## define

- Define: names a variable and bin

```
==> (define pi 3.14159)
==> (define radius 10)
```

```
;; Calculate x-squared
(define (square x) (* x x))
> (square 21)
441
> (square (+ 2 5))
49
> (square (square 3))
81
```

```
(lambda (<formal parameters>) (<body>))
```

e: a lambda expression that creates an unnamed
ıre that squares its argument:

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

```
(if <predicate> <consequent> <alternative>)
```

ample Absolute value procedure

```
(define (abs x)
  (if (< x 0) (- x) x))
```

```
(cond (<p1> <e1>)
      (<p2> <e2>)
        ...
      (<pn> <en>))
```

```
(cond
  [<p1> <e1>]
  [<p2> <e2>]
    ...
  [<pn> <en>])
```

of expressions `(<p> <e>)` is a clause

- Recursive formulation for $x^n$
  - $x^0 = 1, x > 0$
  - $x^n = x * x^{n-1}, n > 0$

```
(define (power x n)
  (if (= n 0)
      1
      (* x (power x (- n 1)))))
```

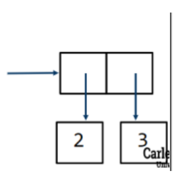
```
(power 2 4)
==> (* 2 (power 2 3))
==> (* 2 (* 2 (power 2 2)))
==> (* 2 (* 2 (*  2 (power 2 1))))
==> (* 2 (* 2 (*  2 (* 2 (power 2 0)))))
==> (* 2 (* 2 (*  2 (* 2 1))))
==> (* 2 (* 2 (*  2 2)))
==> (* 2 (* 2 4))
==> (* 2 8)
==> 16
```

Expansion

Contraction

5

(cons 2 3)




```
> (define x (cons 2 3))
> (car x) ; Extraction from 1st cell
2
> (cdr x) ; Extraction from 2nd  cell
3
```

```
(define (list_sum items)
  (if (empty? items)
      0
      (+ (car items)
         (list_sum (cdr items)))))
```

- Retrieve the *n*th item in list `items` (*first item is item 0*)

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
```

- Summing all number in  a list of lists

```
(define (deep-list-sum p)
  (if
   (null? p) 0      ; base case
   (+               ; Sum
    (cond           ; 1st element of the list
          [(list? (car p)) ; inner list
             (deep-list-sum (car p))]
          [(null? (car p)) 0] ; end of list
          [else    (car p)])  ;primitive number
    (deep-list-sum (cdr p)))));with rest oflst
```

```
(define (contains? items target)
  (cond
      [(empty? items) false]
      [(= (car items) target) true]
      [else (contains? (cdr items) target)]))
```

Carleton University

```
> ((lambda (x) (* (sin x) (sin x))) (/ pi 4))
0.5
```

Creating Local Variables

```
(define (roots a b c)
  (cons (/ (+ (- b)
              (sqrt (- (* b b) (* 4 a c))))
           (* 2 a)) ; x = (-b + sqrt(b^2-4ac)) / 2a
        (/ (- (- b)
              (sqrt (- (* b b) (* 4 a c))))
           (* 2 a)))) ; x = (-b - sqrt(b^2-4ac)) / 2a
```

## let Expressions

- Read this as:

`let` <*var*$_1$> have the value <*exp*$_1$> and
<*var*$_2$> have the value <*exp*$_2$> and
...
<*var*$_n$> have the value <*exp*$_n$>
in <*body*>

```
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c)))))
    (cons (/ (+ (- b) d) (* 2 a))
          (/ (- (- b) d) (* 2 a)))))
```

- d is now beside the expression that calculates its value
- value of d is calculated above the code where it's used

- A common list operation: apply a transformation to each element in a list, returning the results in a list

```
;; Scale each element in a list by a factor
(define (scale-list items factor)
  (if (null? items)
      '()
      (cons (* (car items) factor)
            (scale-list (cdr items) factor))))
> (scale-list '(1 2 3 4 5) 10)
'(10 20 30 40 50)
```

Carleton University



```
(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))
```

```
> (accumulate + 0 '(1 2 3 4 5))
15
```

```
> (foldr + 0 '(1 2 3 4))
10
```

```
(define new-count-up
  (let ((counter 0))
    (lambda ()
      (set! counter (+ counter 1))
      counter)))
> (new-count-up)
1
> (new-count-up)
2
```

`new-count-up`



## begin

- Special form

(begin *exp*$_1$ *exp*$_2$ .. *exp*$_k$)

- Expressions *exp*$_1$ *exp*$_2$ .. *exp*$_k$ are evaluated in sequence
- Value of final expression *exp*$_k$ is returned as the value of the entire `begin` form

```
(begin (set! counter (- counter 1))
       counter)
```

Carl Un

## equal?

- By default, same as `eqv?`
- For certain datatypes (e.g, strings, pairs, lists), `equal?` has further specification
- To determine if two lists are equal (contain equal elements arranged in the same order)

```
> (equal? '(1 2 3) '(1 2 3))
#t
> (equal? '(1 2 3) '(4 5 6))
#f
```

Carleton

## eq?

- `eq?` is true if its arguments refer to the same object
- Used to determine if two symbols are the same; e.g.,

```
> (eq? 'apples 'oranges)
#f
> (eq? 'apples 'apples)
#t
```

# SYSC 3101 Winter 2022
# Applications of programming techniques
# (Sorting)

Carleton University

# Sorting Problems

- Input:
  - a list of elements
  - a comparison procedure.
- Output:
  - list containing same elements as the input list ordered according to the comparison procedure
- Method:
  - Best-First Sort
  - Insertion Sort
  - ……………

# Best- First Sorting

- Find the best element in the list and put that at the front.

# Best- First Sorting

- Find the best element in the list and put that at the front.
- Steps:
  - Find the best element
  - Remove the element of the list
  - Add that element at the first
  - Repeat till
    - List is null
    - List has only one element.

# Best- First Sorting

- Find the best element:
  - Remember : Divide and conquer → Find the better
  - Then , Find the best

```
(define (find-better cf p1 p2)
  (if (cf p1 p2) p1 p2))

(define (find-best cf lst)
  (if  (null? (cdr lst))
          (car lst)
          (find-better cf (car lst) (find-best cf (cdr lst)))))
```

# Best- First Sorting

- Remove the element of the list

```
(define (Remove-element lst el)
  (if
   (null? lst) null
   (if (equal? (car lst) el)
       (cdr lst) ; found match, skip this element
       (cons (car lst) (Remove-element (cdr lst) el)))))

>(Remove-element '( 5 15) (find-better > 15 5)
```

# Best- First Sorting

```
(define (best-first-sort cf lst)
  (if (null? lst)
      null
      (cons (find-best cf lst)
            (best-first-sort cf (Remove-element lst (find-best cf lst))))))

>( best-first-sort < '(1 8 6 4 3 2 4 9 7))
```

# Best- First Sorting

```
(define (best-first-sort cf lst)
  (if (null? lst)
      null
      (cons (find-best cf lst)
            (best-first-sort cf (Remove-element lst (find-best cf lst))))))

>( best-first-sort < '(1 8 6 4 3 2 4 9 7))
```



Carleton University

# Best- First Sorting

- Optimize your code

```
(define (best-first-sort-let cf p)
 (if
  (null? p) null
  (let ((best (find-best cf p)))
    (cons best (best-first-sort-let cf (Remove-element p best))))))
```

# Insertion Sort

- Method: put the first element in the list in the right place in the list that results from sorting the rest of the elements.
- Steps:
  - Insert item in the sorted list
  - repeat

# Insertion Sort

- Insert item in the sorted list

```
(define (insert-one-sortedList cf el p) ; requires: p is sorted by cf
  (if
   (null? p) (list el)
   (if  (cf el (car p))
      (cons el p)
      (cons (car p) (insert-one-sortedList cf el (cdr p))))))
```

Carleton University

# Insertion Sort

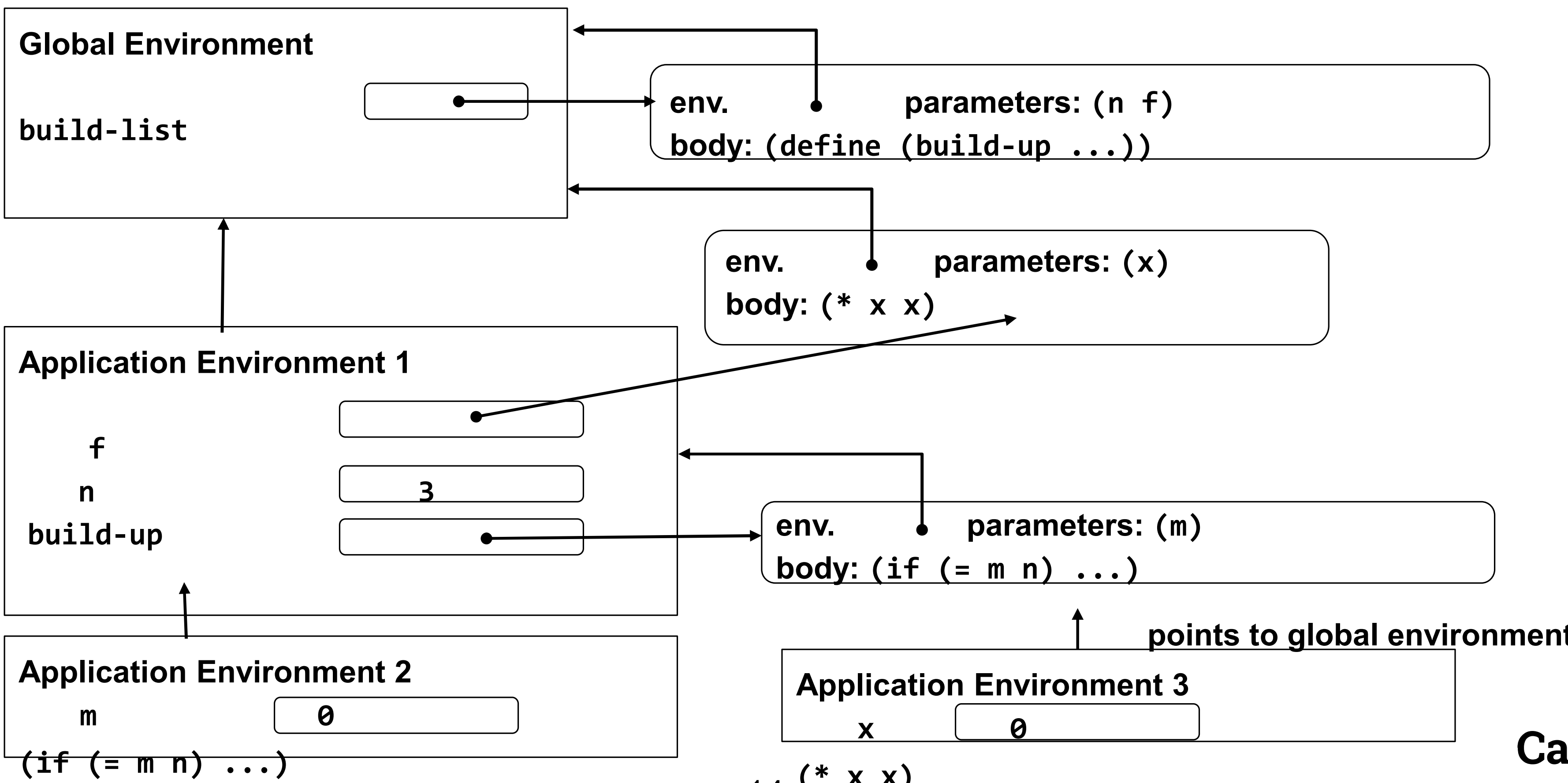
- Repeat

```
(define (insertion-sort cf p)
 (if
  (null? p) null
  (insert-one-sortedList cf (car p) (insertion-sort cf (cdr p)))))
```

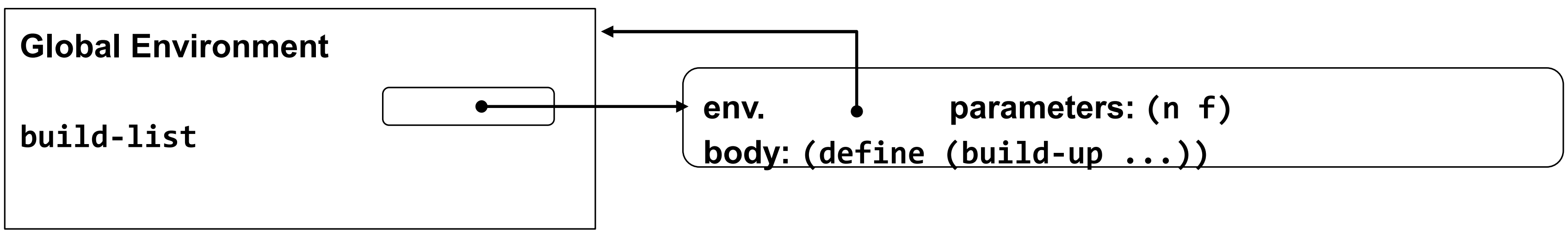
# Lab 6 Question 2

```
(define (build-list n f)
   (define (build-up m)
    (if (= m n)
        '()
        (cons (f m) (build-up (+ m 1)))))

  (build-up 0))

