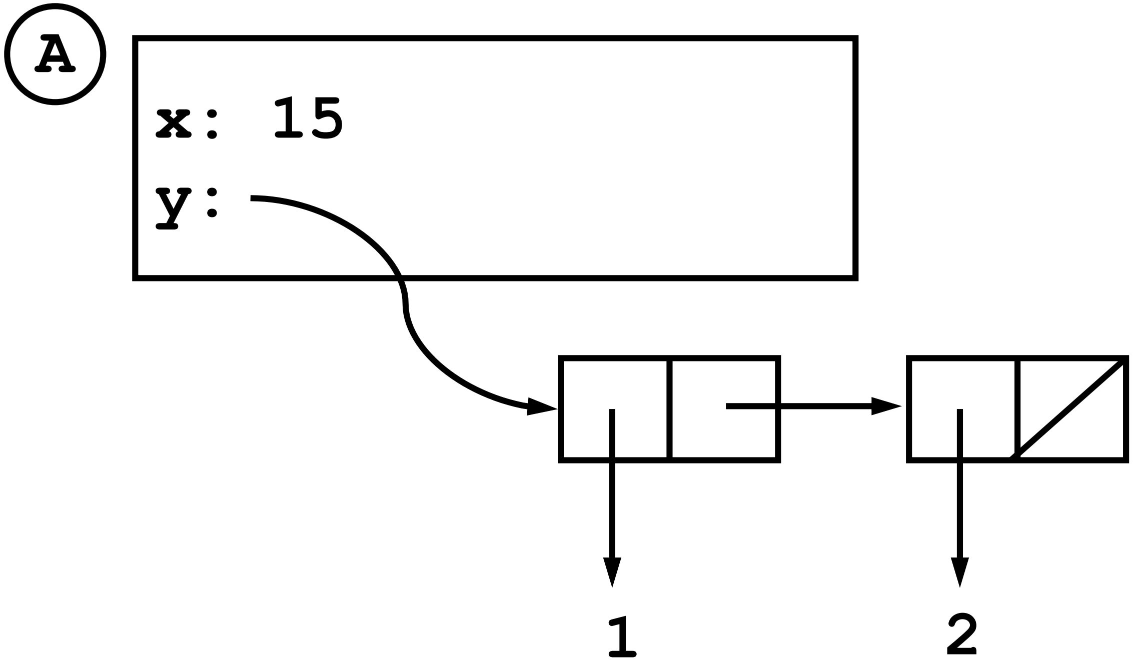
- Basis for implementing a scheme interpreter
    - for now: draw EM state with frames and pointers
    - later on: implement with code

Frame: a table of bindings

- Binding: a pairing of a name and a value

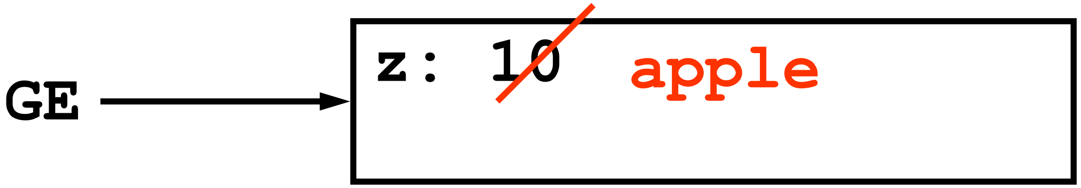
Example: **`x`** is bound to **`15`** in frame A
`y` is bound to **`(1 2)`** in frame A

the value of the variable **`x`** in frame A is 15



Environment

- Generally, an **environment** is a **sequence** of frames
  - Simplest example: the global environment (GE)
- All evaluation occurs with respect to an environment
  - Notation: $\texttt{<exp>}|_{\texttt{<env>}}$



GE

z: ~~10~~ apple

`(define z 10)` $|_{GE}$ **`==> unspecified`** (side effect!)

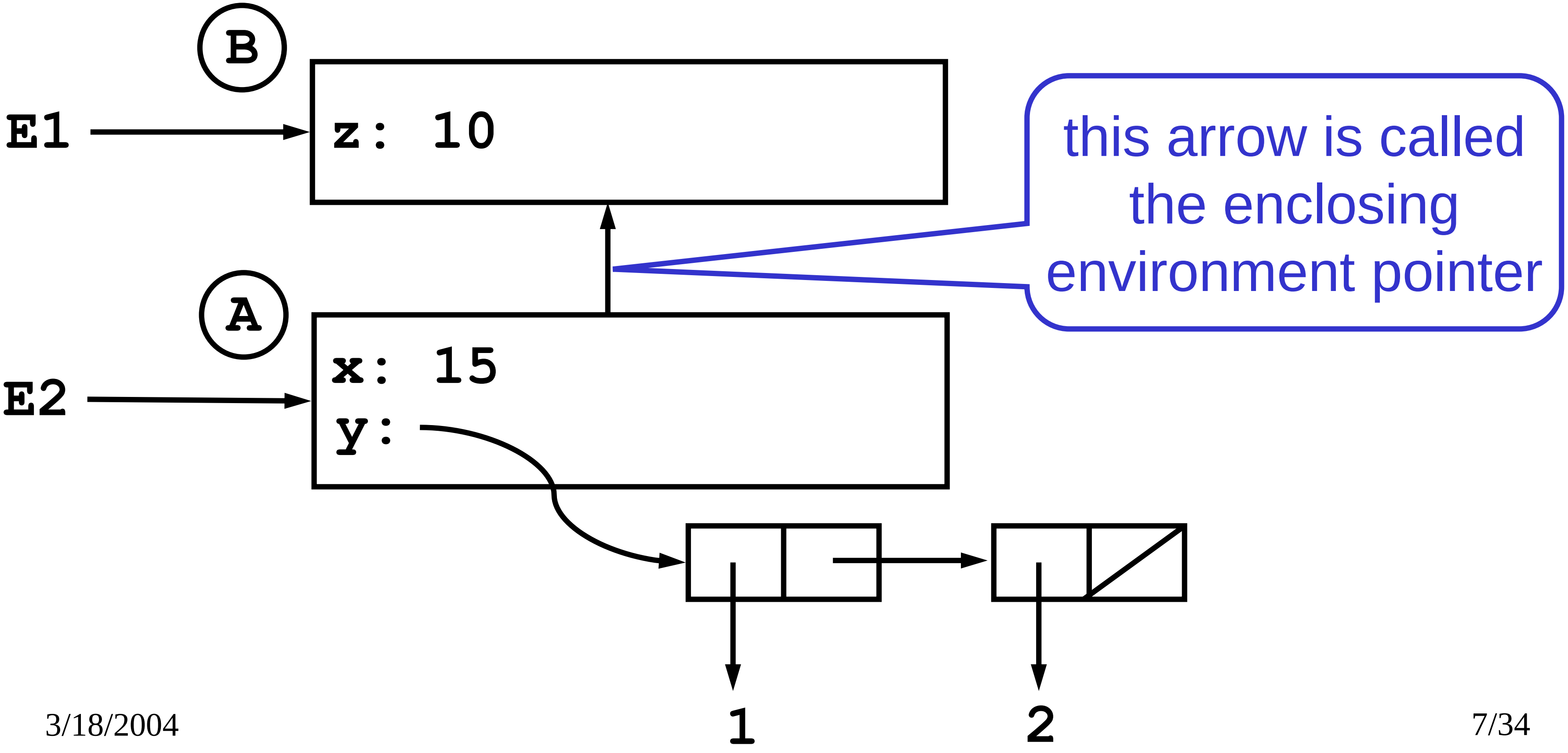
`z` $|_{GE}$ **`==> 10`**

`(set! z 'apple)` $|_{GE}$ **`==> unspecified`** (side effect!)

`z` $|_{GE}$ **`==> apple`**

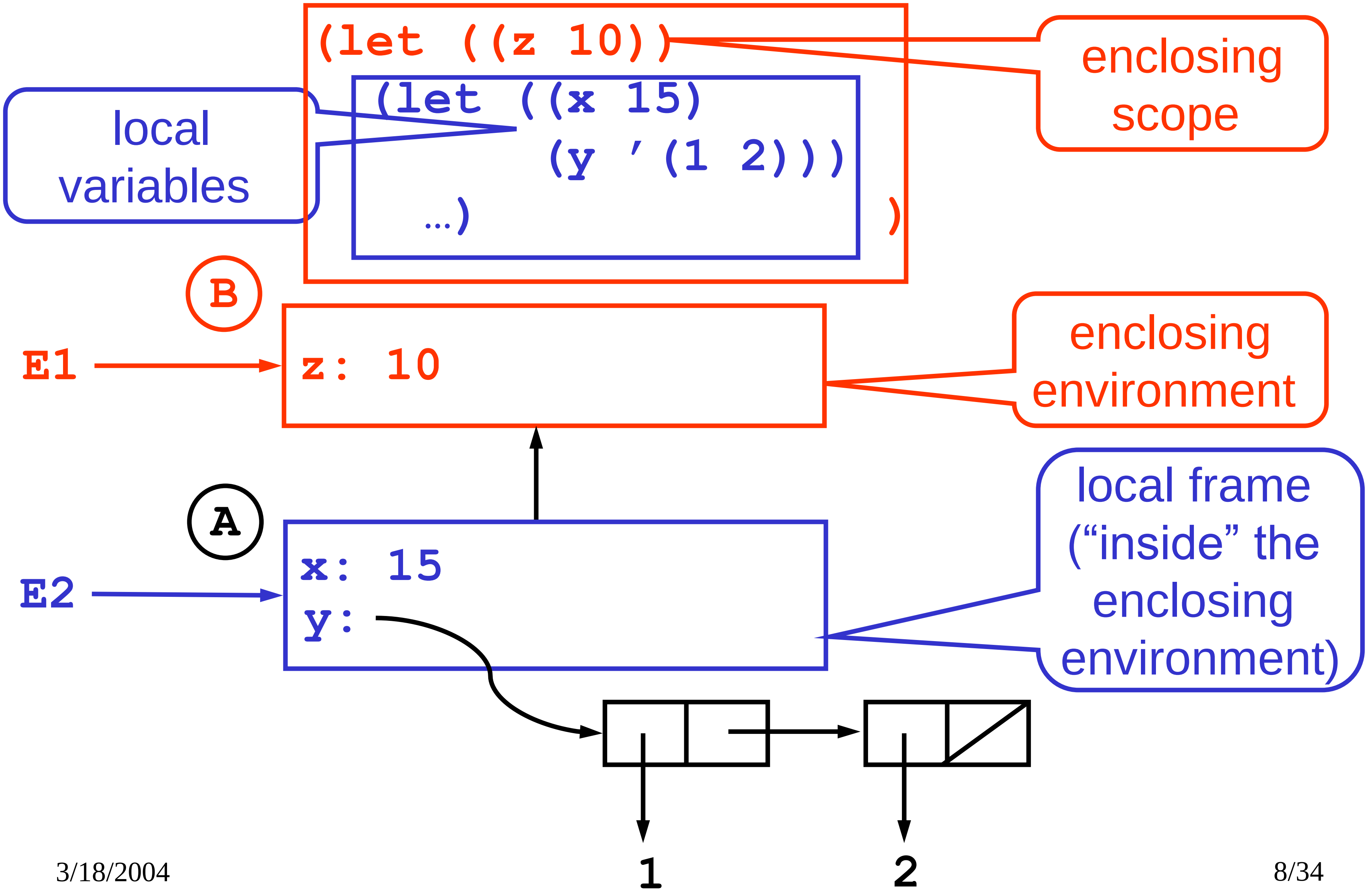
Environment as a sequence of frames

- Environment E1 consists of frame B only
- Environment E2 consists of frames A and B
  - A frame may be shared by multiple environments