(build-list 3 (lambda (x) (* x x)))
```

# Lab 6 Question 2



# Lab 6 Question 2



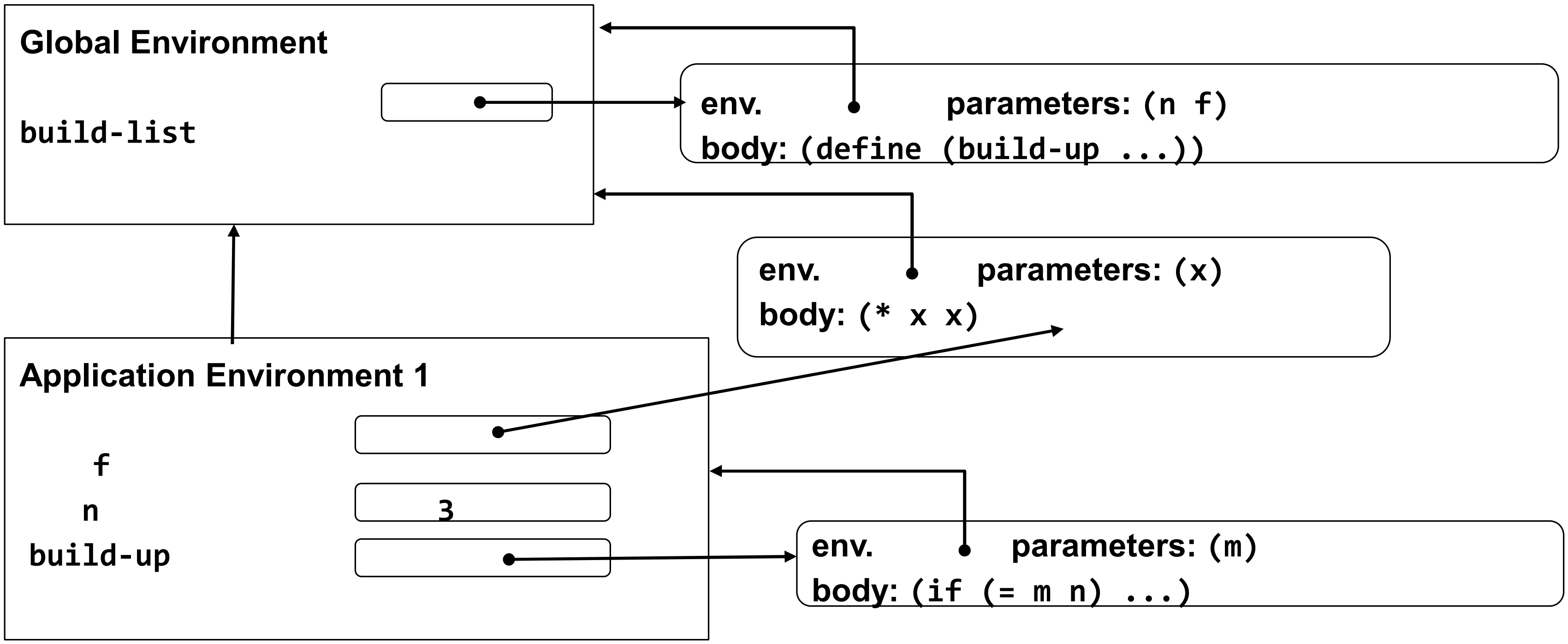

Step 1: `(define (build-list n f) ...)` is evaluated. This defines variable `build-list` in the global environment and binds it to the procedure. Notice that the procedure's environment pointer points to the global environment.

points to global environment



Carleton University

# Lab 6 Question 2




Step 2: The expression `(build-list 3 (lambda (x) (* x x)))` is evaluated.

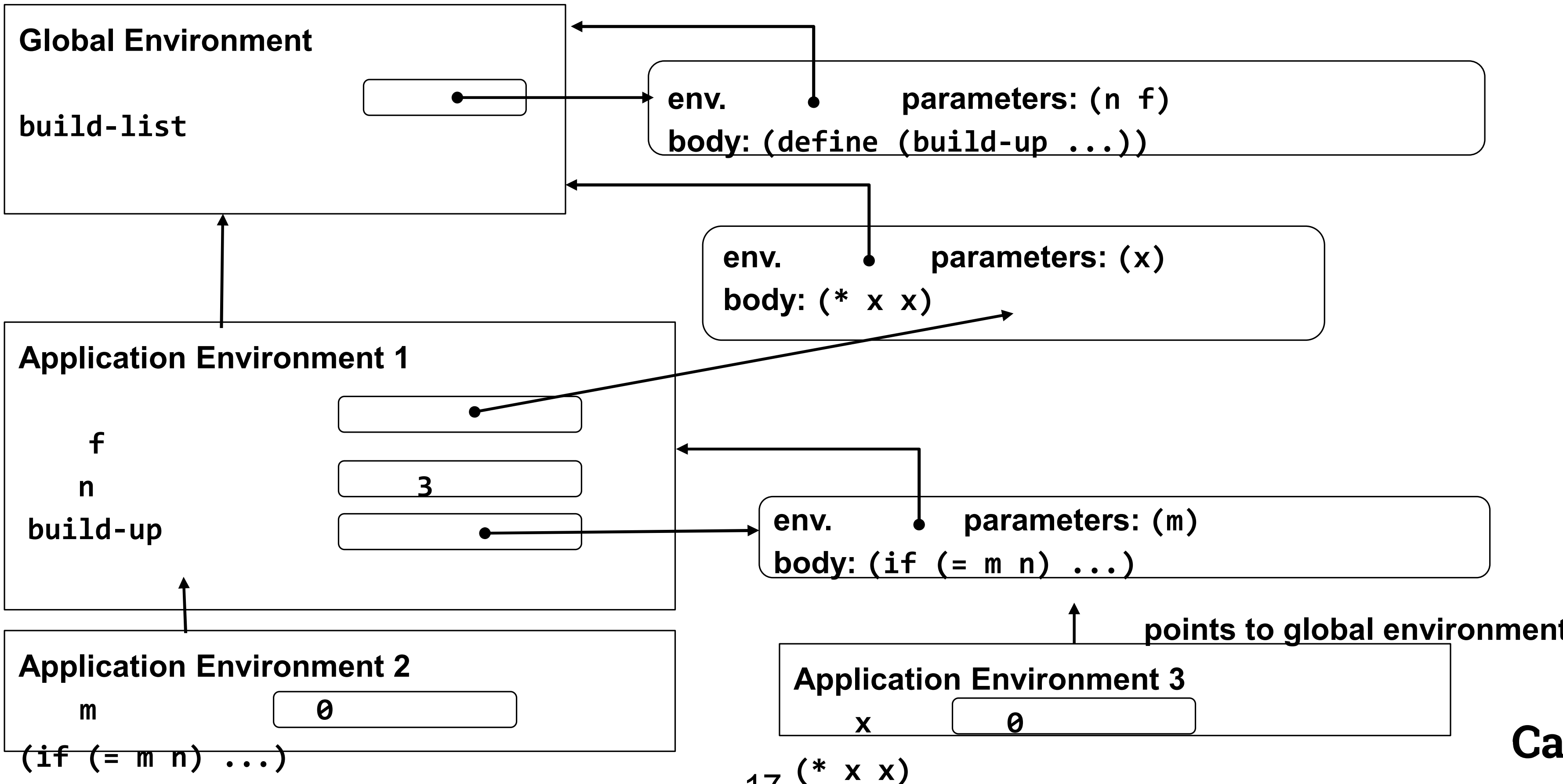
Step 3: `build-list` is called.

Step 4: The first expression in the body of `build-list`, `(define (build-up m)...)`, is evaluated.



Carleton University

# Lab 6 Question 2






Carleton University

# Lab 6 Question 2

points to App. Env. 1

Application Environment 2
`m` `0`

`(if (= m n) ...)`

points to App. Env. 1

Application Environment 4
`m` `1`

`(if (= m n) ...)`

points to App. Env. 1

Application Environment 6
`m` `2`

`(if (= m n) ...)`

points to App. Env. 1

Application Environment 8
`m` `3`

`(if (= m n) ...)`

points to global environment

Application Environment 3
`x` `0`

`(* x x)`

points to global environment

Application Environment 5
`x` `1`

`(* x x)`

points to global environment

Application Environment 7
`x` `2`

`(* x x)`

# SYSC 3101 Winter 2022
# The Environment Model of Evaluation
# Lexical vs Dynamic Scope

The slides are adapted form SYSC 3101 W19,D.L. Bailey, Department of Systems and Computer Engineering

Carleton University

# Other Notations

- The notation used in *SICP* isn't the only way to depict the environment model
- Here's a prototype of an online Scheme interpreter visualizes environments using "Python Tutor notation"

  [https://scheme.cs61a.org/](https://scheme.cs61a.org/)



Carleton University

# Scope: Lexical vs. Dynamic

- Racket and many (most?) widely-used languages use lexical scoping
  - when a procedure is called, the procedure's defining environment is extended
- Some languages (e.g., original LISP, Logo) use dynamic scoping
  - when a procedure is called, the current environment is extended

# Scope: Lexical vs. Dynamic

```
(define PI 3.1415926)
(define (area radius)
  (* PI radius radius))
(define (mess-up PI)
  (area (+ PI 5)))
(mess-up 4)
```

- What does `mess-up` return (lexical scope)?
- What would `mess-up` return if Racket used dynamic scope?

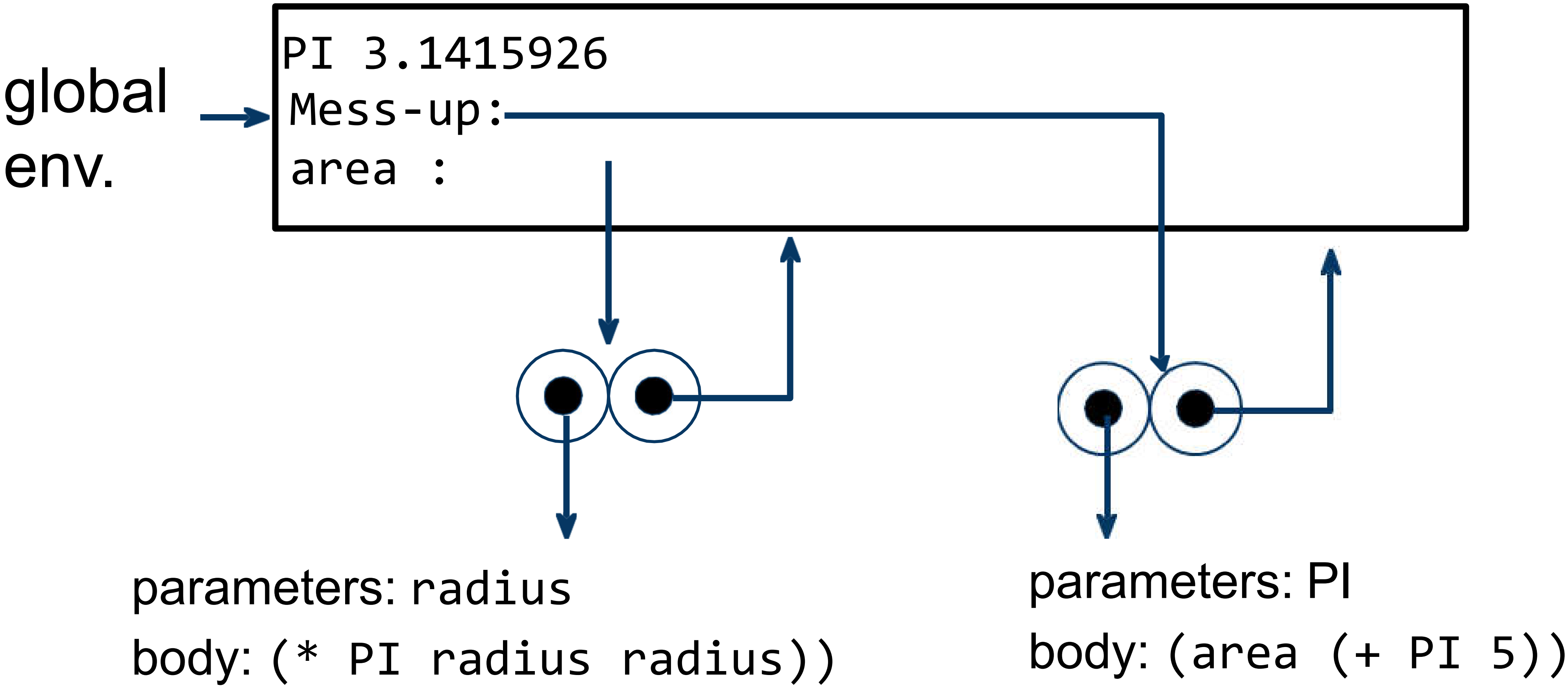
# Lexical Scope

- Draw an environment diagram that shows that `(mess-up 4)` returns the area of a circle with radius 9

# Lexical Scope

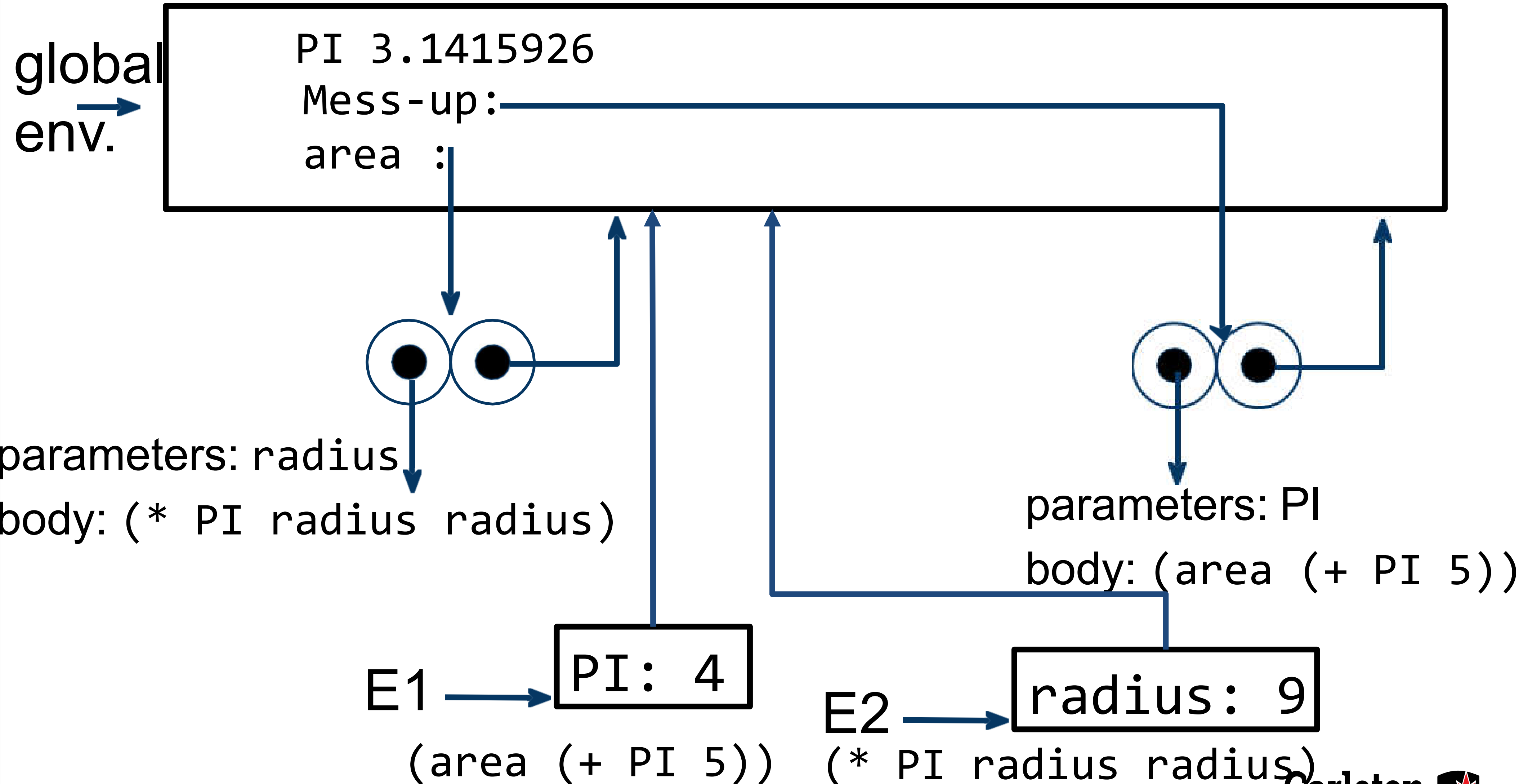
- `PI` is defined in the global environment
- the definition of procedure `area` was evaluated with respect to the global env.
  - its procedure object stores a pointer to the global environment
- the definition of procedure `mess-up` was evaluated with respect to the global env.
  - its procedure object stores a pointer to the global environment

# Evaluate (define …)




- Add binding to global environment's frame

# Evaluate `(mess-up 4)`



# Lexical Scope `(mess-up 4)`

- when `area` is called by `mess-up`, a new frame is created with parameter `radius` bound to 9, and `area`'s frame extends the global environment
- when `(* PI radius radius)` is evaluated, there's no binding for `PI` in `area`'s frame, so Racket uses `PI` in the global environment, and calculates 3.141526 * 9 * 9

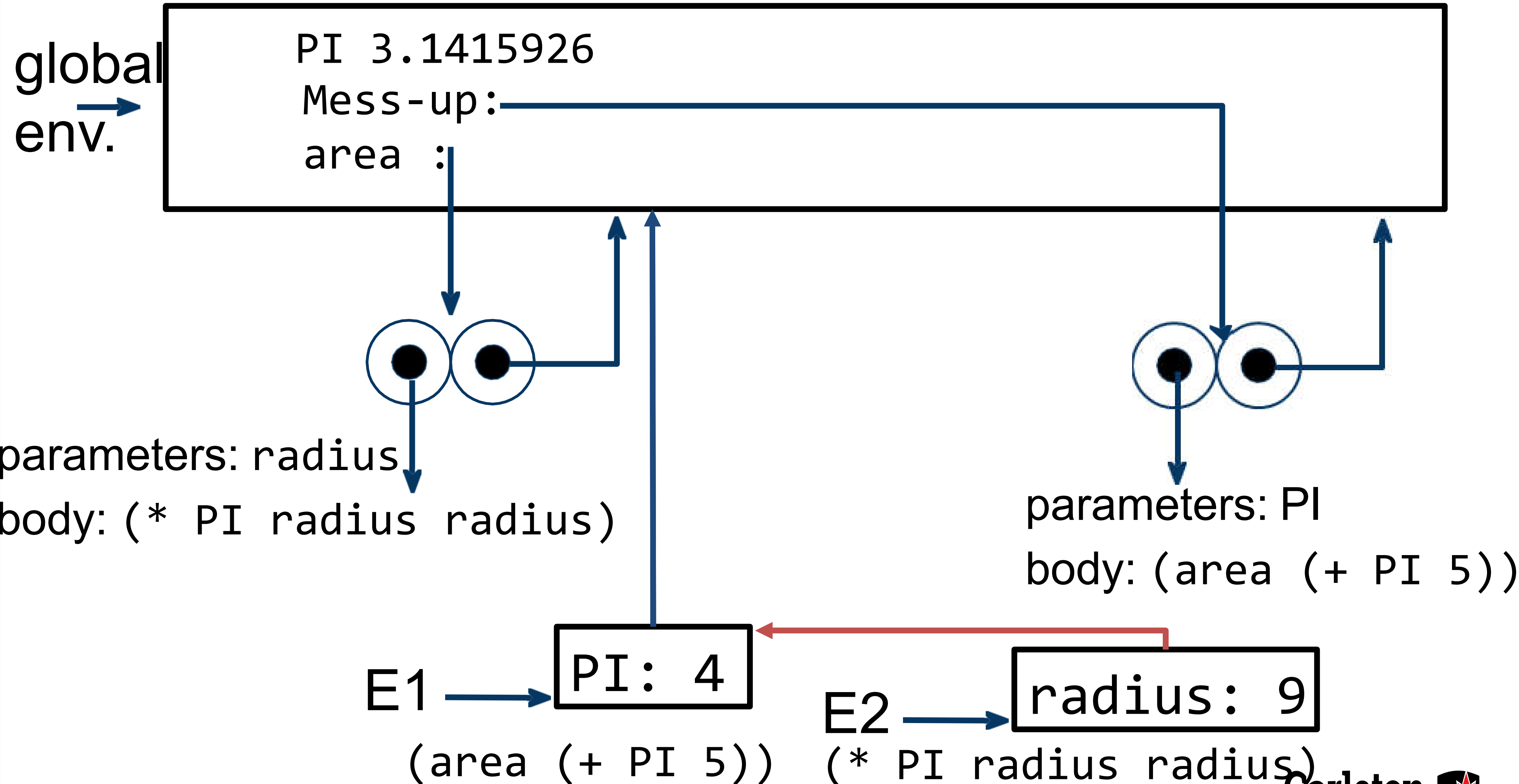
# Dynamic Scope

- Draw an environment diagram that shows that `(mess-up 4)` returns 324, which is not the area of a circle with radius 9

# Dynamic Scope

- when `mess-up` is called, its environment extends the global environment
  - parameter `PI` in `mess-up`'s frame is bound to 4

# Evaluate (mess-up 4)



# Dynamic Scope

- when `area` is called by `mess-up`, a new frame is created with parameter `radius` bound to 9, and `area`'s frame extends the current environment; i.e., `area`'s frame points to `mess-up`'s frame
- when `(* PI radius radius)` is evaluated, there's no binding for `PI` in `area`'s frame, so Racket uses `PI` in `mess-up`'s frame, and calculates 4 * 9 * 9

# make-adder Revisited

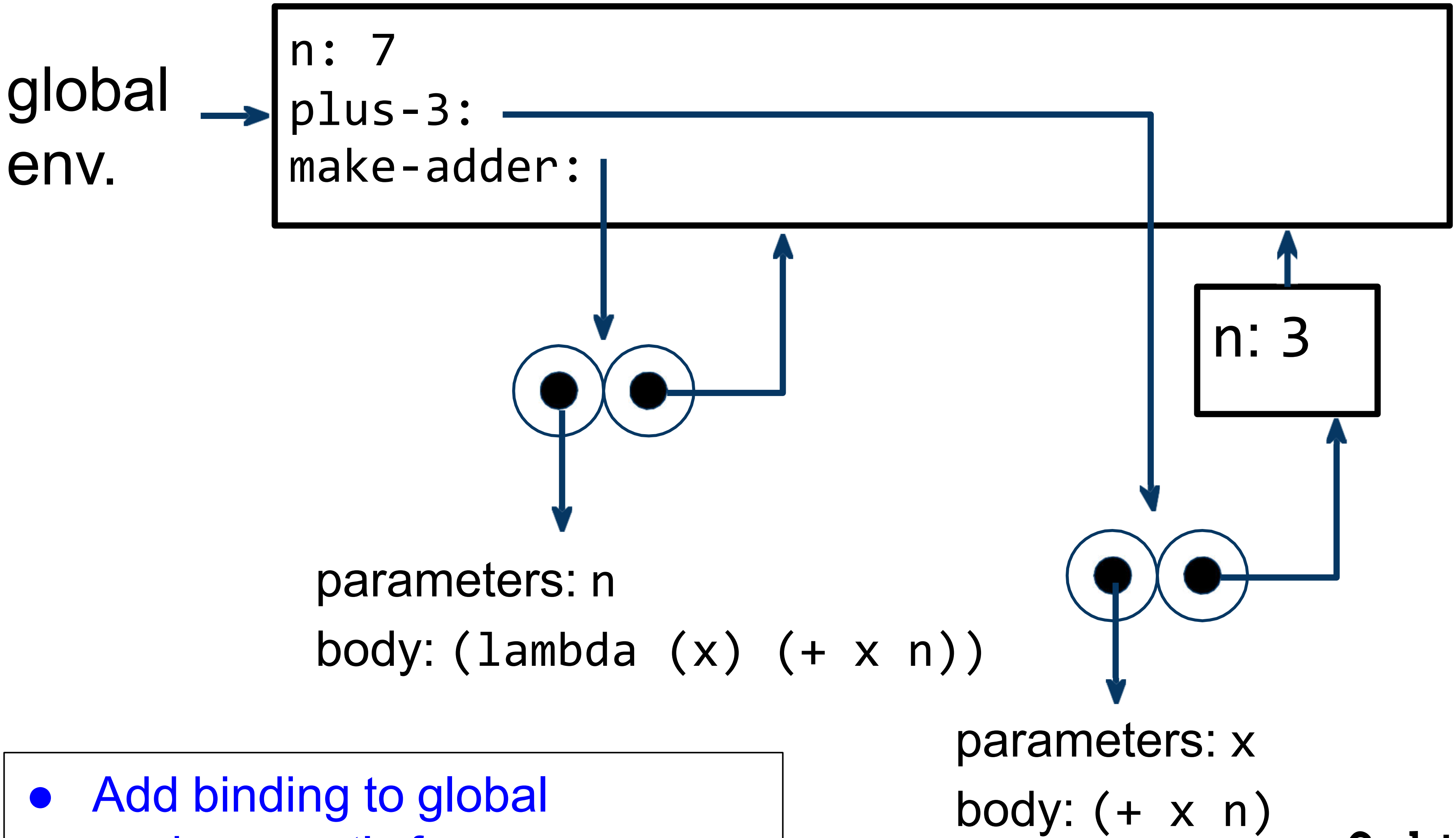
```
(define (make-adder n)  (lambda (x)
   (+ x n)))
(define plus-3 (make-adder 3))
(define n 7)

(plus-3 n)
```

- What happens when these expressions are evaluated?



Carleton University

# Evaluate `(define …)`

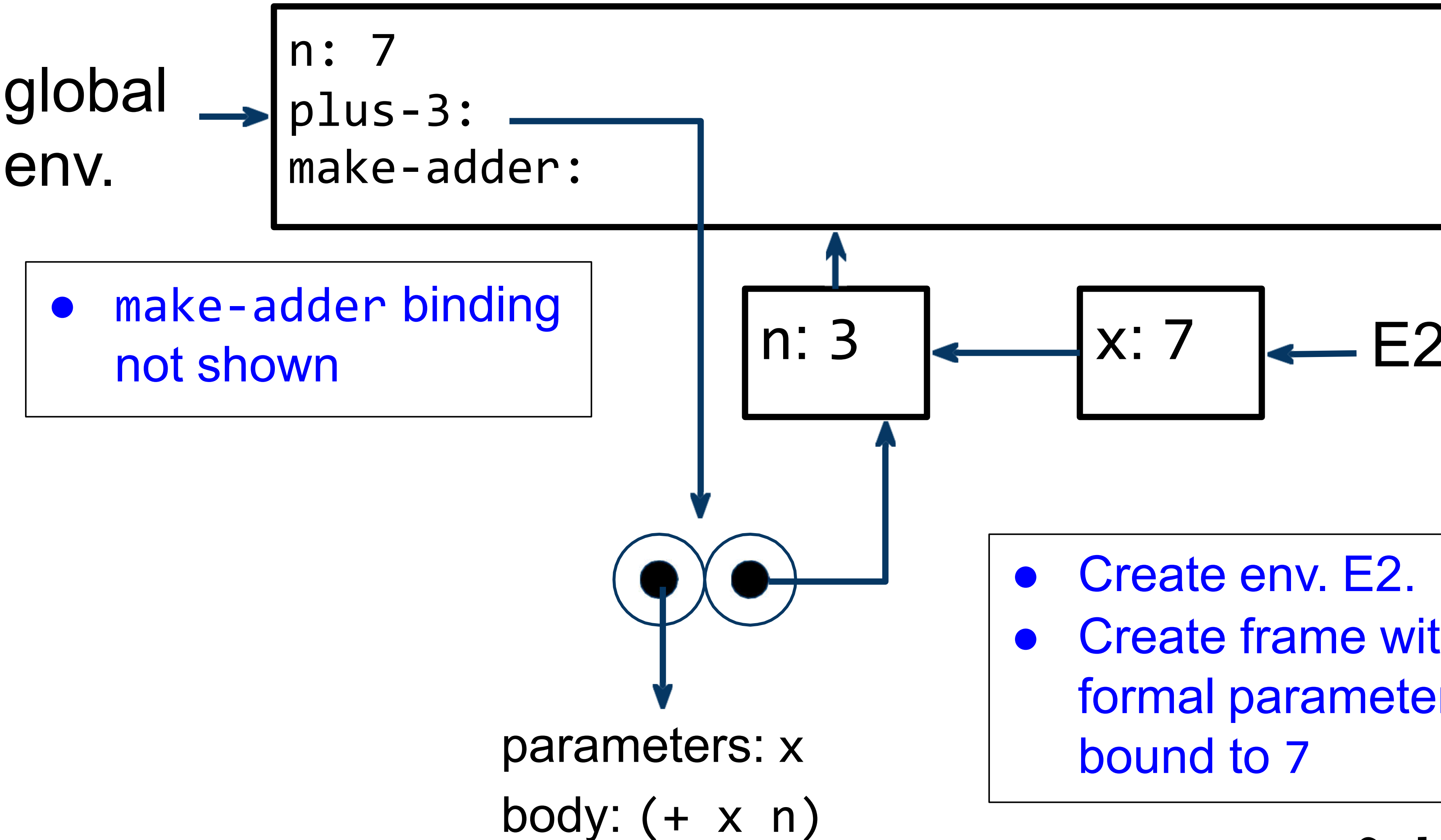



- Add binding to global environment's frame



Carleton University

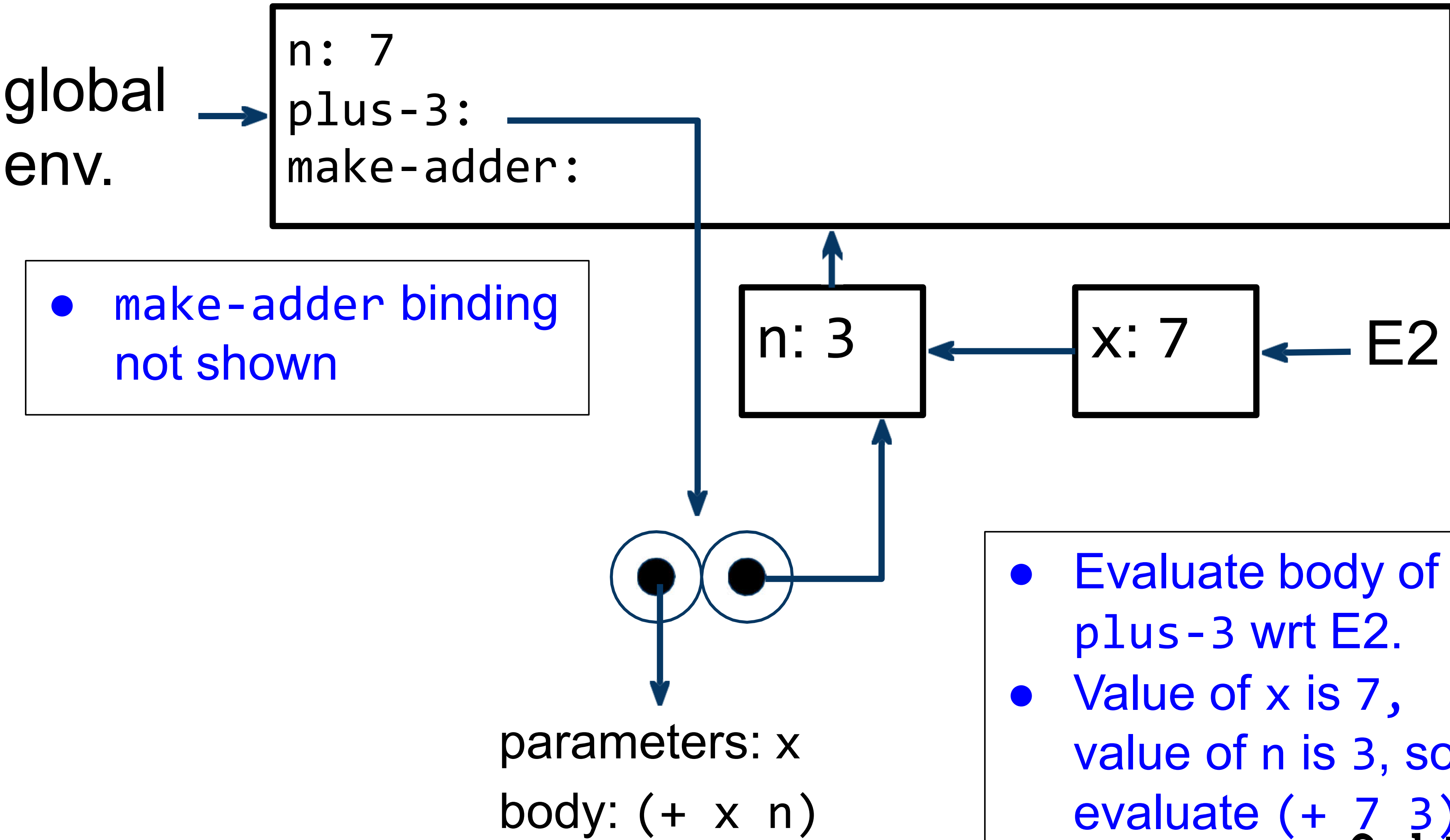
# Evaluate `(plus-3 n)`




- `make-adder` binding not shown

- Create env. E2.
- Create frame with formal parameter x bound to 7

# Evaluate `(plus-3 n)`




- `make-adder` binding not shown

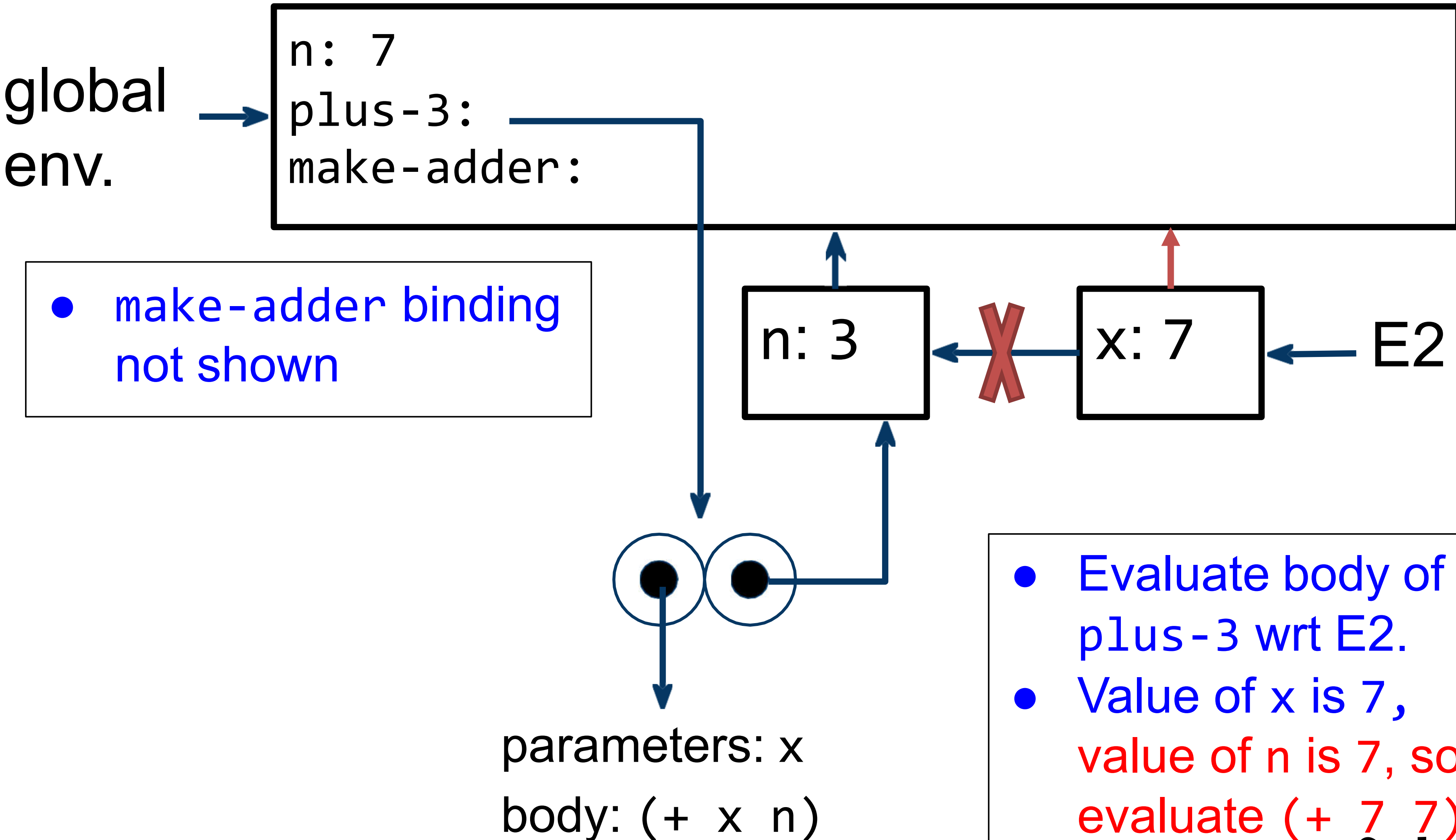
- Evaluate body of `plus-3` wrt E2.
- Value of x is `7`, value of n is `3`, so evaluate `(+ 7 3)`, which is `10`

Carleton University

# make-adder :Dynamic Scope

- If Scheme used **dynamic scope**, when a procedure is called a new frame would be created, but this *frame would point to the current environment, not the procedure's defining environment*

# Evaluate `(plus-3 n)`




- `make-adder` binding not shown

- Evaluate body of `plus-3` wrt E2.
- Value of x is `7`, value of n is 7, so evaluate `(+ 7 7)`, which is 14



Carleton University

# make-adder :Dynamic Scope

- (plus-3 n)
  - (plus-3 n) is evaluated with respect to the global environment
  - calling (plus-3 7) creates a frame in which parameter x is bound to 7.This frame extends the current environment (*the global environment*)

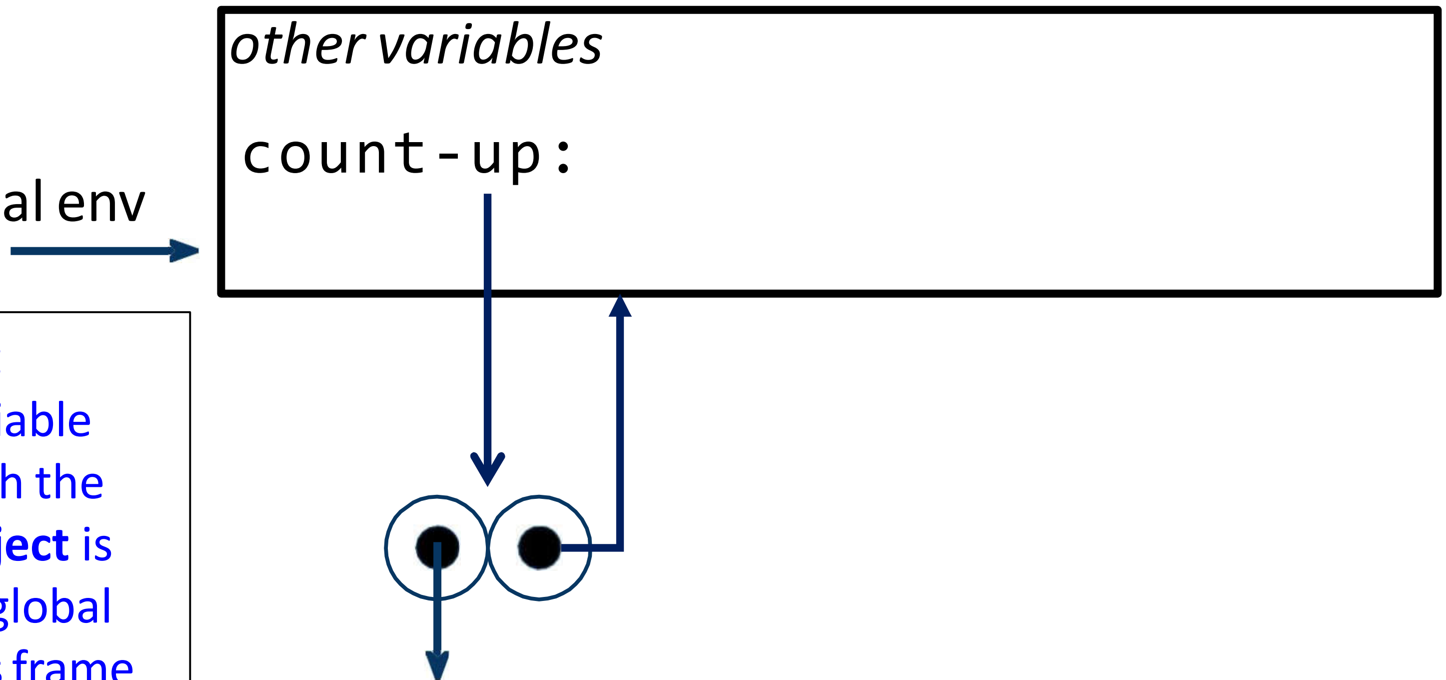
# Env. Model with let(Lexical