this arrow is called the enclosing environment pointer

Environments & Lexical Scope (Block Structure)



```
(let ((z 10))
  (let ((x 15)
        (y '(1 2)))
    ...)
)
```

local variables

enclosing scope

B

E1

z: 10

enclosing environment

A

E2

x: 15

y:

local frame ("inside" the enclosing environment)

1

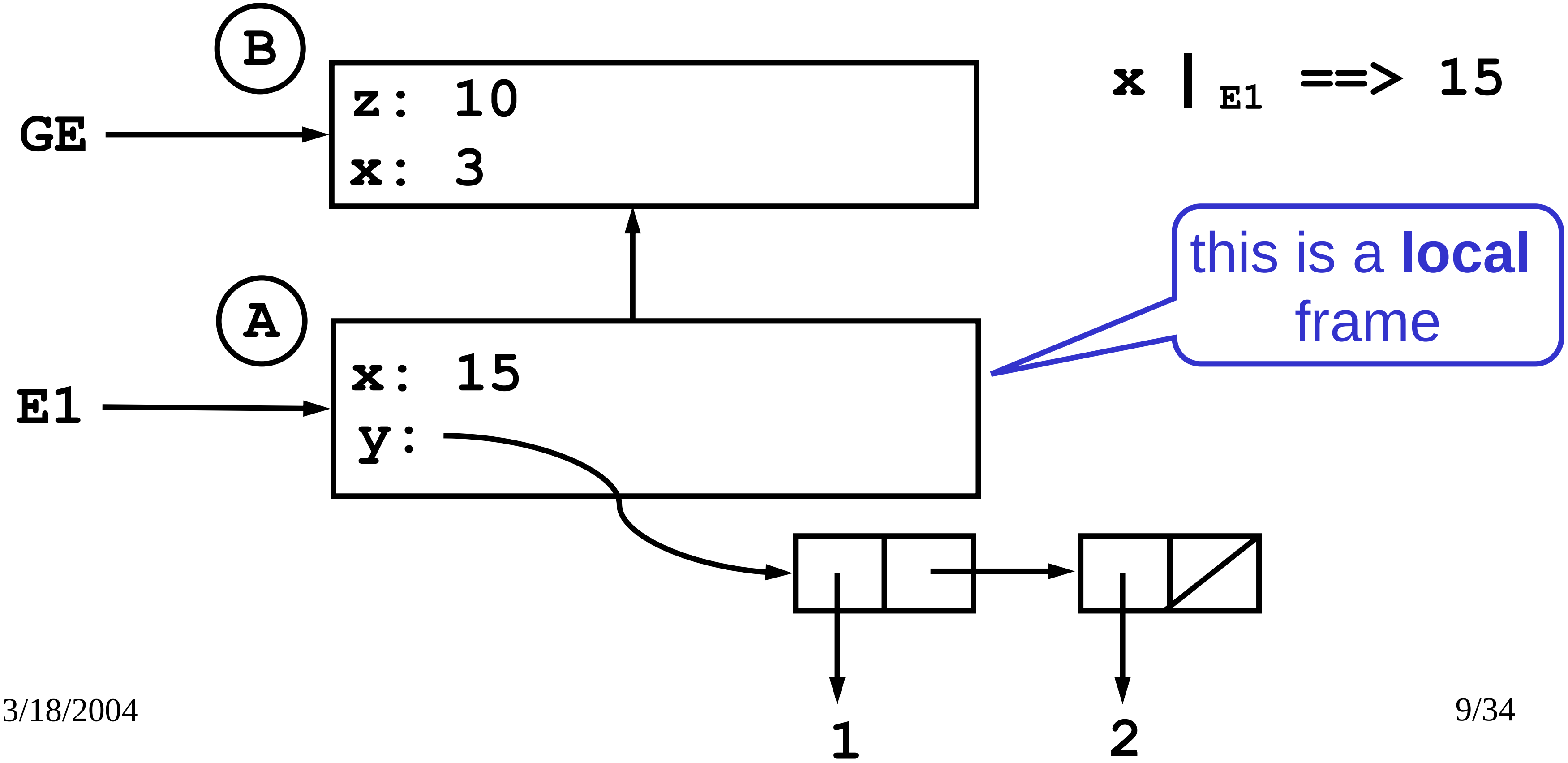
2

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 $x \,|_{GE}$ ==> 3

- In E1, the binding of x in frame A shadows the binding of x in B



$x \,|_{E1}$ ==> 15

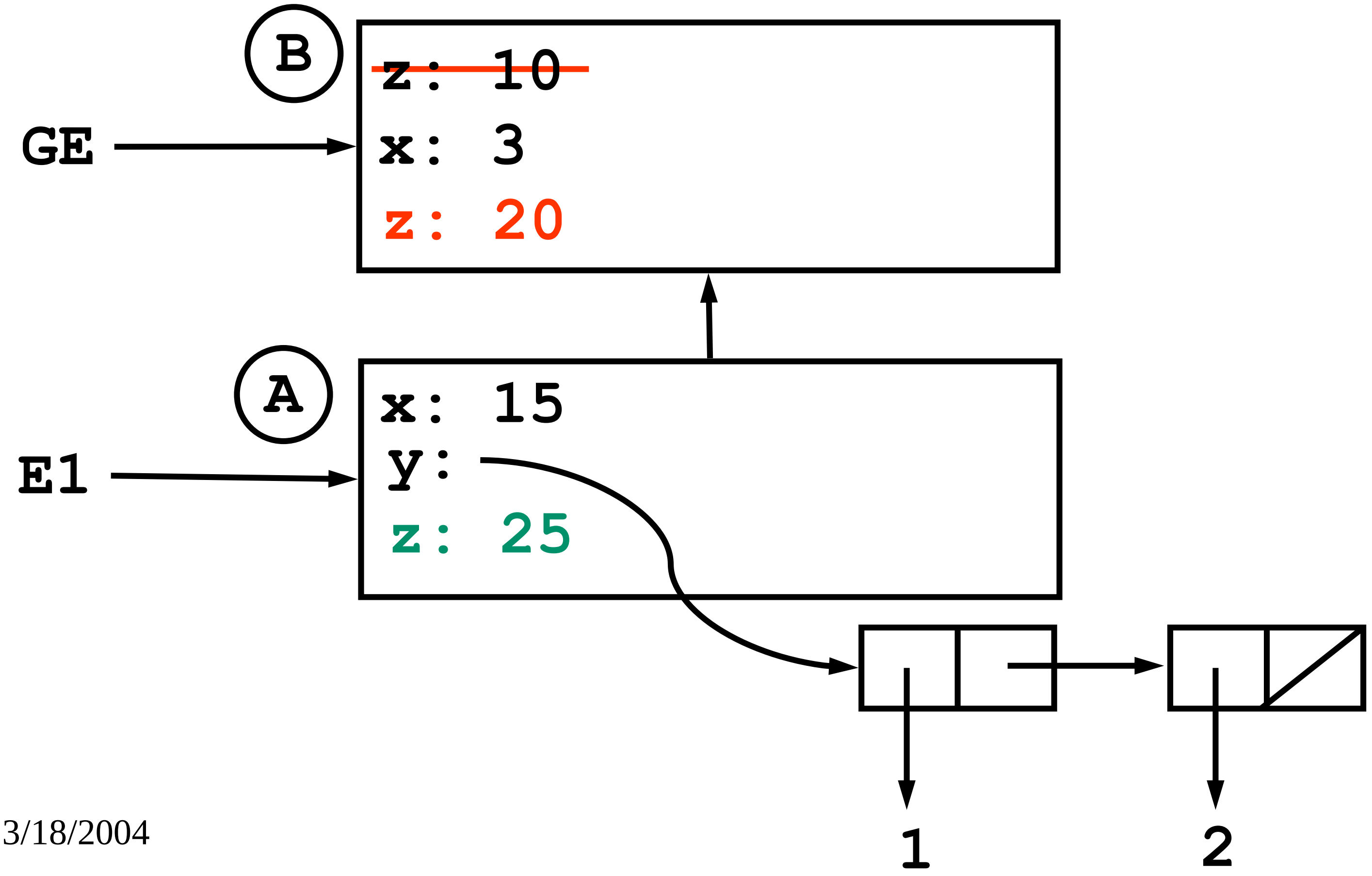
this is a **local** frame