```
(define (count-up)
  (let ([counter 0])
    (set! counter (+ counter 1))
    counter))

(define (new-count-up)
  (let ([counter 0])
    (lambda () (set! counter (+ counter 1))
counter)))

(define x (new-count-up))
(define y (new-count-up))
```

# define (count-up)

```
(define (count-up)
 (let ([counter 0])
     (set! counter (+ counter 1))
      counter))
```

global env




A binding that associates variable count-up with the **procedure object** is added to the global environment's frame

parameters:
body: (let ([counter 0])
  (set! counter (+ counter 1))
  counter))



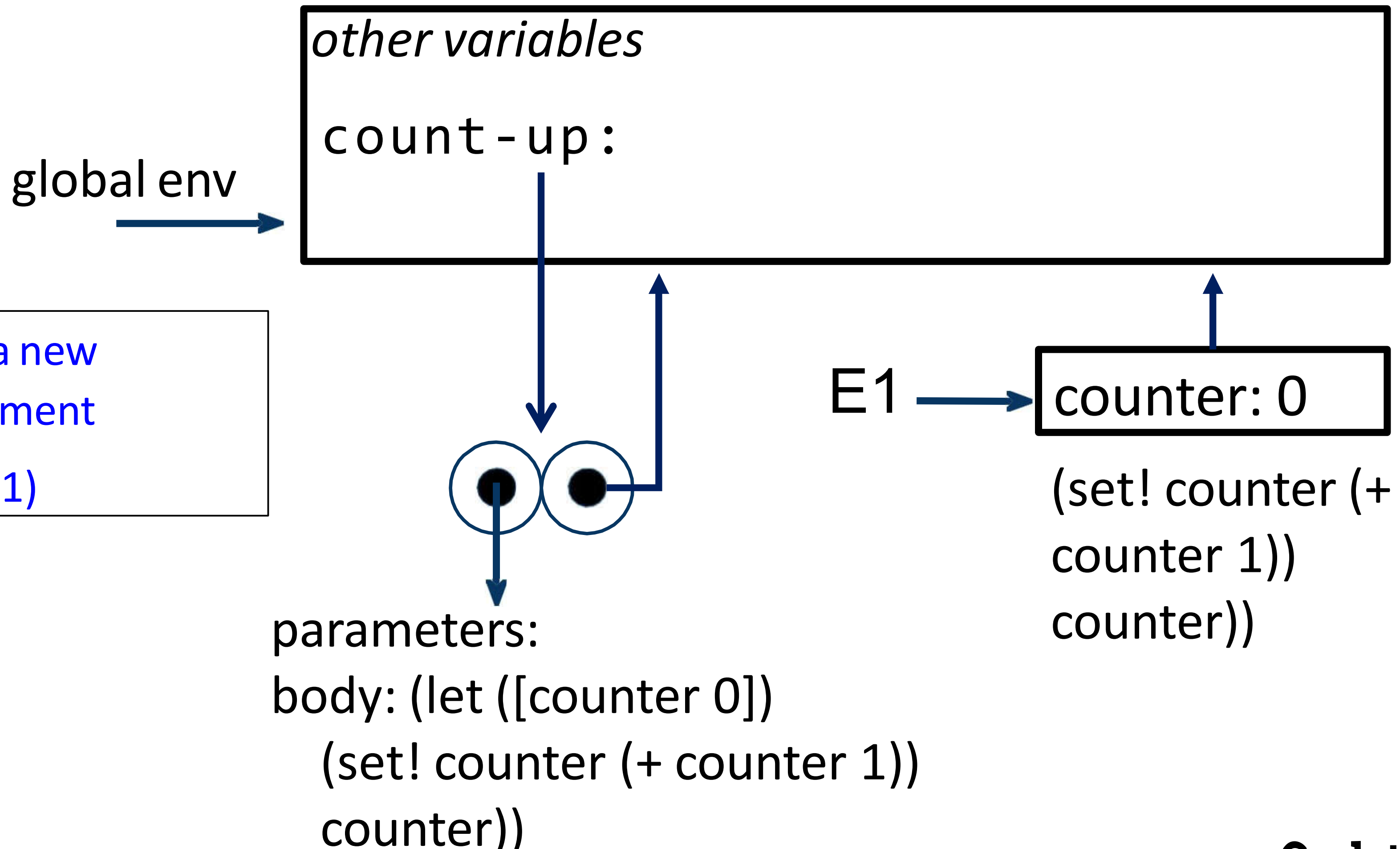
Carleton University

# Applying (count-up) Procedure

`(count-up)`

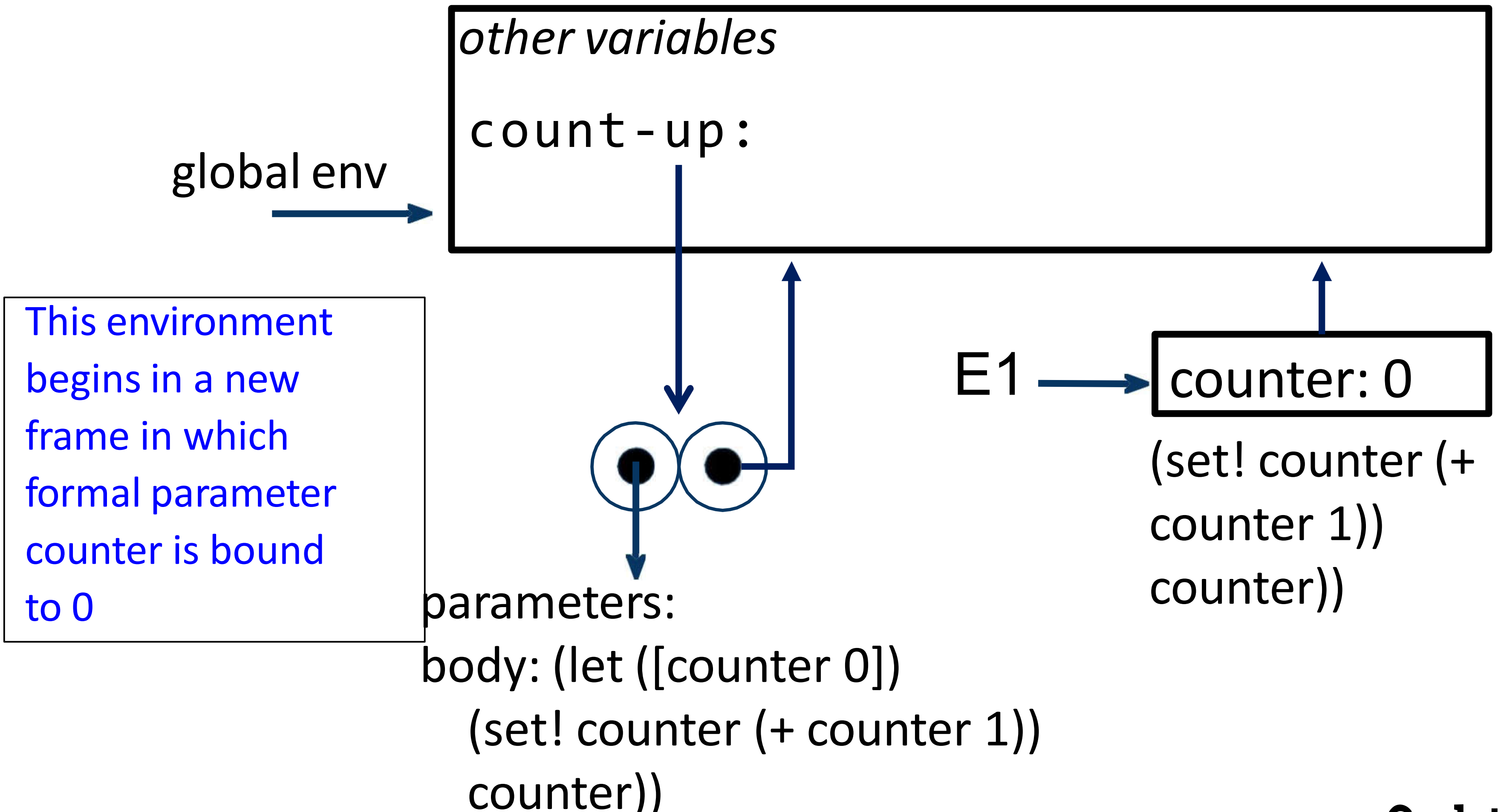
Create a new environment

(call it E1)



# Applying (count-up) Procedure

`(count-up)`

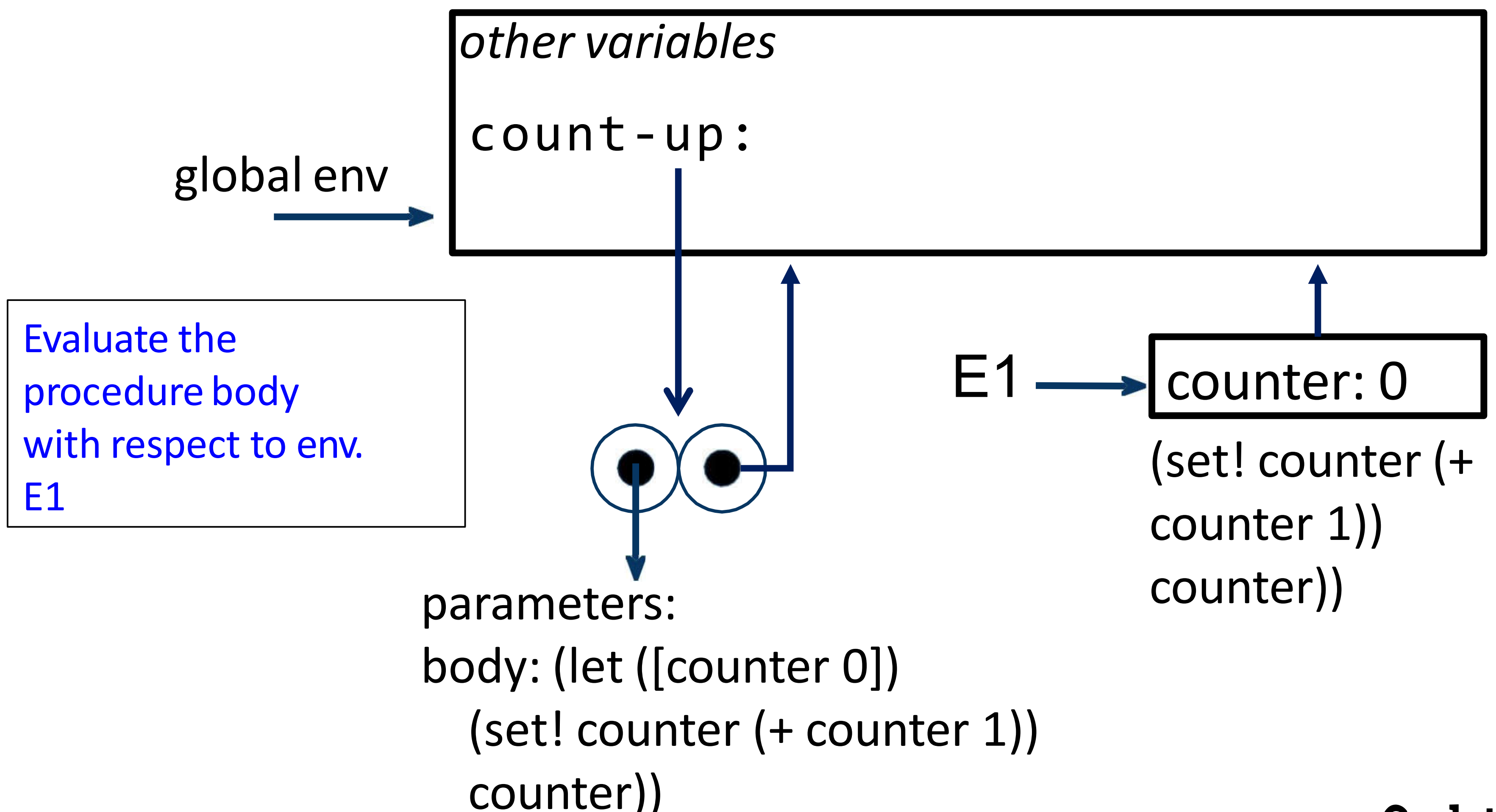





Carleton University

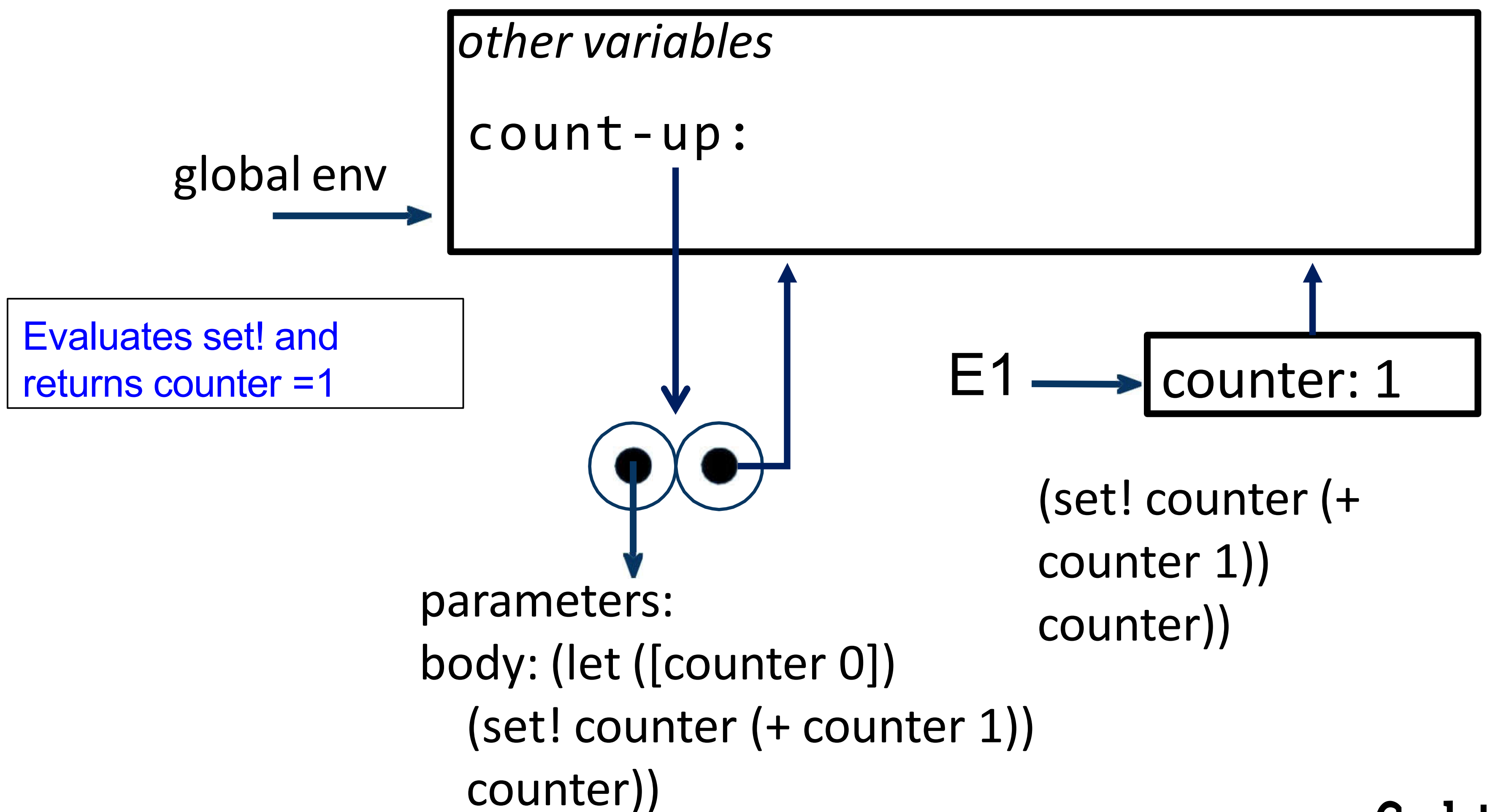
# Applying (count-up) Procedure

`(count-up)`



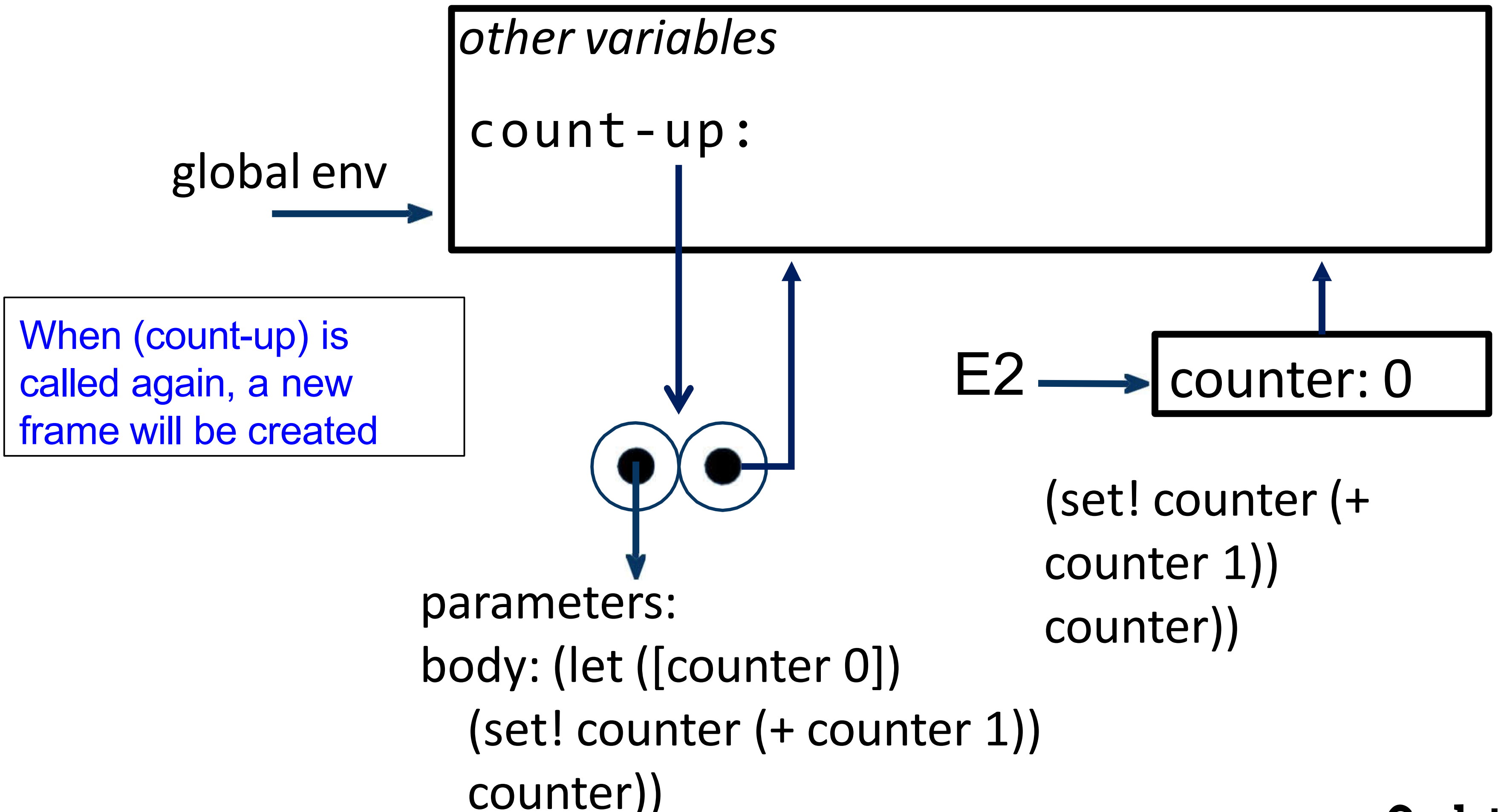
# Applying (count-up) Procedure

`(count-up)`



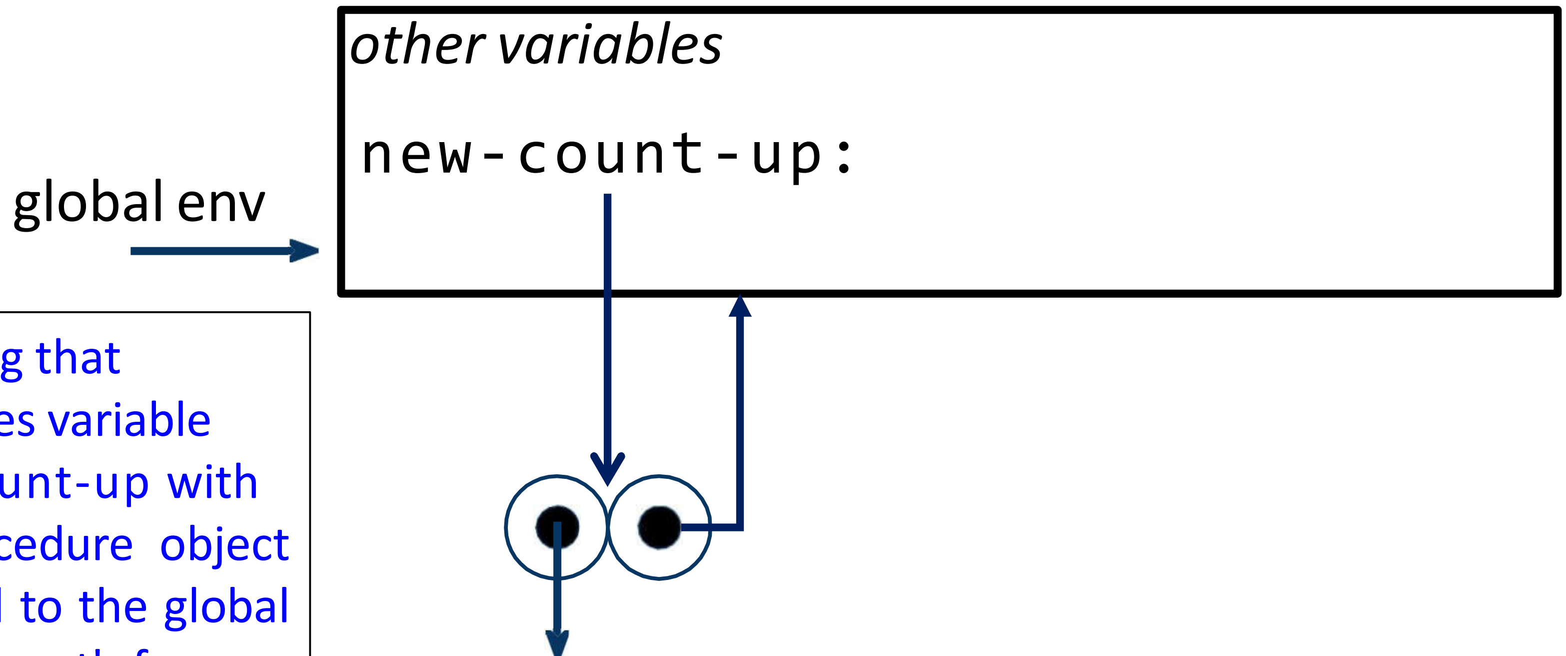
# Applying (count-up) 2Nd time

`(count-up) – 2nd time`



# define (new-count-up)