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



Set!-rule

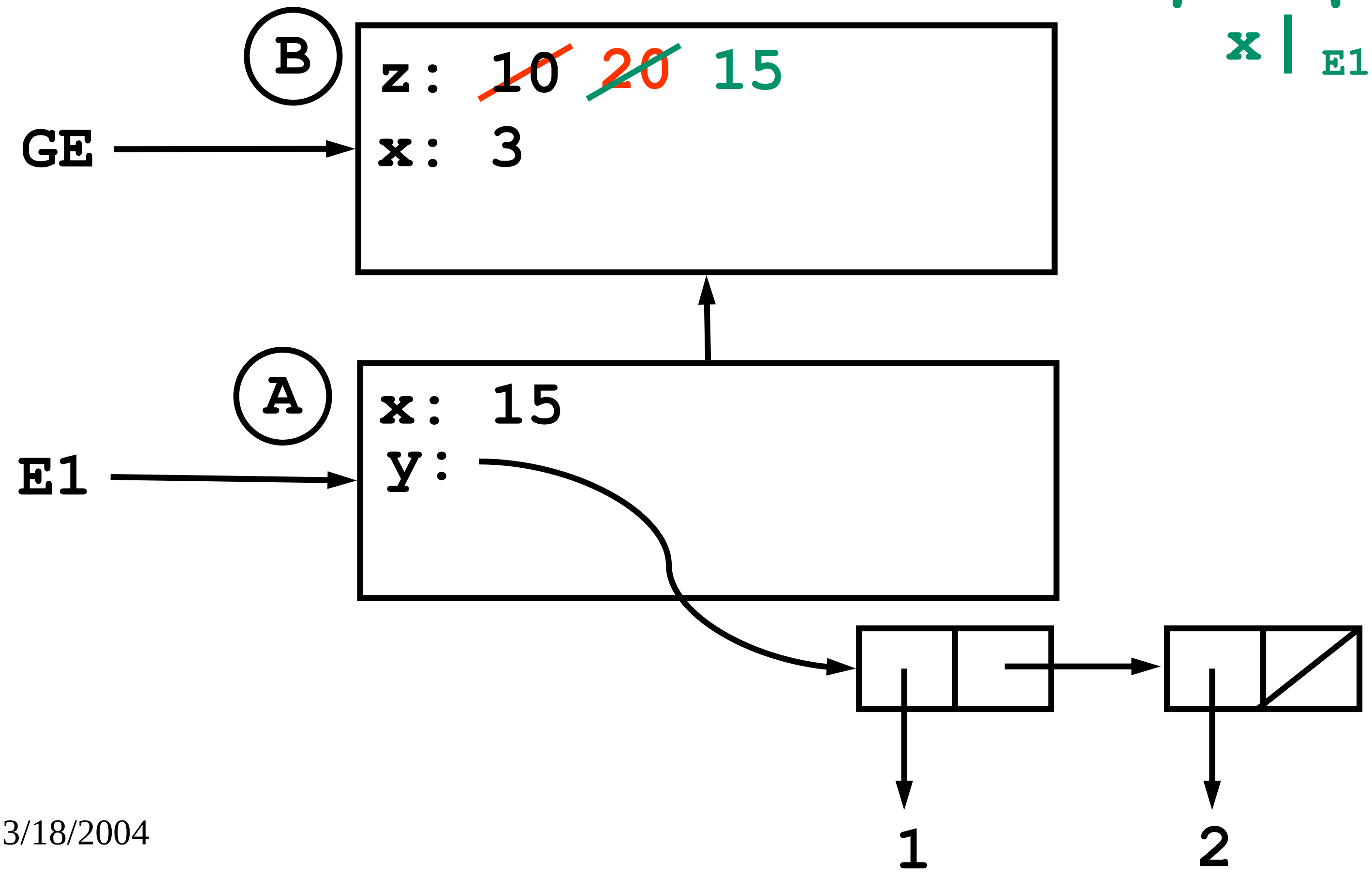
- A set! of variable X evaluated in environment E changes the binding of X in the first frame of E where X is bound

`(set! z 20)` $|_{GE}$

`(set! z x)` $|_{E1}$

$\underbrace{x}_{x|_{E1} \Rightarrow 15}$

`x` $|_{E1}$ => 15



Your turn: evaluate the following in order

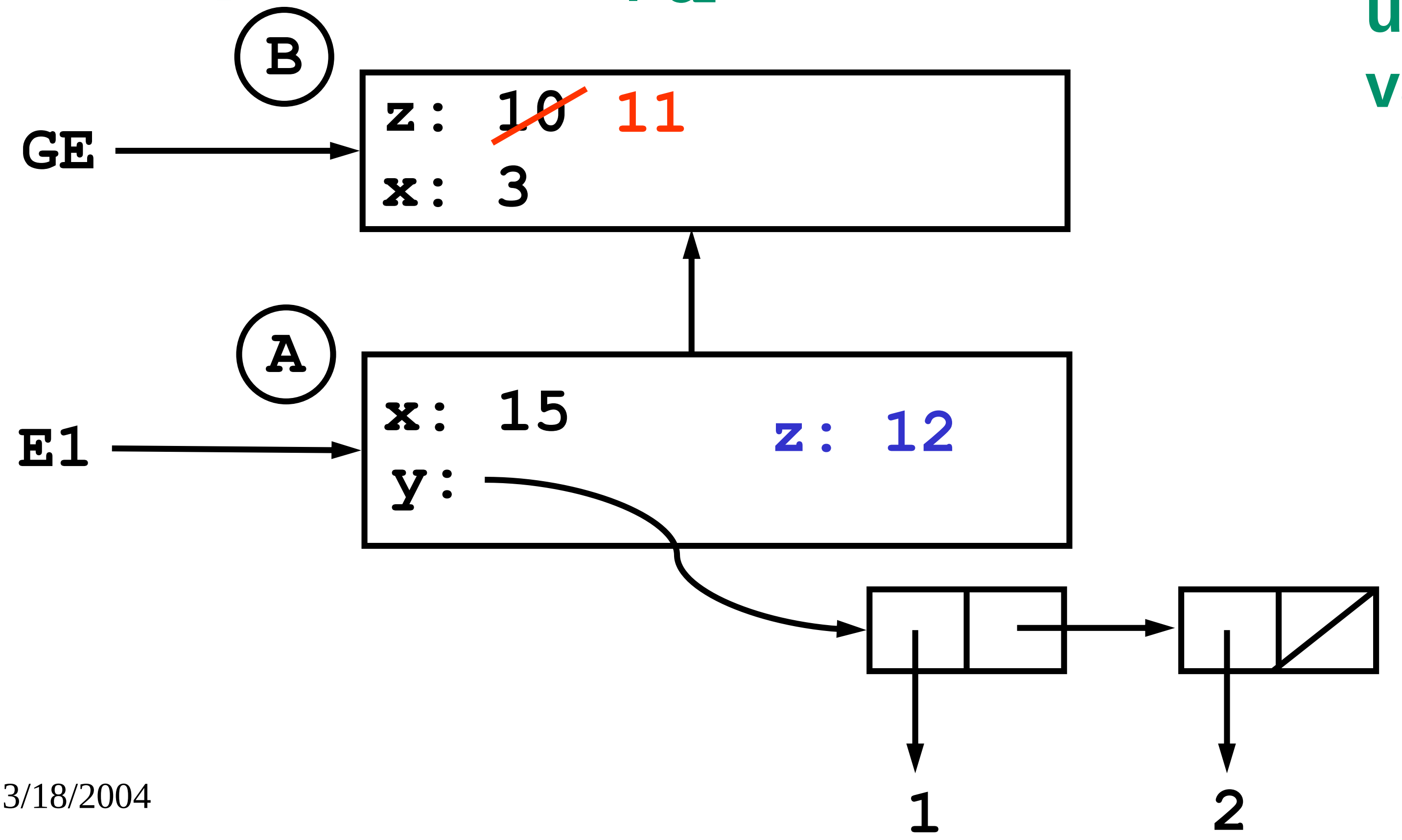
`(+ z 1)` $|_{E1}$ ==> 11

`(set! z (+ z 1))` $|_{E1}$ (modify EM)

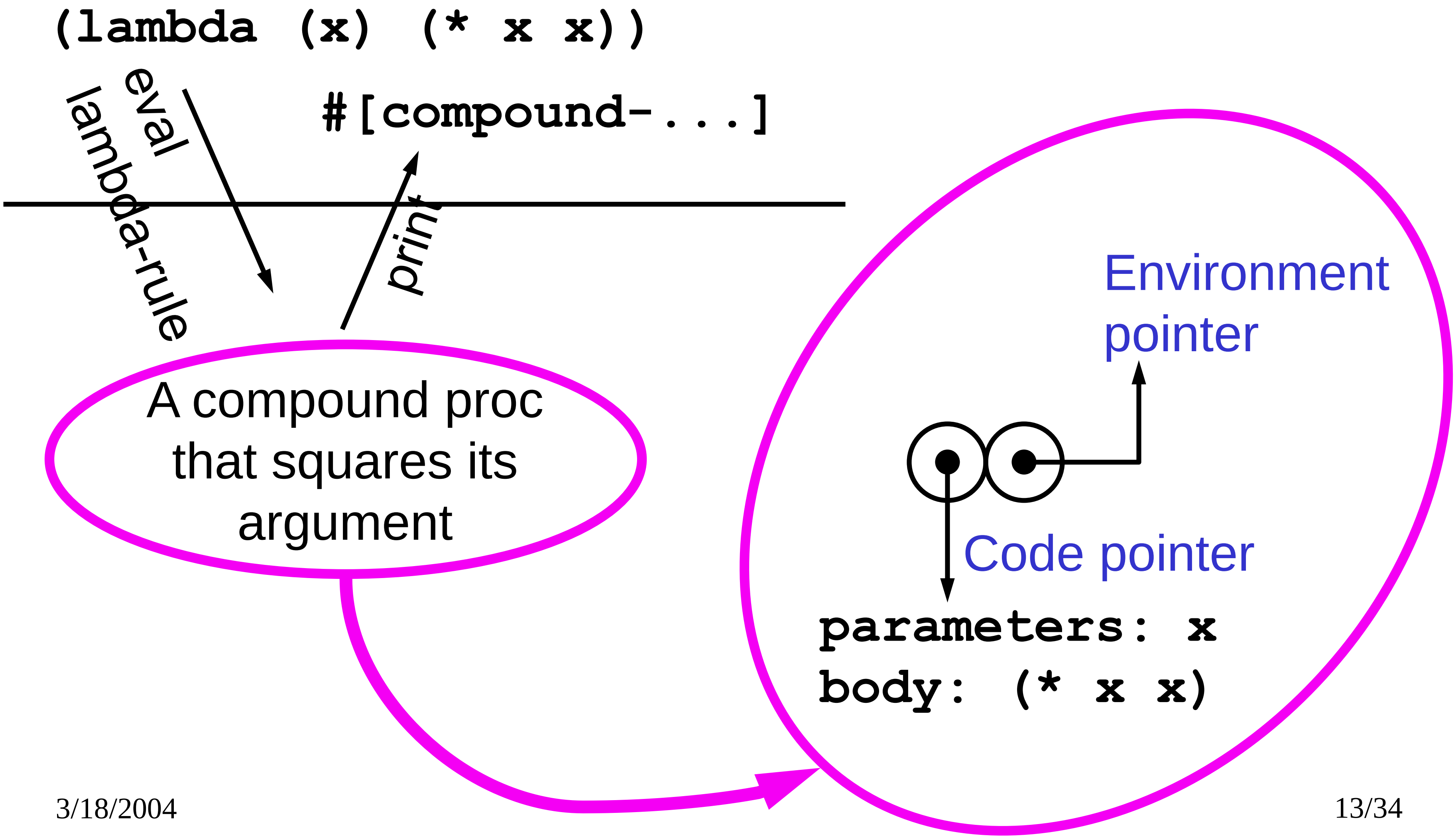
`(define z (+ z 1))` $|_{E1}$ (modify EM)