```
(define (new-count-up)
```

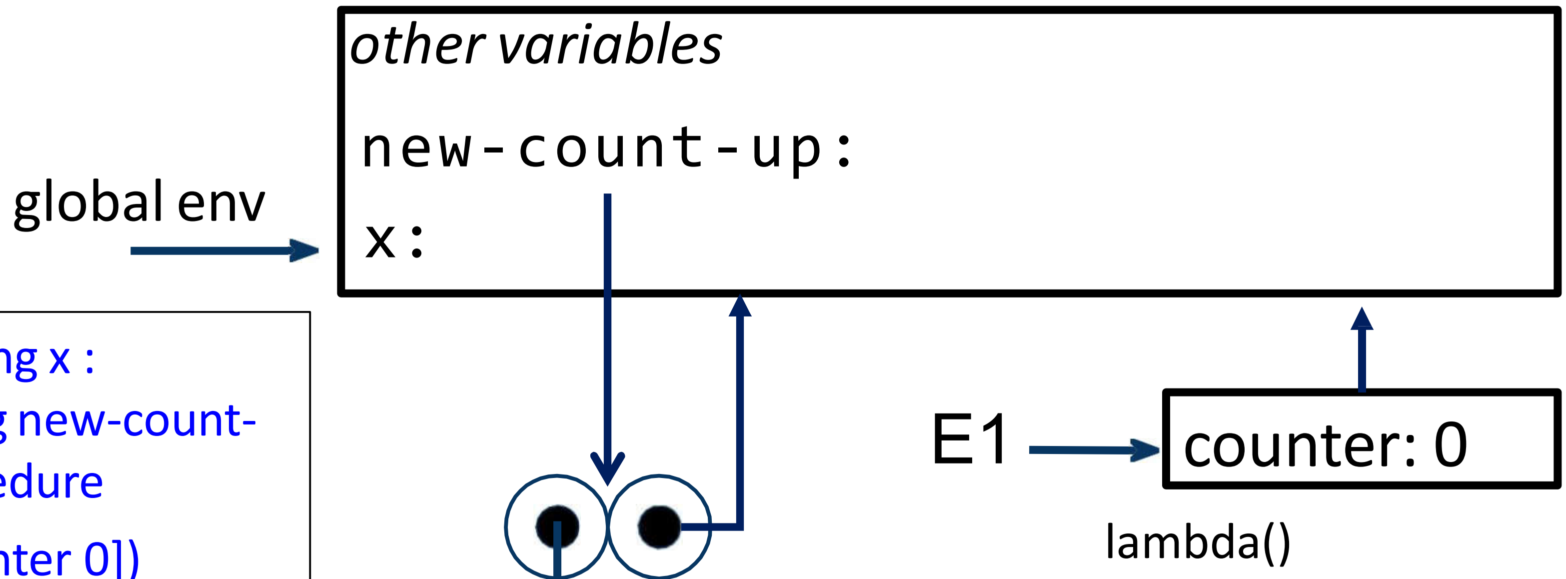



A binding that associates variable `new-count-up` with the procedure object is added to the global environment's frame

parameters:
body: (let ([counter 0])
  lambda()
  (set! counter (+ counter 1))
  counter))

# Applying x (new-count-up) 1