`(set! y (+ z 1))` $|_{GE}$ (modify EM)

Error: unbound variable: y



Double bubble: how to draw a procedure



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

`#[compound-...]`

A compound proc that squares its argument

Environment pointer

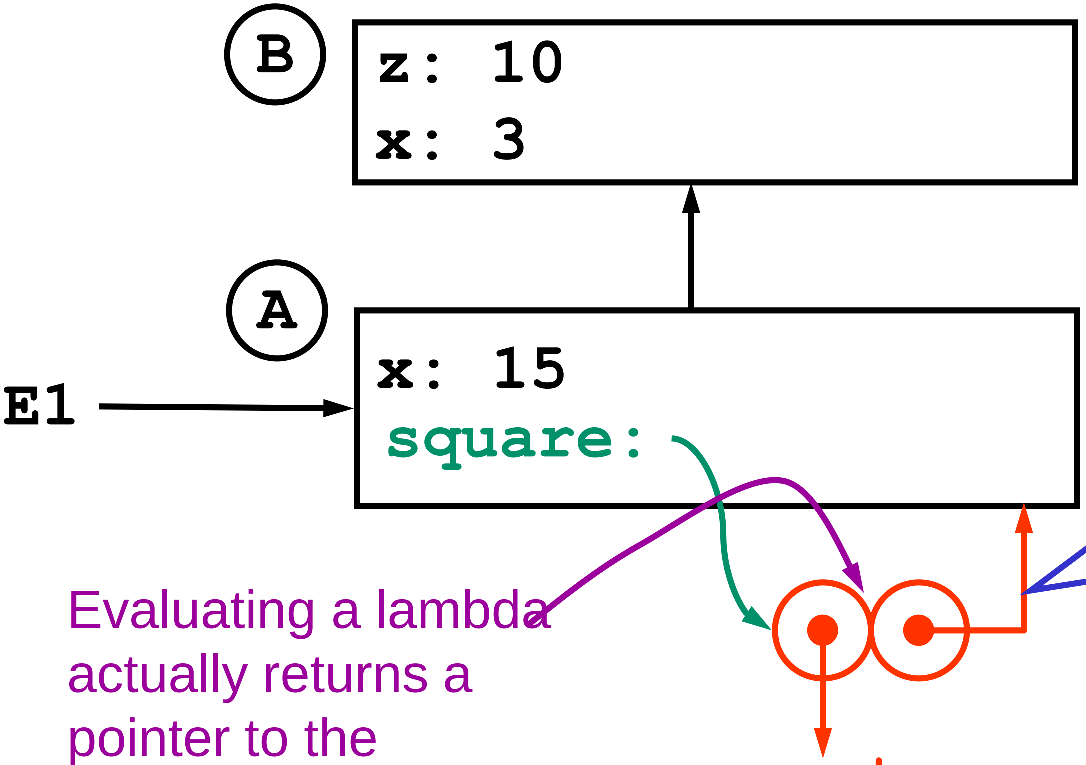
Code pointer

```
parameters: x
body: (* x x)
```

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
```



parameters: x
body: (* x x)

environment pointer points to frame A because the lambda was evaluated in E1

Evaluating a lambda actually returns a pointer to the procedure object

To apply a compound procedure P to arguments:

1. Create a new frame A

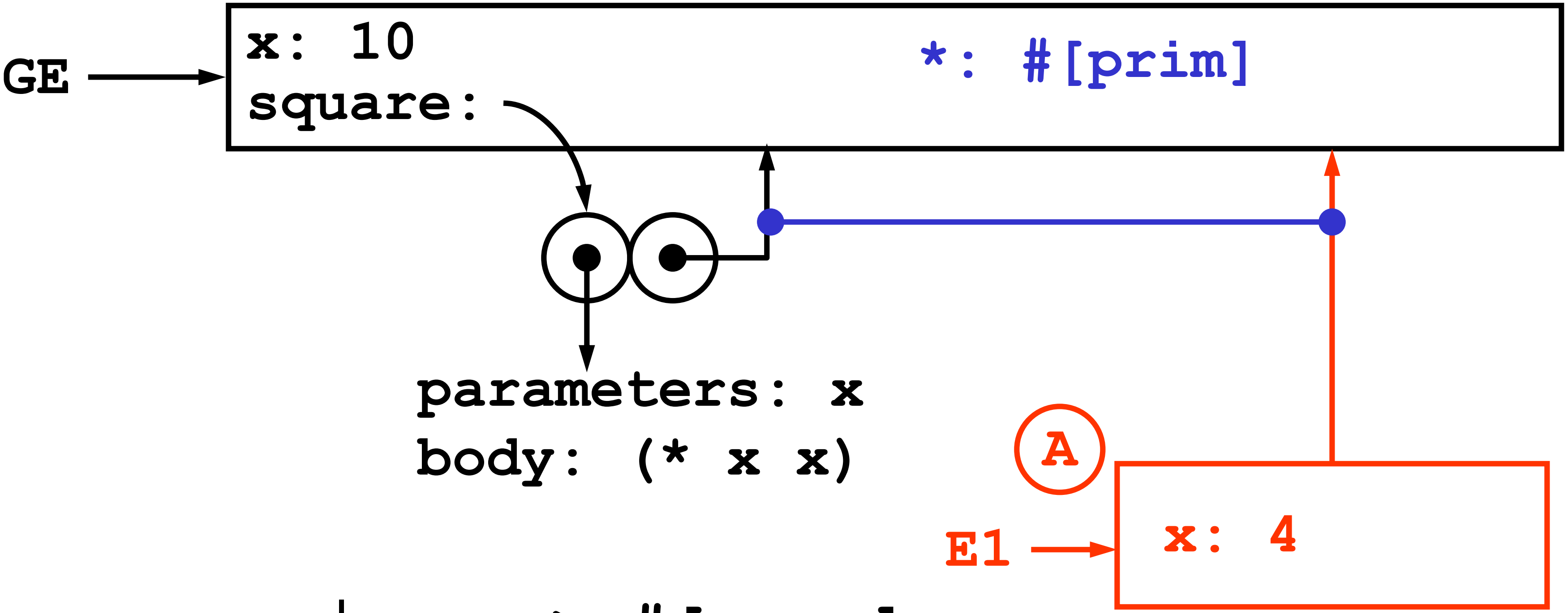
2. Make A into an environment E:
   A's enclosing environment pointer goes to the same frame as the environment pointer of P

3. In A, bind the parameters of P to the argument values

4. Evaluate the body of P with E as the current environment

You must memorize these four steps

(square 4) $|_{GE}$



GE

x: 10

square:

*: #[prim]

parameters: x

body: (* x x)

A

E1

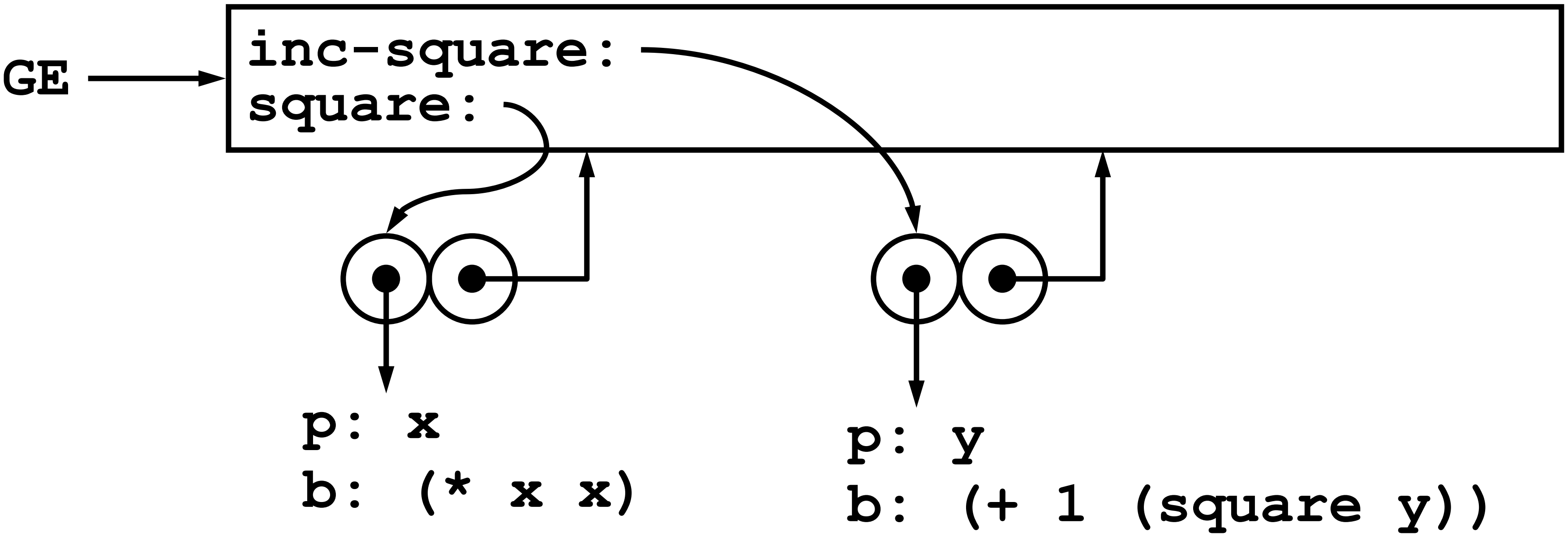
x: 4

square $|_{GE}$ ==> #[proc]

(* x x) $|_{E1}$ ==> 16

* $|_{E1}$ ==> #[prim]   x $|_{E1}$ ==> 4

Example: inc-square



GE

inc-square:

square:

p: x

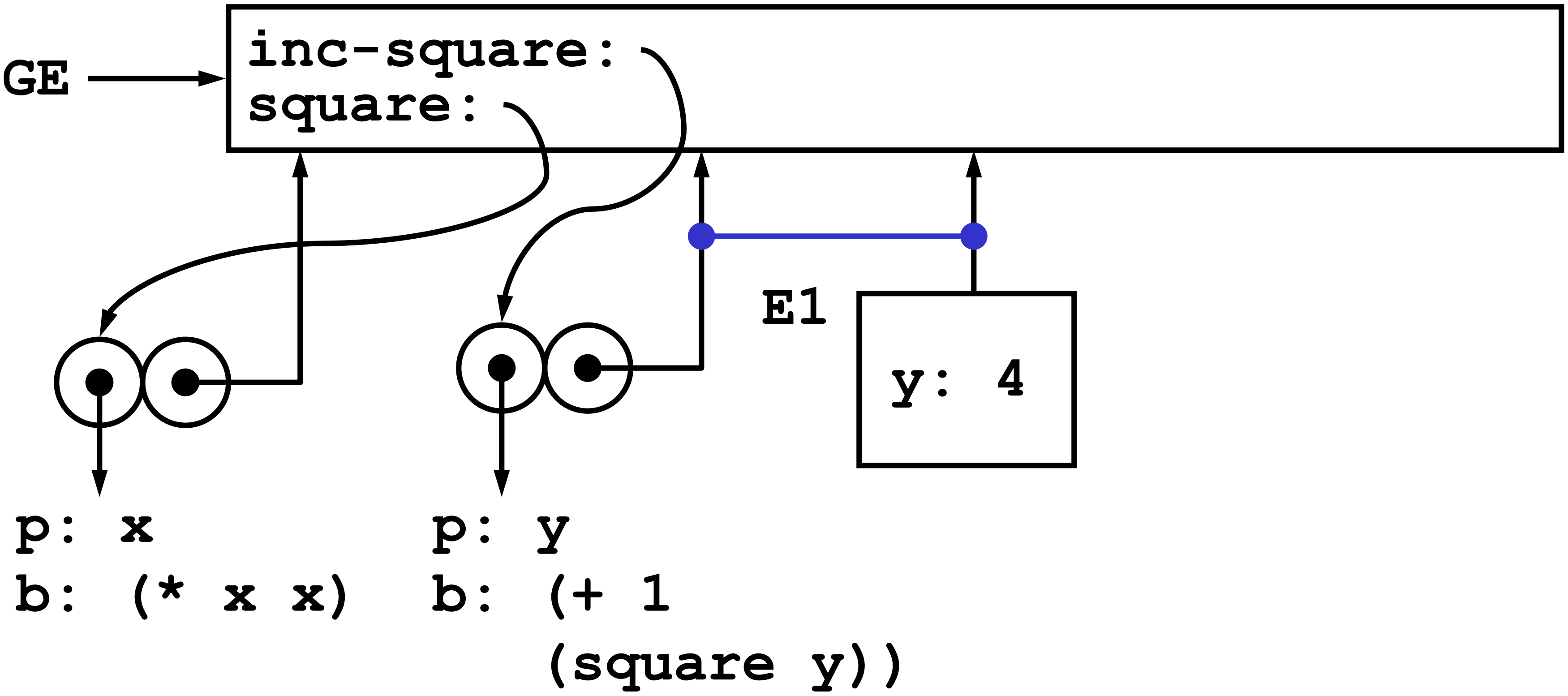
b: (* x x)

p: y

b: (+ 1 (square y))

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

Example cont'd: `(inc-square 4)` $|_{GE}$



GE

inc-square:

square:

E1

y: 4