```
(define x (new-count-up))
```




Evaluating x :
Applying new-count-up procedure

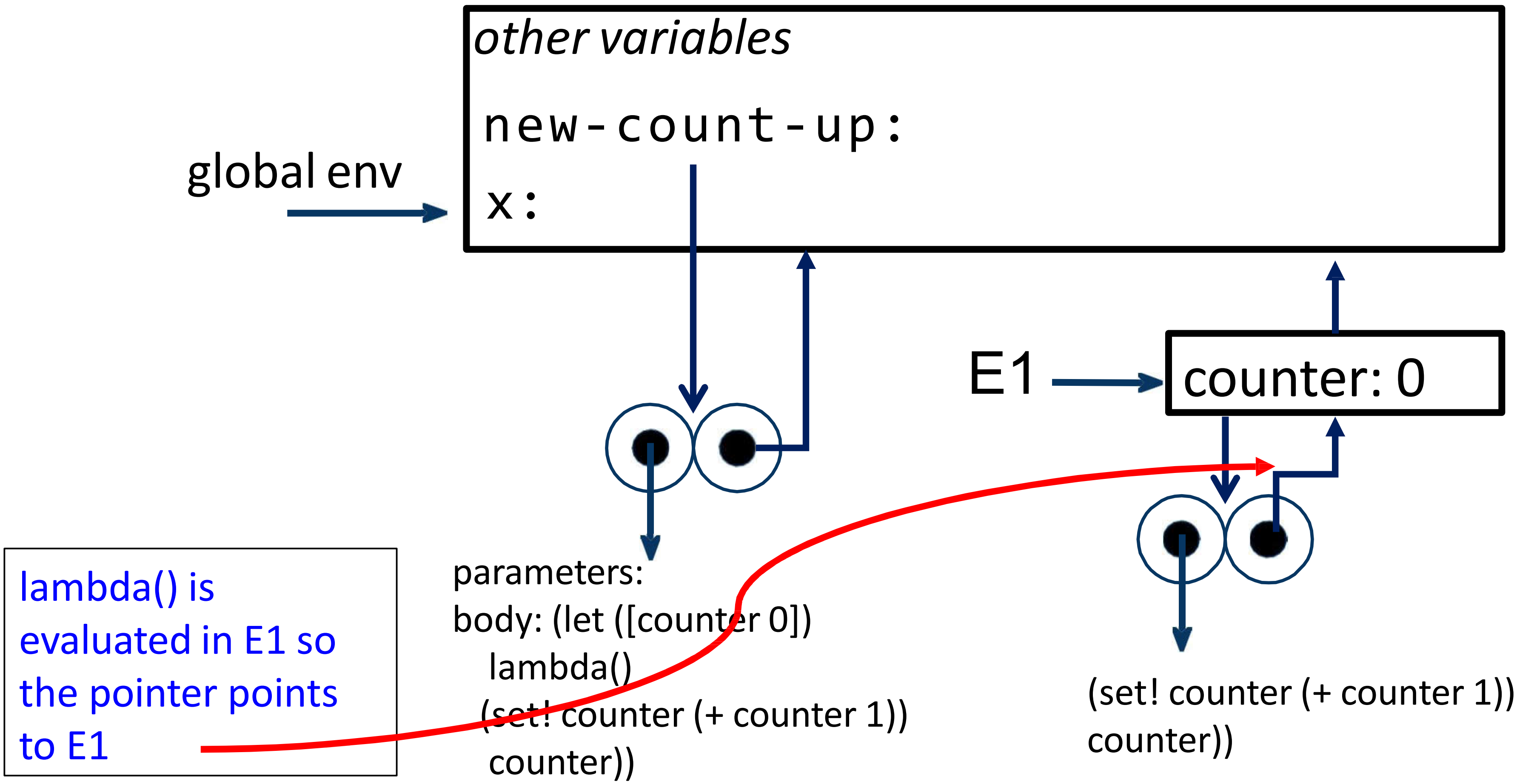
let([counter 0]) creates environment E1 with counter bound to 0

parameters:
body: (let ([counter 0])
lambda()
(set! counter (+ counter 1))
counter))

lambda()
(set! counter (+ counter 1))
counter))

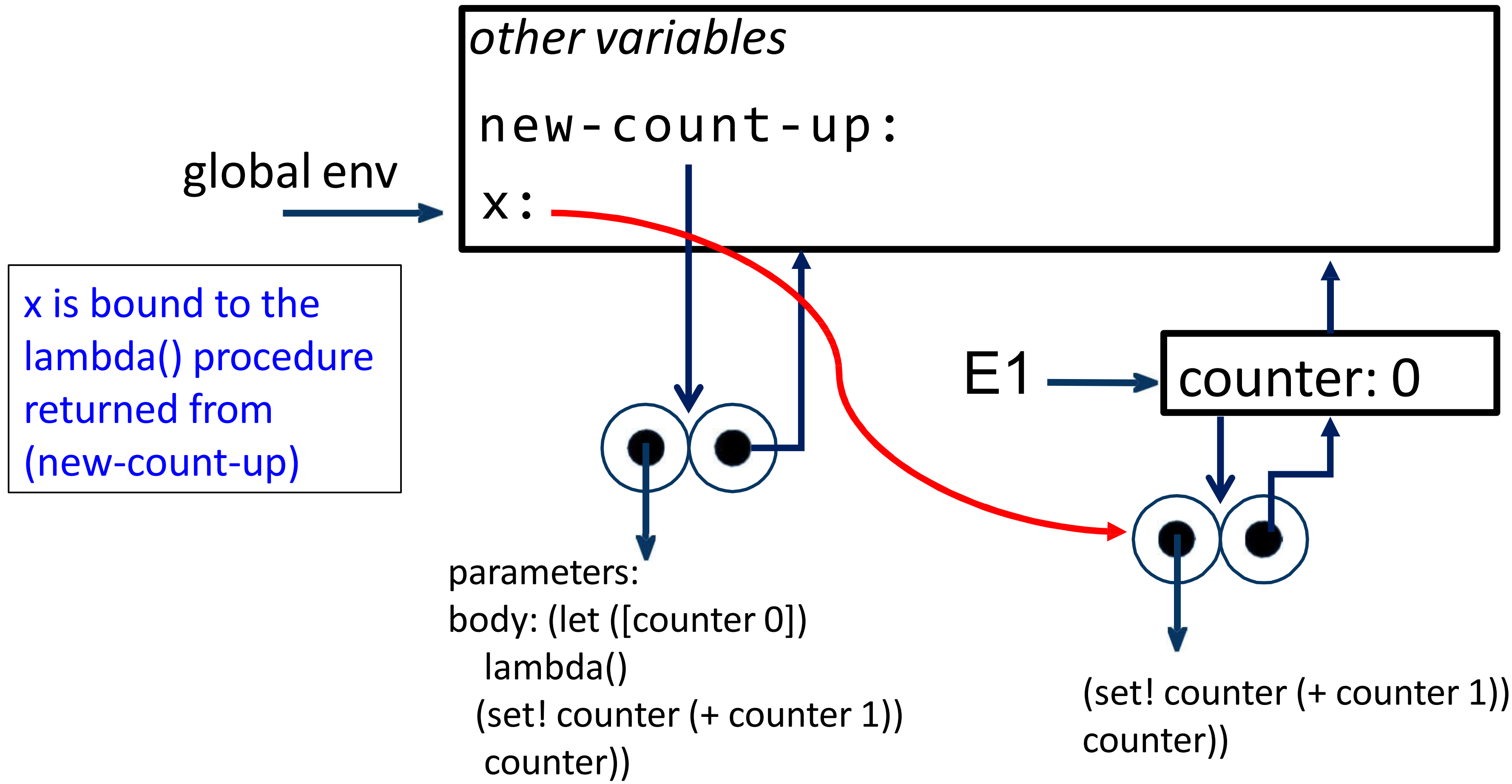
# Applying x (new-count-up) 2

```
(define x (new-count-up))
```



# Applying x (new-count-up) 3

```
(define x (new-count-up))
```

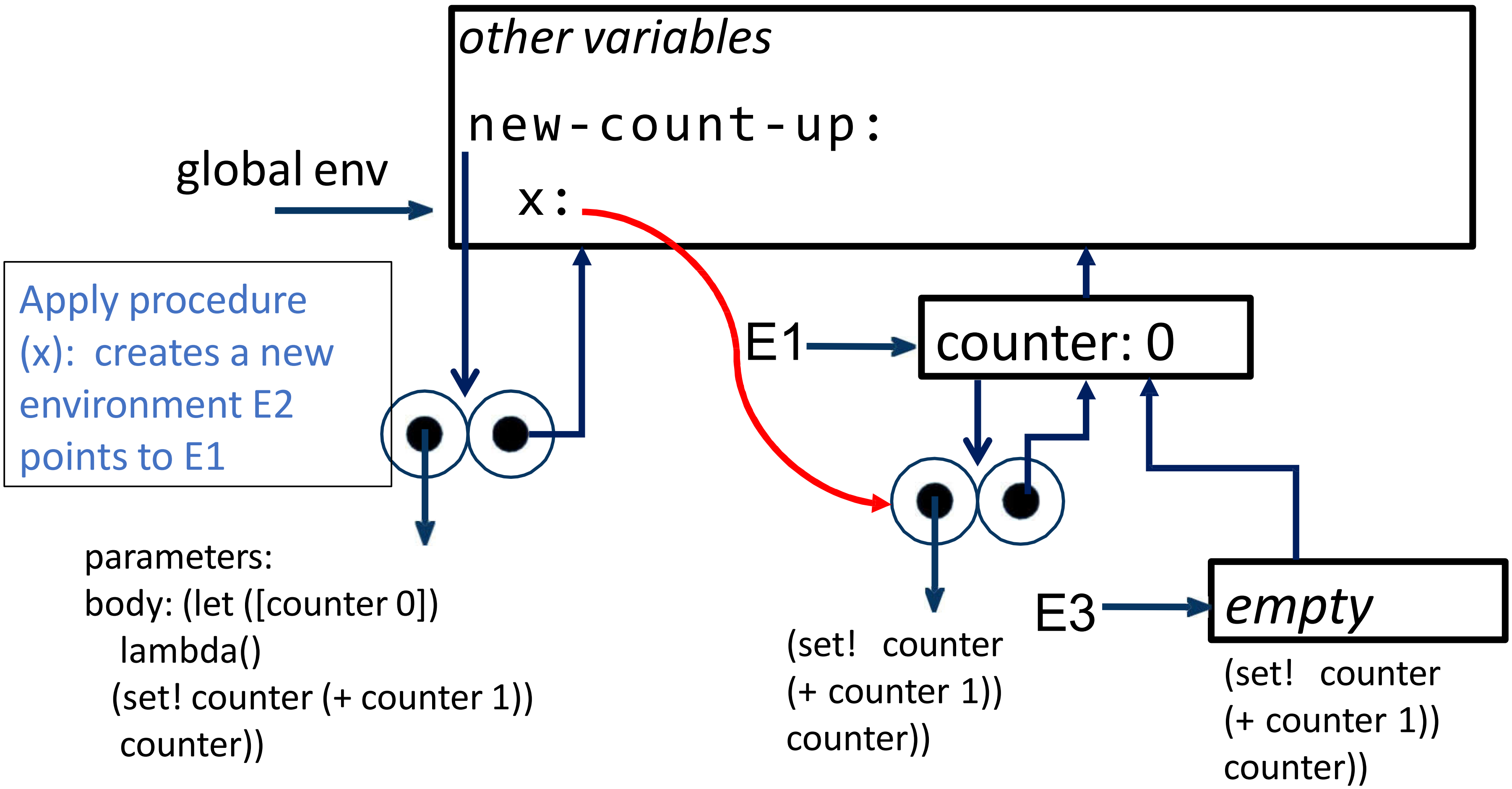





Carleton University

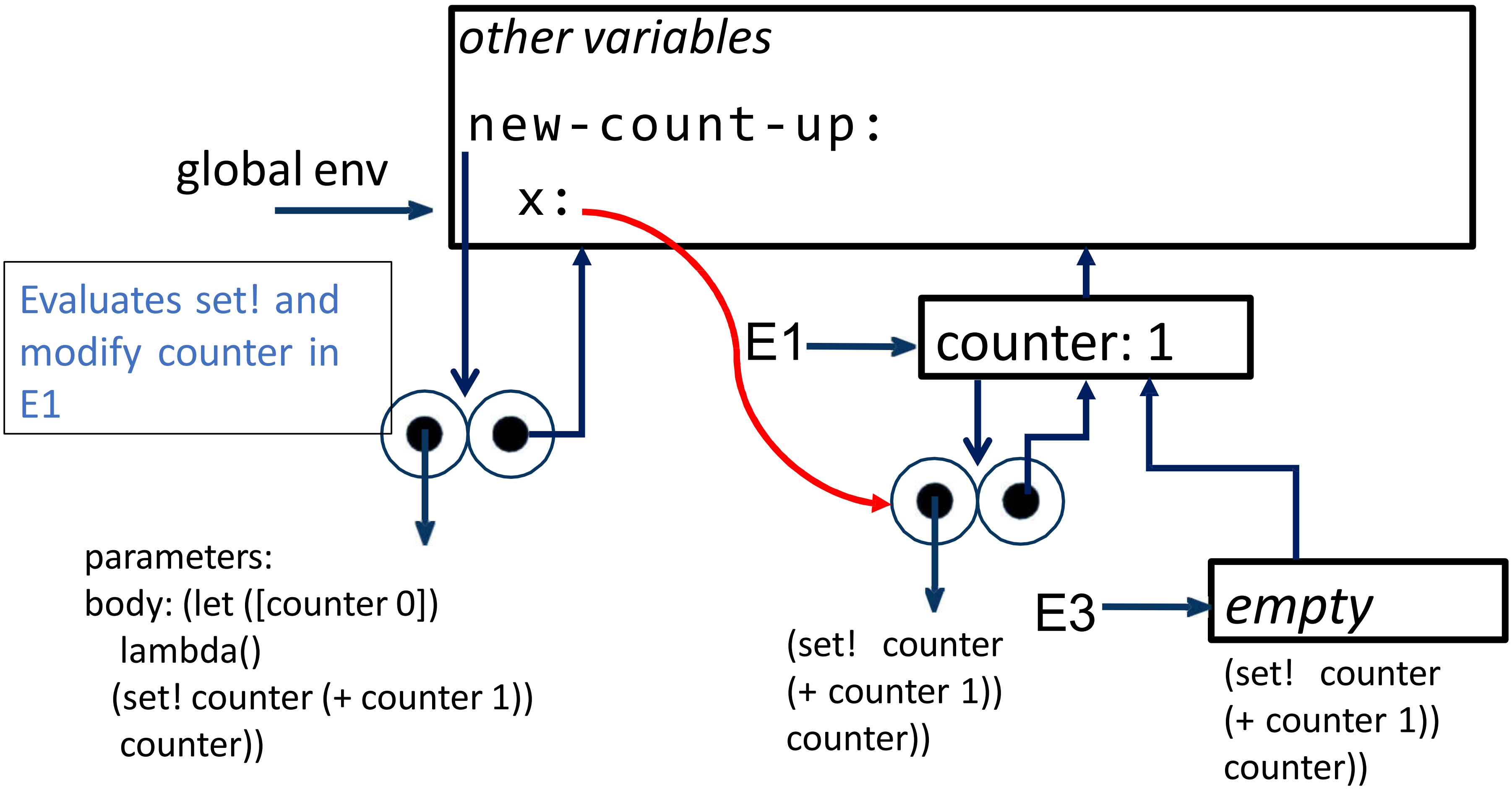
# What happens when (x) is called?

```
(x)
```



# What happens when (x) is called?

```
(x)
```

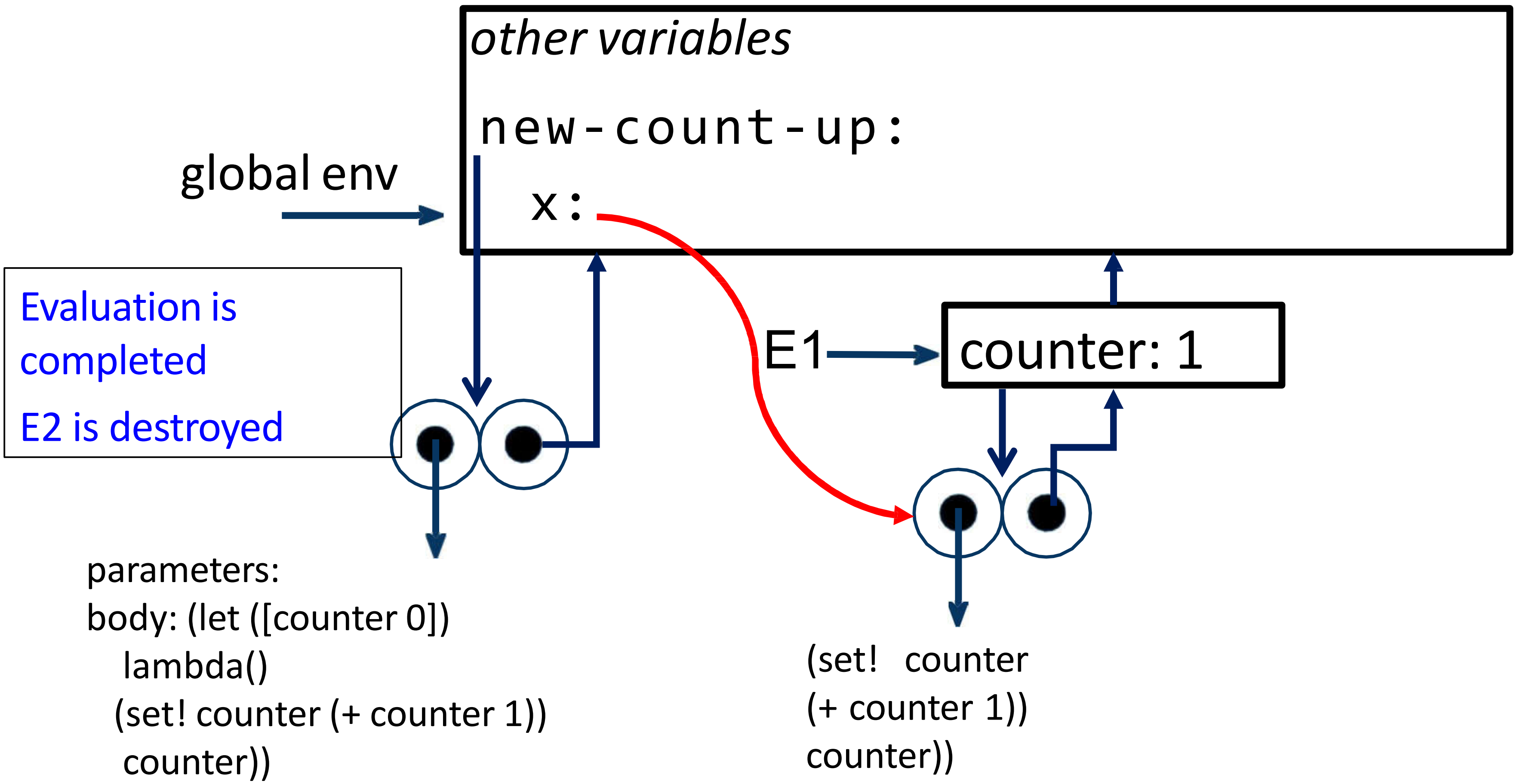





Carleton University

# What happens when (x) is called?

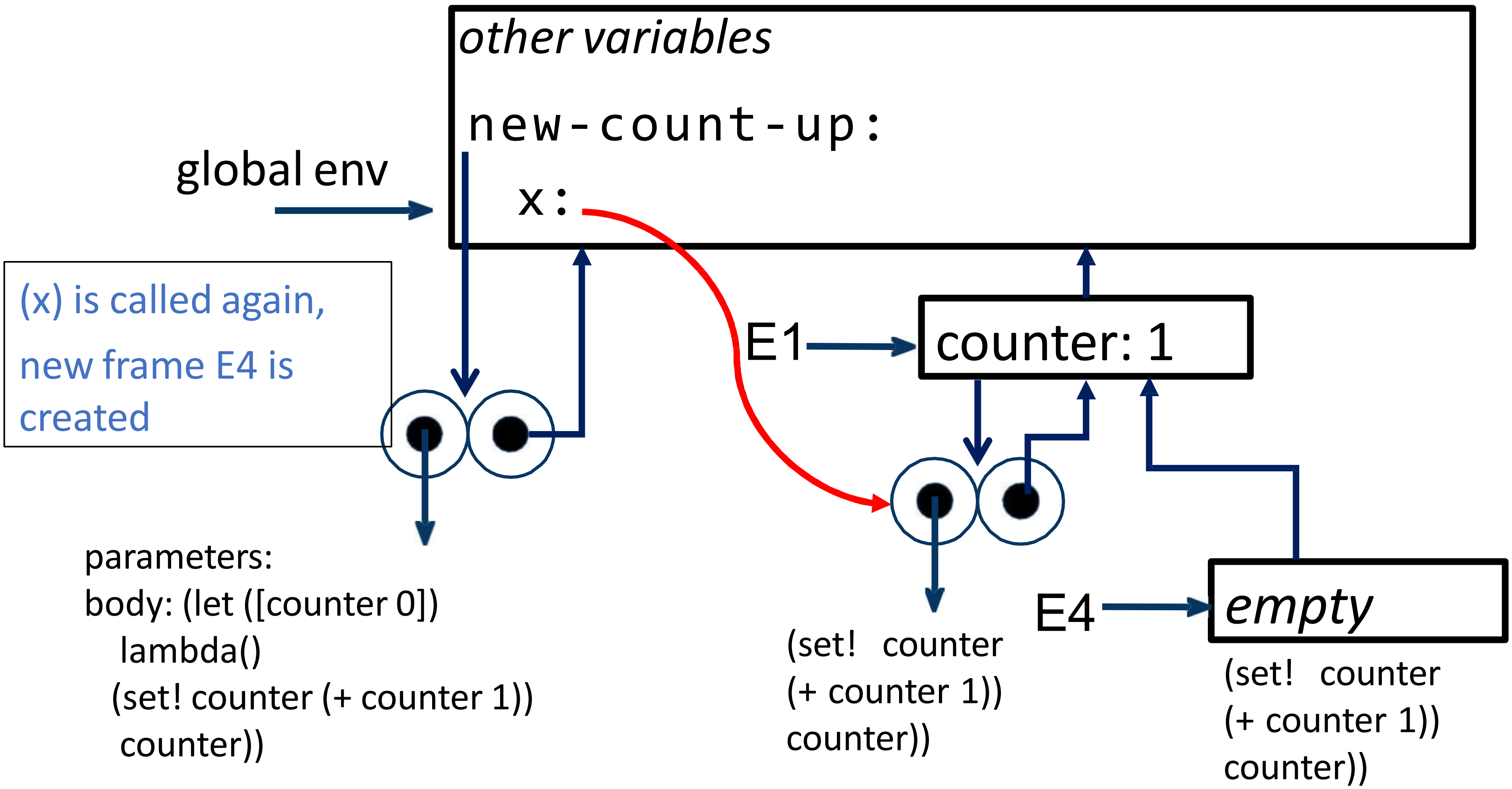
(x)




Carleton University

# What happens when (x) is called?

`(x) - 2nd time`

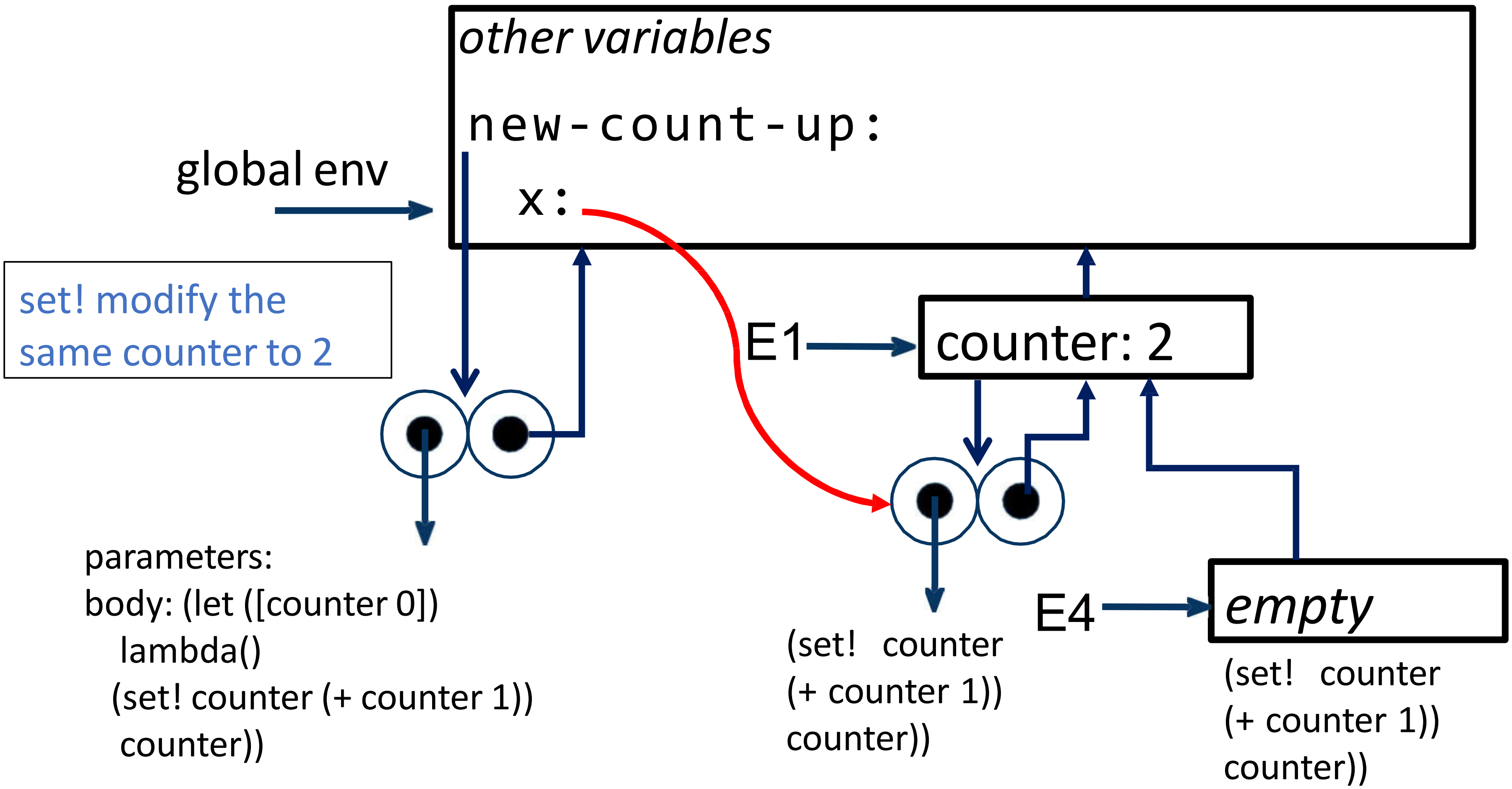





Carleton University

# What happens when (x) is called?

`(x) - 2`$^{nd}$ `time`



# SYSC 3101 Winter 2022
# The Environment Model of Evaluation (Part 1)

The slides are adapted form SYSC 3101 W19,D.L. Bailey, Department of Systems and Computer Engineering

Carleton University

# Substitution Model: Recap

- Steps:
  - Substitute the actual argument value(s) for the formal parameter(s) in the body of the function.
  - Evaluate the resulting expression.
- Example:

```
(define (add-one x) (+ x 1))
(add-one 10)
==> (+ 10 1)
==> 11
```

# Substitution Model: Recap

- Procedure `plus-one` does the same thing as `add-one`, but uses assignment

```
(define (plus-one var)
  (set! var (+ var 1))
  var)
```

```
> (plus-one 10)
```

```
11
```

# Substitution Model: Recap

- Try using the substitution model

```
(define (plus-one var)
  (set! var (+ var 1))
  var)

(plus-one 10)