```
p: x
b: (* x x)
```

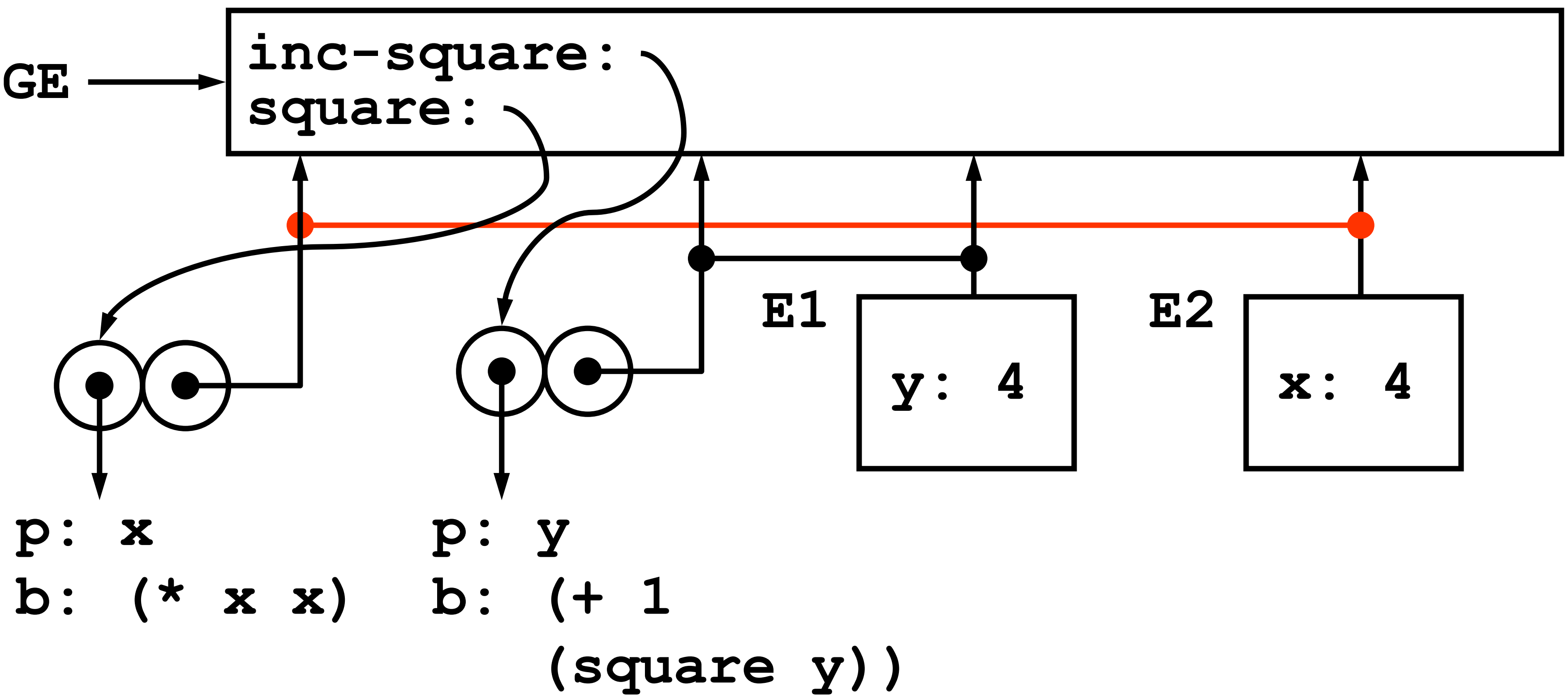
```
p: y
b: (+ 1
     (square y))
```

`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)|$_{E1}$ ==> 17

* |$_{E2}$ ==> #[prim]   x |$_{E2}$ ==> 4

Lessons from the `inc-square` example

- Environment model (EM) doesn't show the complete state of the interpreter
  - missing the stack of pending operations

- The GE contains all standard bindings (**`*`**, **`cons`**, etc)
  - usually omitted from EM drawings

- Useful to link environment pointer of each frame to the procedure that created it
  - reminds us where that frame came from, and what next steps are… binding args and then evaluating proc body

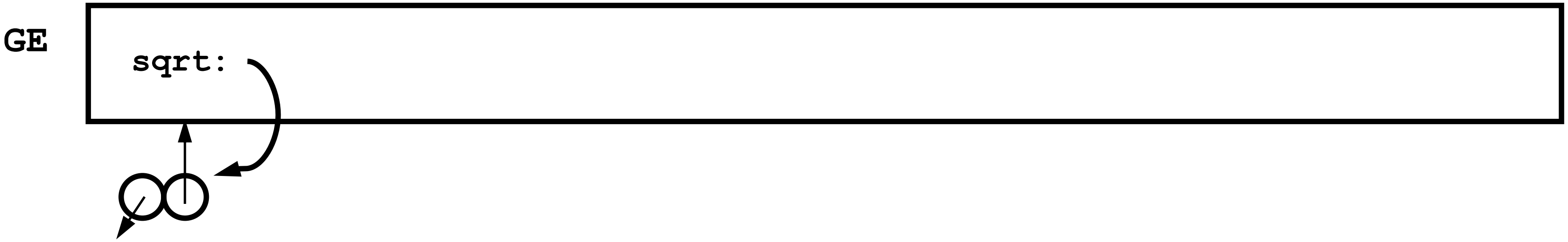
Lexical Scoping and the EM – Key Ideas

- Local environments
  - “Inside” other environments in code text
  - Local frames pointing to enclosing environment
- Procedures remember their environments!
  - What matters is the surrounding environment at procedure creation time,
    - which will be the surrounding lexical environment,
  - NOT the environment that the procedure finally gets applied in
  - Benefit: if you can view/read the code, then you always know where the variable values are to be found

Lexical Scoping Example – `sqrt`

```
(define sqrt
 (lambda (x)
  (define good-enough?
     (lambda (guess)
       (< (abs (- (square guess) x)) 0.001)))
  (define improve
     (lambda (guess)
       (average guess (/ x guess))))
  (define sqrt-iter
     (lambda (guess)
       (if (good-enough? guess)
           guess
           (sqrt-iter (improve guess)))))
  (sqrt-iter 1)))
```

`sqrt` Example



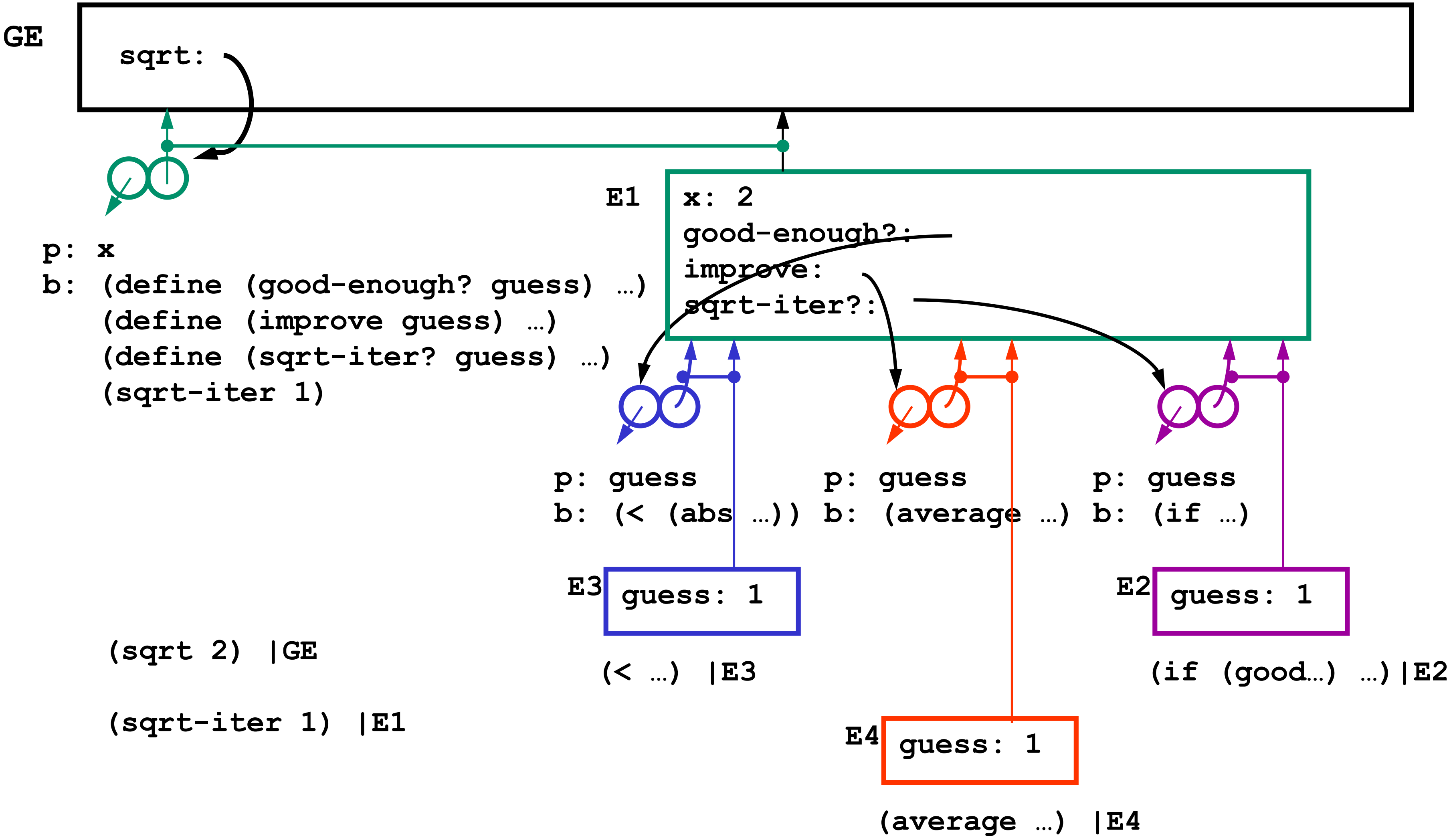
GE

sqrt:

```
p: x
b: (define (good-enough? guess) …)
   (define (improve guess) …)
   (define (sqrt-iter? guess) …)
   (sqrt-iter 1)
```

```
(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)) |GE
```

sqrt Example



GE

sqrt:

p: x

b: (define (good-enough? guess) …)
   (define (improve guess) …)
   (define (sqrt-iter? guess) …)
   (sqrt-iter 1)

E1

x: 2

good-enough?:

improve:

sqrt-iter?:

p: guess

b: (< (abs …))

p: guess

b: (average …)

p: guess

b: (if …)

E3

guess: 1

E2

guess: 1

(sqrt 2) |GE

(sqrt-iter 1) |E1

(< …) |E3

(if (good…) …)|E2

E4

guess: 1

(average …) |E4

Environment Model

- A model for computation consistent with mutation
  - tells us where variable bindings live
  - tells us where bindings are changed
- A graphical model for how Scheme works
  - shows how lexical scoping (or block structure) is achieved
- A means to create and manipulate local state

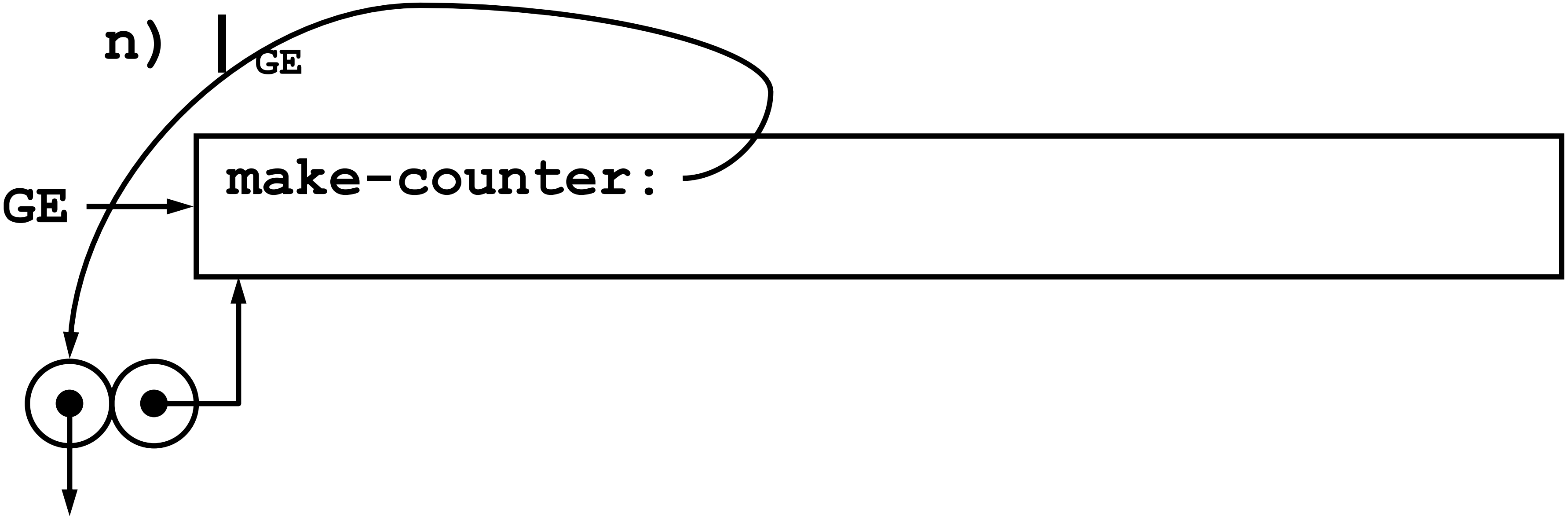
Example: make-counter

- 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
```

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



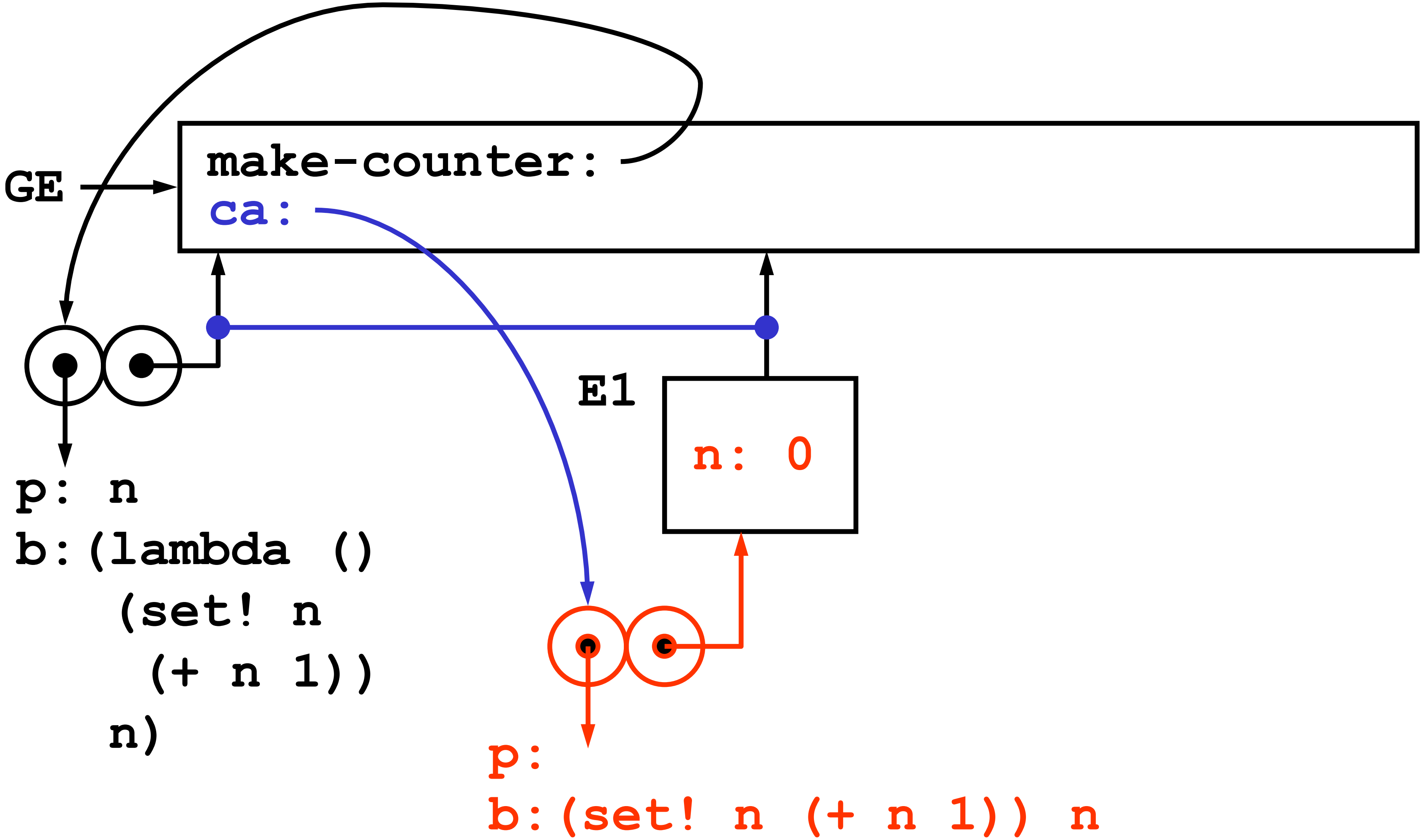
GE

GE

make-counter:

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

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



GE

make-counter:

ca:

E1

n: 0

p: n

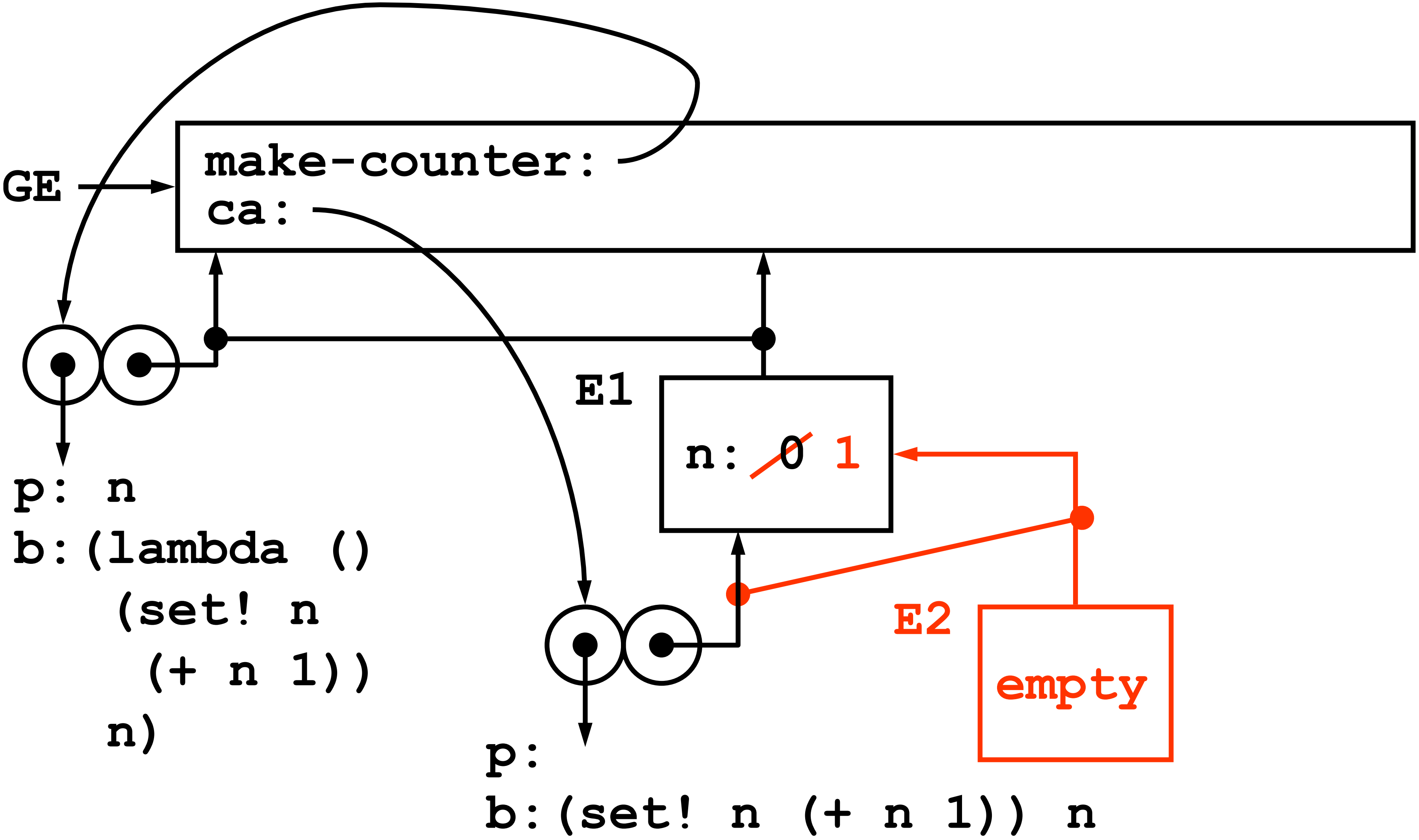
```
b:(lambda ()
    (set! n
     (+ n 1))
    n)
```

p:

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

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

(ca) $|_{GE}$ ==> 1



GE

make-counter:

ca:

E1

n: ~~0~~ 1

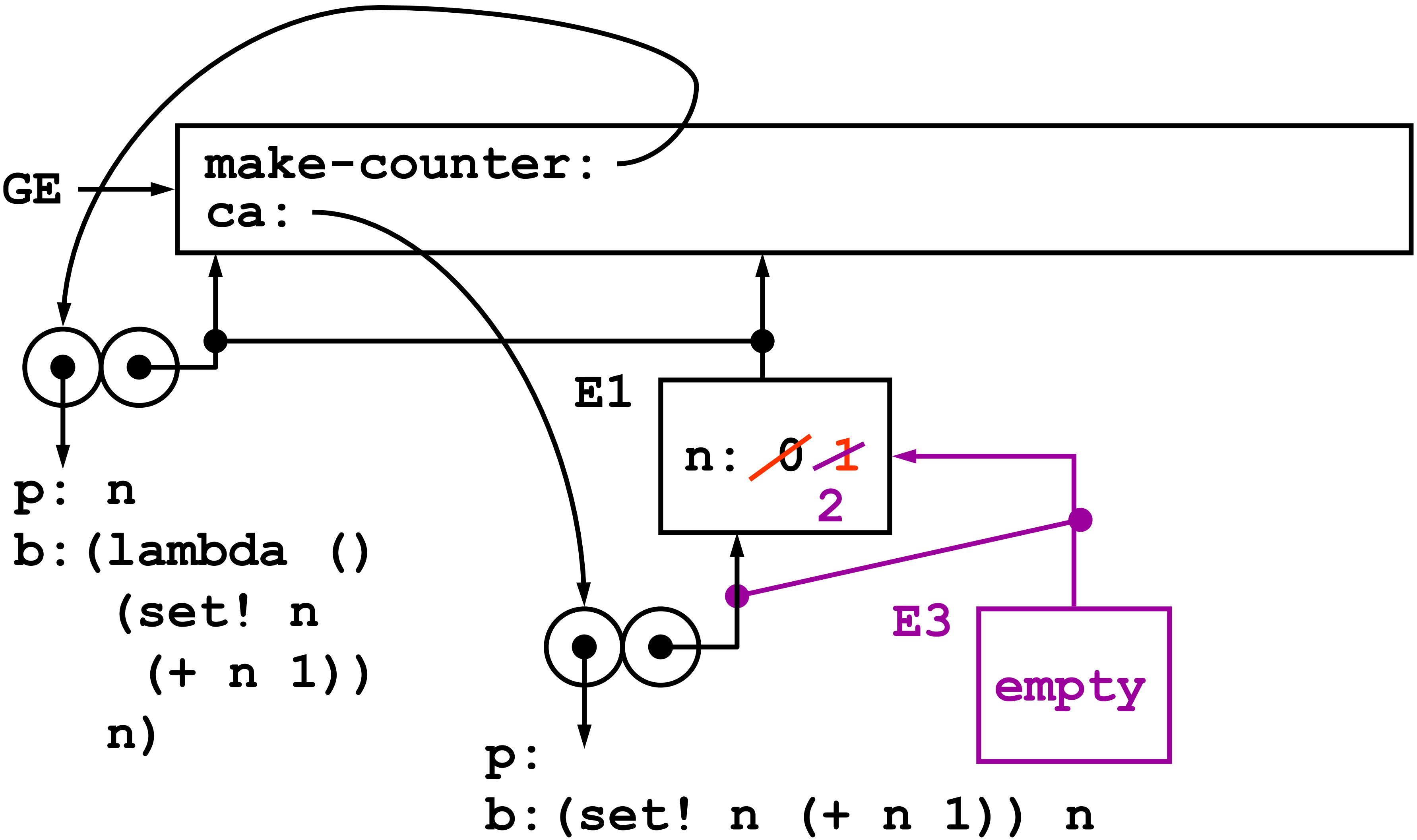
E2

empty

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

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

(set! n (+ n 1)) $|_{E2}$ n $|_{E2}$ ==> 1 (ca) $|_{GE}$ ==> 2



GE

make-counter:

ca:

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

E1

n: ~~0~~ ~~1~~ 2

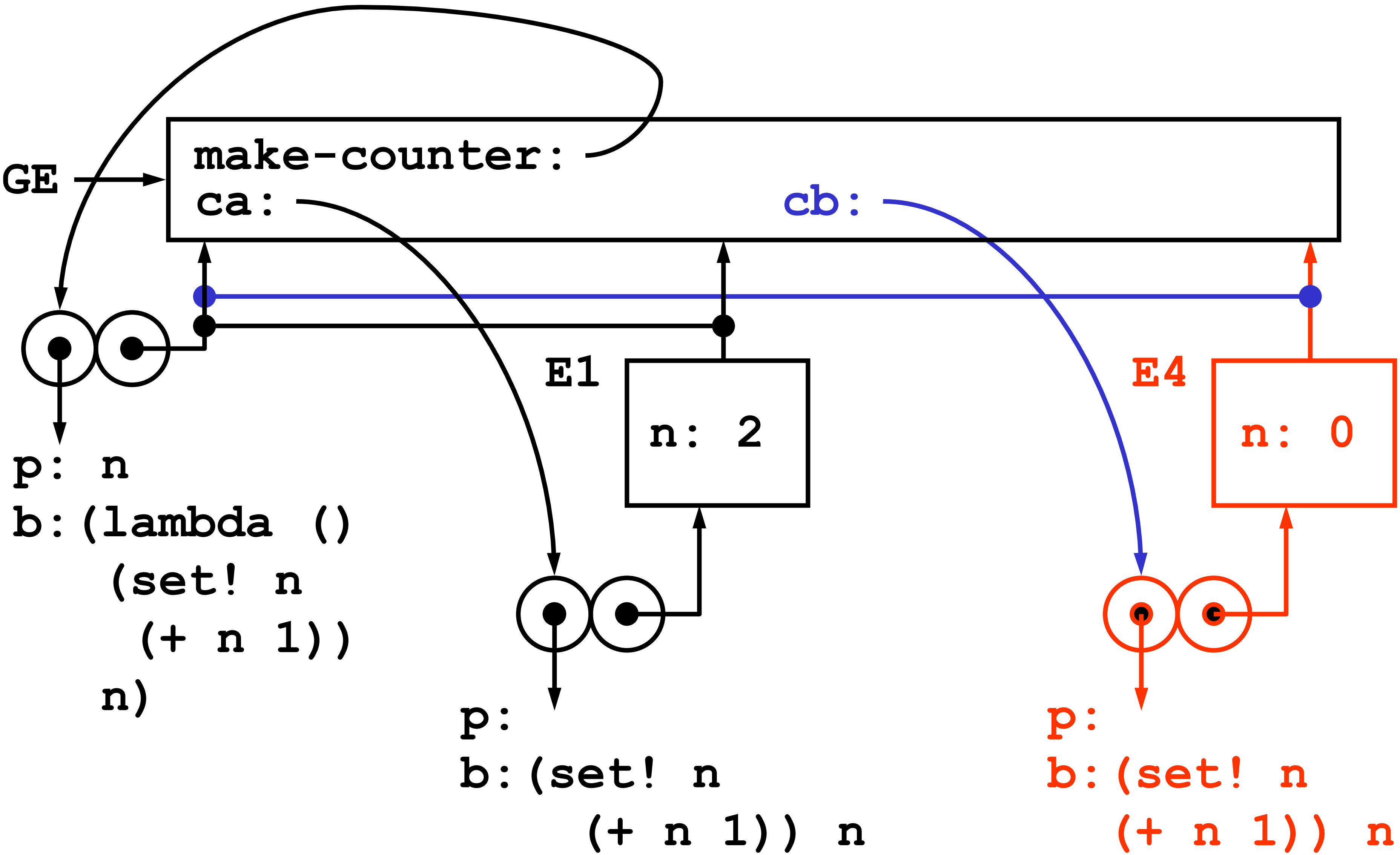
E3

empty

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

(set! n (+ n 1)) $|_{E3}$ n $|_{E3}$ ==> 2

(define cb (make-counter 0)) |$_{GE}$



GE

make-counter:

ca:

cb:

E1

n: 2

E4

n: 0

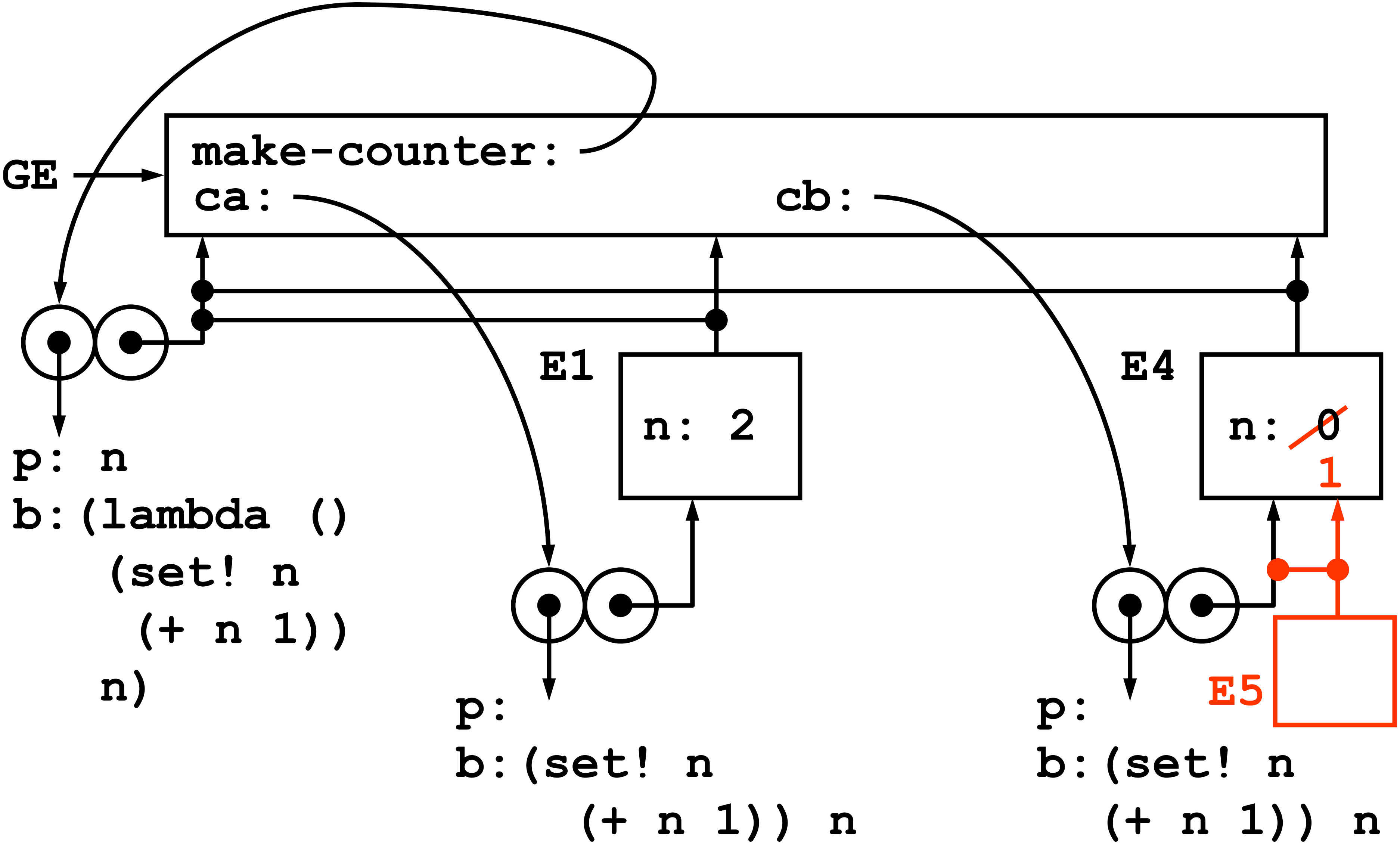
```
p: n
b:(lambda ()
    (set! n
     (+ n 1))
   n)
```

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

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

(lambda () (set! n (+ n 1)) n) |$_{E4}$

(cb) | GE ==> 1



GE

make-counter:

ca:

cb:

p: n

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

E1

n: 2

p:

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

E4

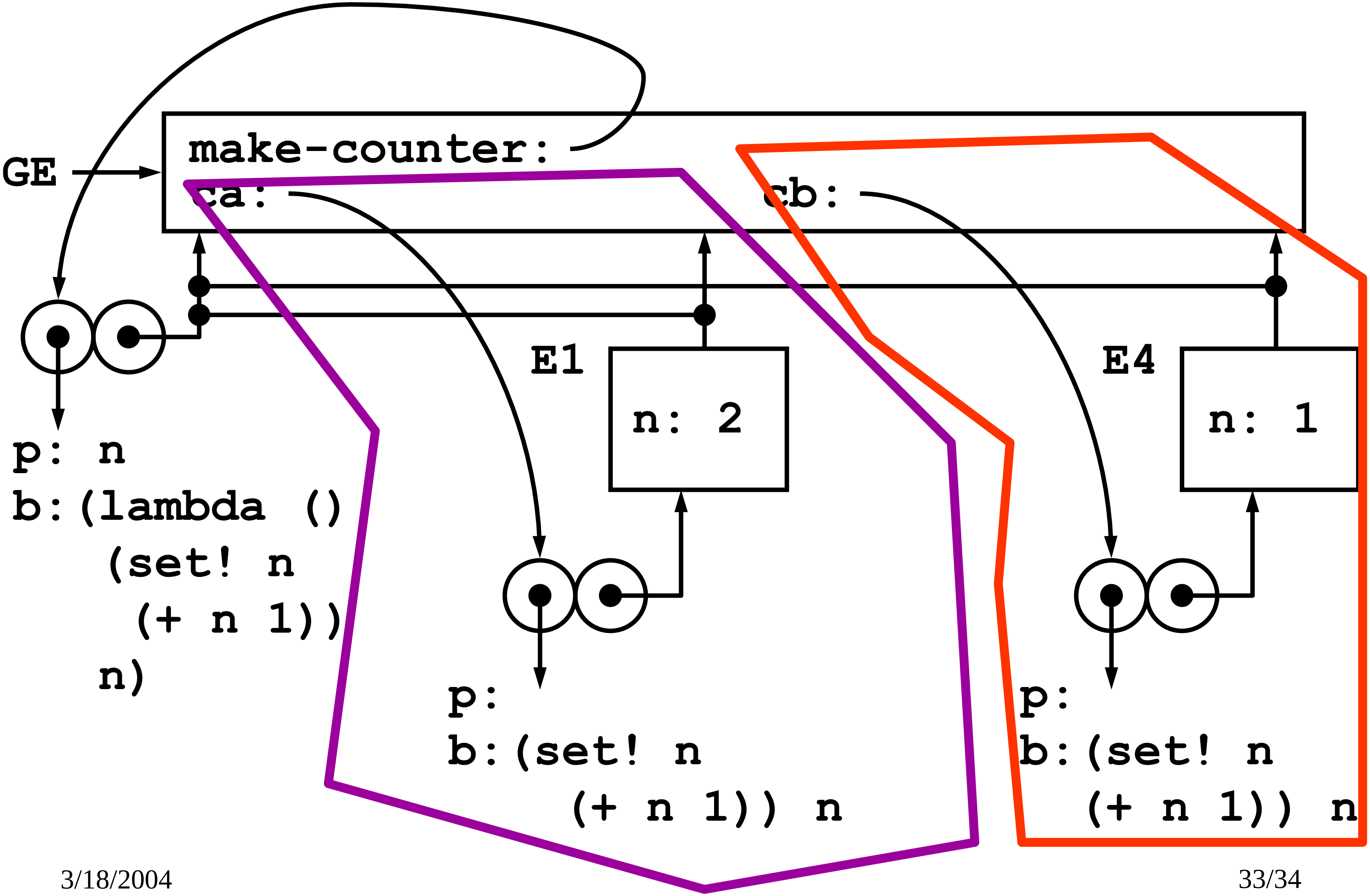
n: ~~0~~ 1

E5

p:

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

Capturing state in local frames & procedures



GE

make-counter:

ca:

cb:

E1

n: 2

E4

n: 1

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

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

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

Lessons Learned

- Environment diagrams get complicated very quickly
  - graphical tool to explain and reason using the environment model
- Environment Model:
  - implements block structure (lexical scoping)
  - shows where variables (bindings) are located
  - shows which values change as a result of mutation
- Implement objects with local state
  - a lambda captures the frame that was active when the lambda was evaluated
  - information hiding – expressions outside the environment do not have access to that local state
  - with environment model, see where local state changes