==> (set! var (+ 10 1)) 10

==> (set! var 11) 10    ; returns 10???
```

- *Cannot use substitution model of evaluation with code that contains assignments*

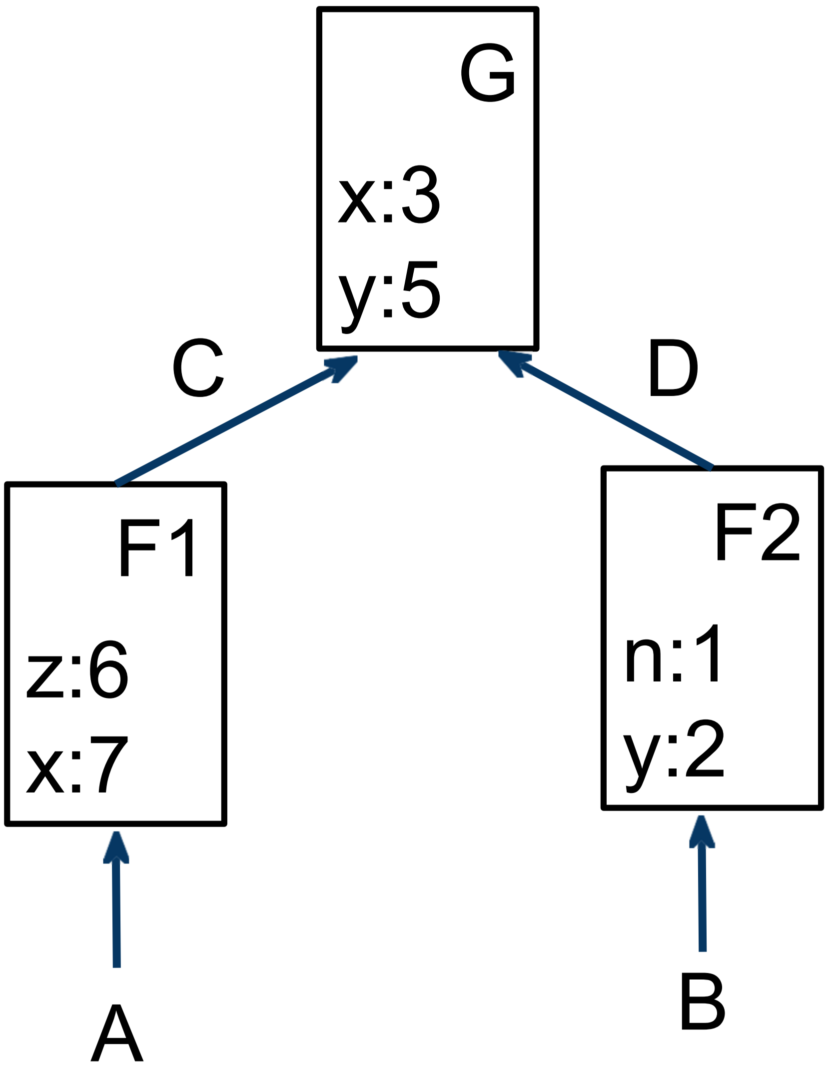
# Substitution Model: Recap

- Underlying assumption in the substitution model: variables (e.g., x) are just names for values
- Assignment implies that a variable is not just a name
    - *it's a place where a value can be stored, and we can replace the value stored there*
- We need a new model of evaluation...

# Environment Model

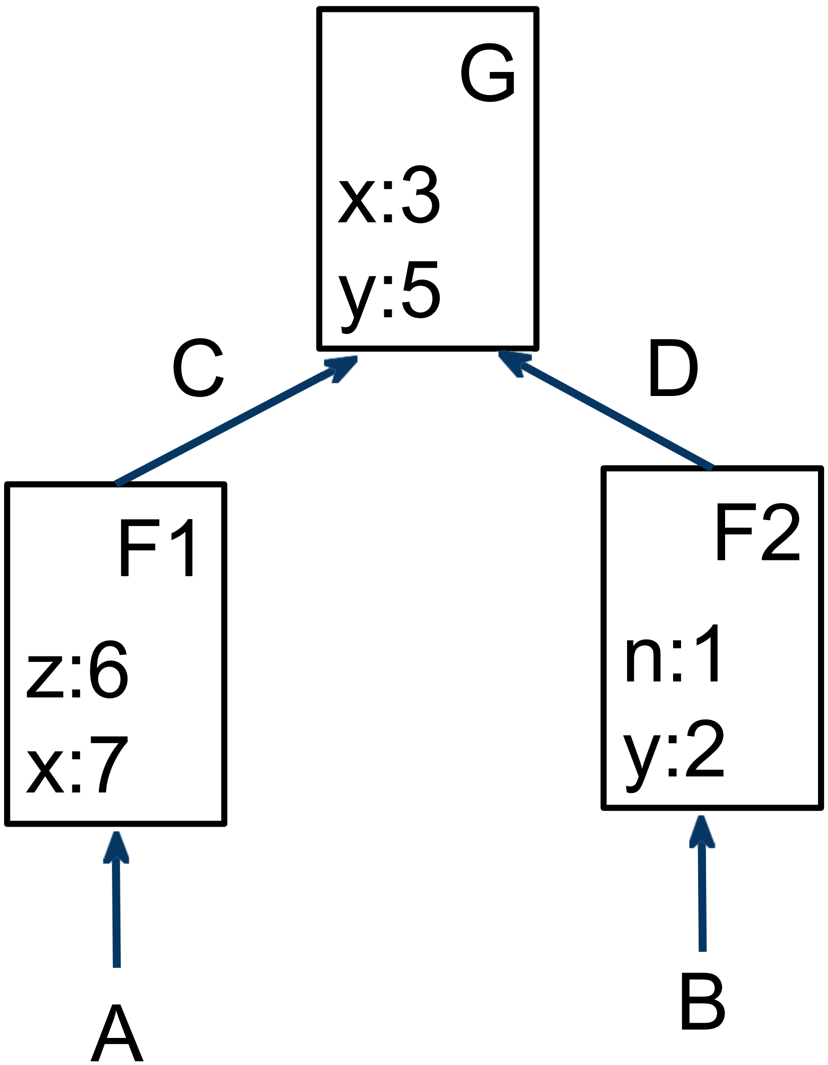
- An *environment* is a sequence of *frames*
- A frame is a table of *bindings* (name-value associations)
- Each frame has a pointer to its enclosing environment, except for a frame that is considered to be global
- Expressions are always evaluated with respect to an environment
  - look up a variable by name in an environment to find the associated value

# Example



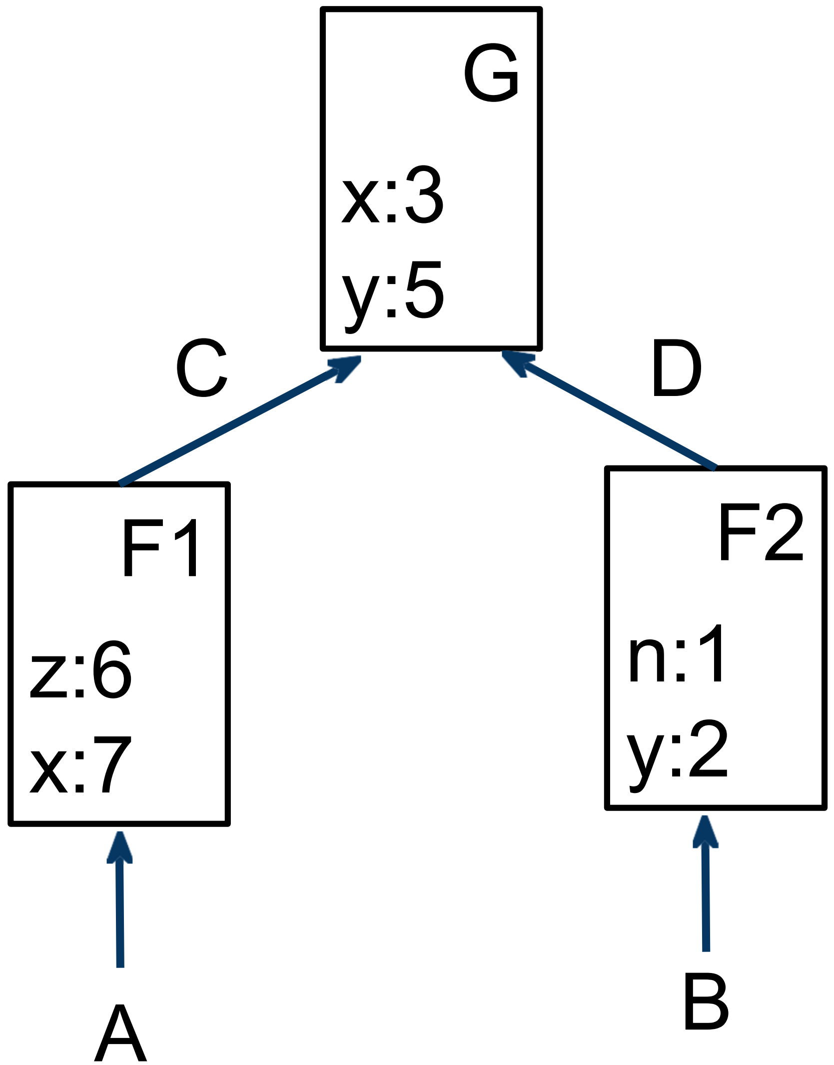

- There are 3 frames, labelled G, F1 and F2

# Example



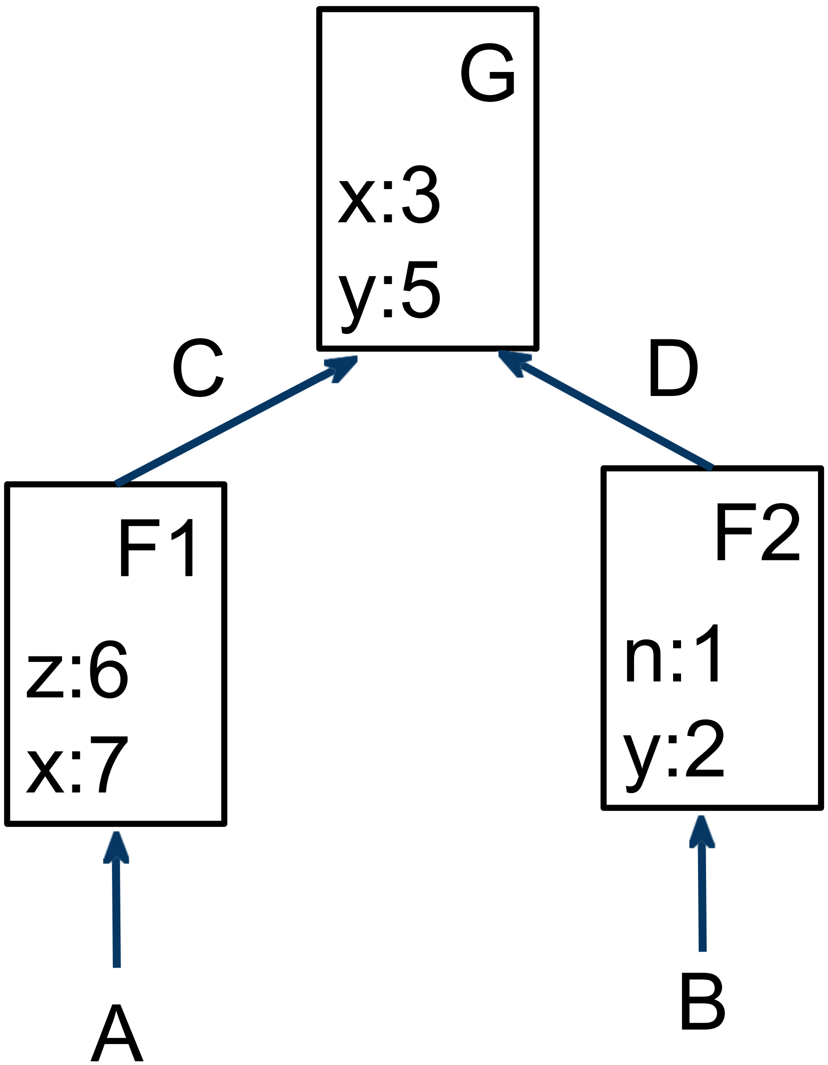

- A, B, C and D are pointers to environments
- C and D point to the same environment

# Example



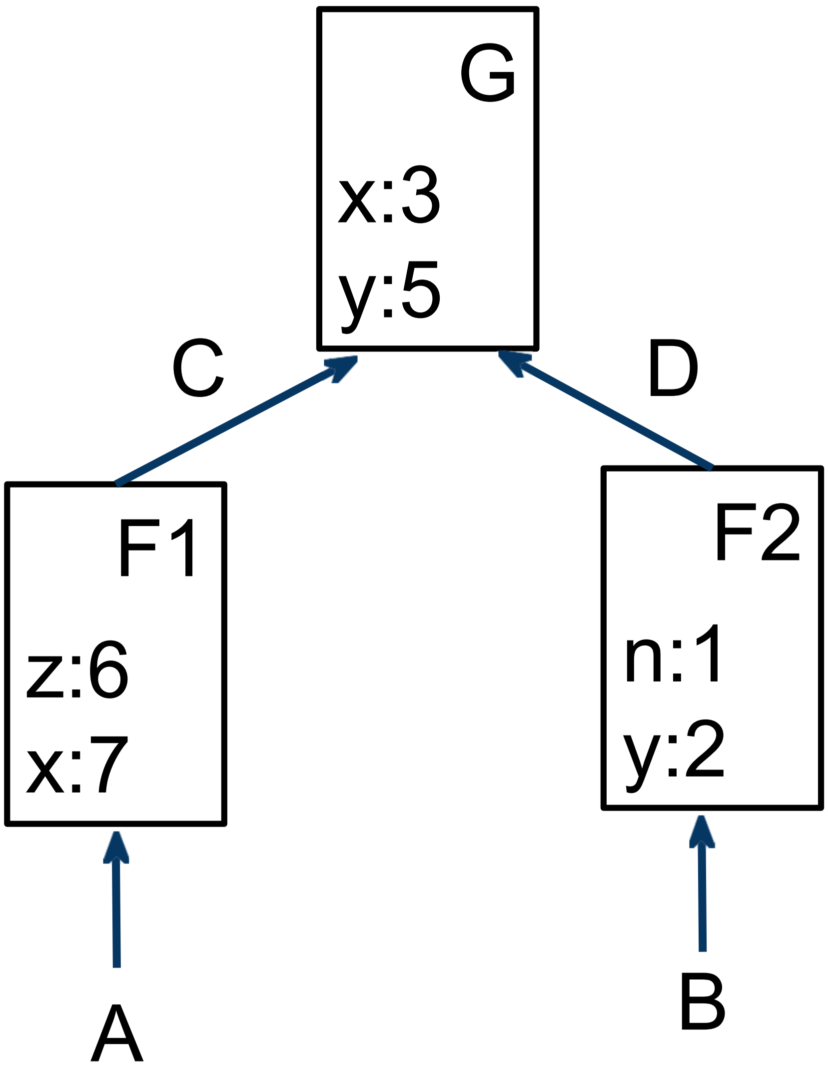

- Variables x and y are bound in frame G
- Variables x and z are bound in frame F1
- Variables n and y are bound in frame F2

# Example



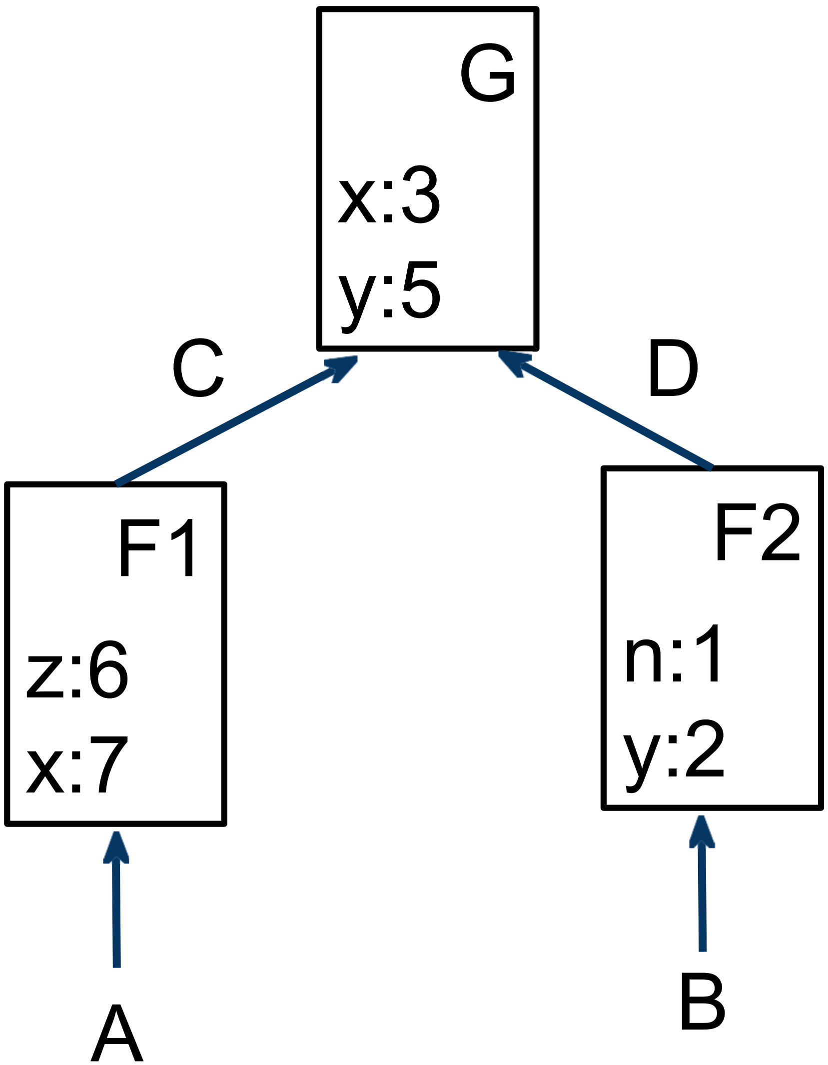

- The value of x with respect to environment D is 3
- How do we determine this?
  - there's a binding for x to 3 in frame G

# Example



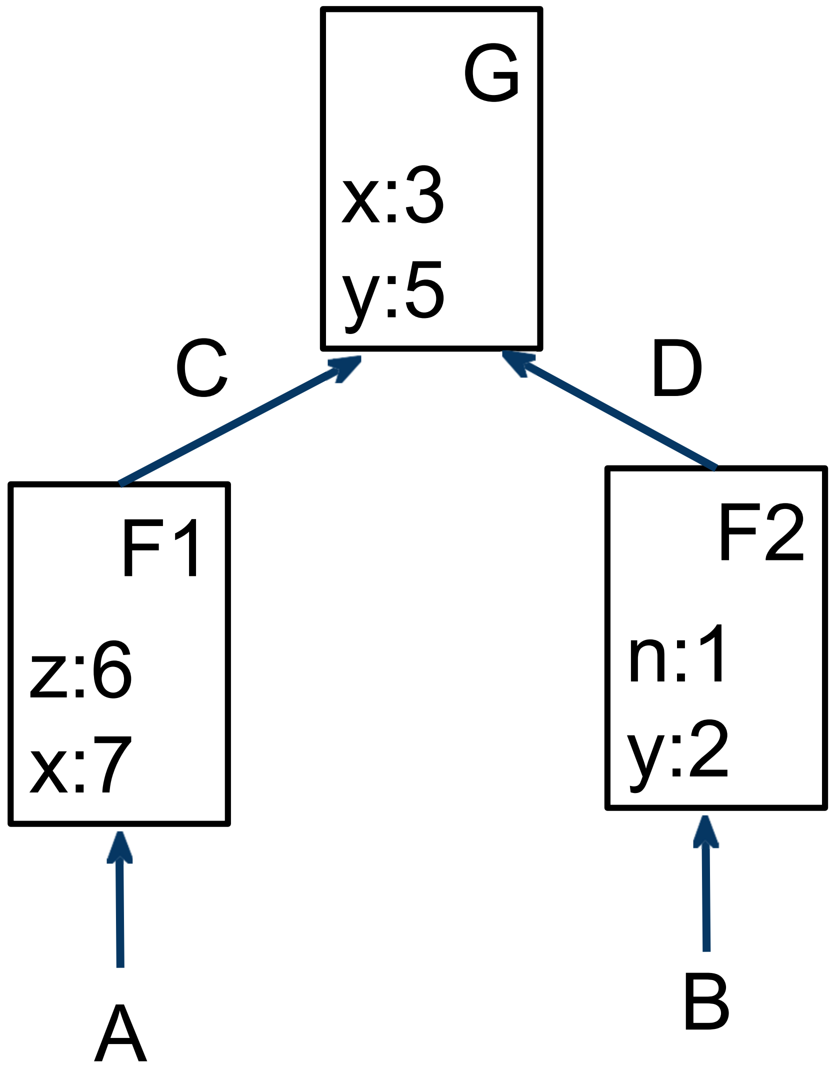

- The value of x with respect to environment B is 3
- How do we determine this?
  - see next slide...

# Example




- First frame in environment B (F2) doesn't have a binding for x
- F2's enclosing environment is D
- Follow the pointer to frame G
- G has a binding of x to 3

# Example




- The value of x with respect to environment A is 7
  - first frame in env. A (F1) has a binding for x to 7
  - with respect to environment A, the binding of x to 7 in F1 *shadows* the binding of x to 3 in G



Carleton University

# Questions

- How are frames & environments created?
- How are procedures represented in the environment model?
- What happens in an environment
  - when a procedure is evaluated?
  - when a procedure is applied?

# Environment Model and Procedures

- Reference: *SICP*, Section 3.2.1
- This procedure definition:

```
(define (square x)
  (* x x))
```

is syntactic sugar for:

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

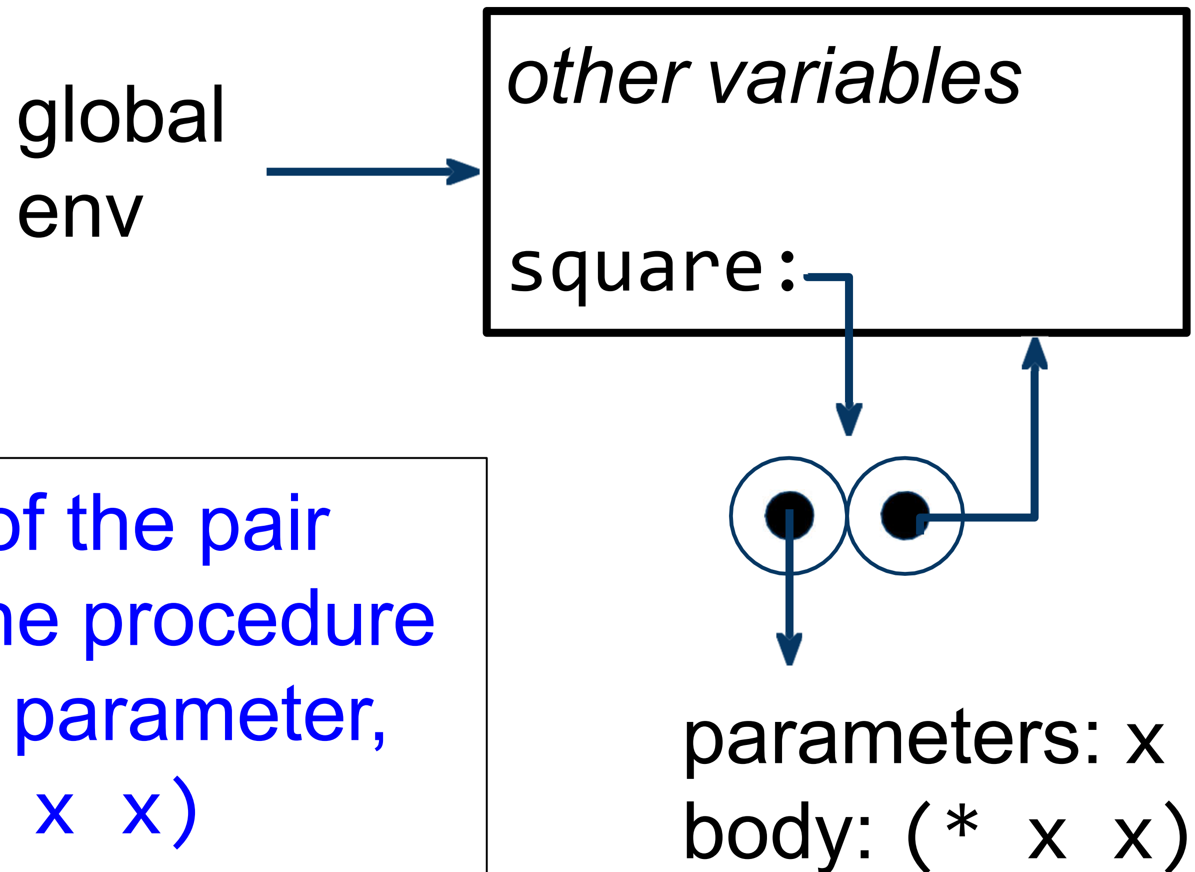
# Evaluating a Procedure

- When a procedure definition is evaluated, a ***procedure object* is created**
- A procedure object is modelled as a **pair**
    - the text of the **`lambda` expression** that was evaluated to create the procedure
    - a **pointer to the environment** in which the `lambda` expression was evaluated
- Example: assume the definition of `square` is evaluated in the global environment

# Evaluating a Procedure

```
(define (square x)
  (* x x))
```

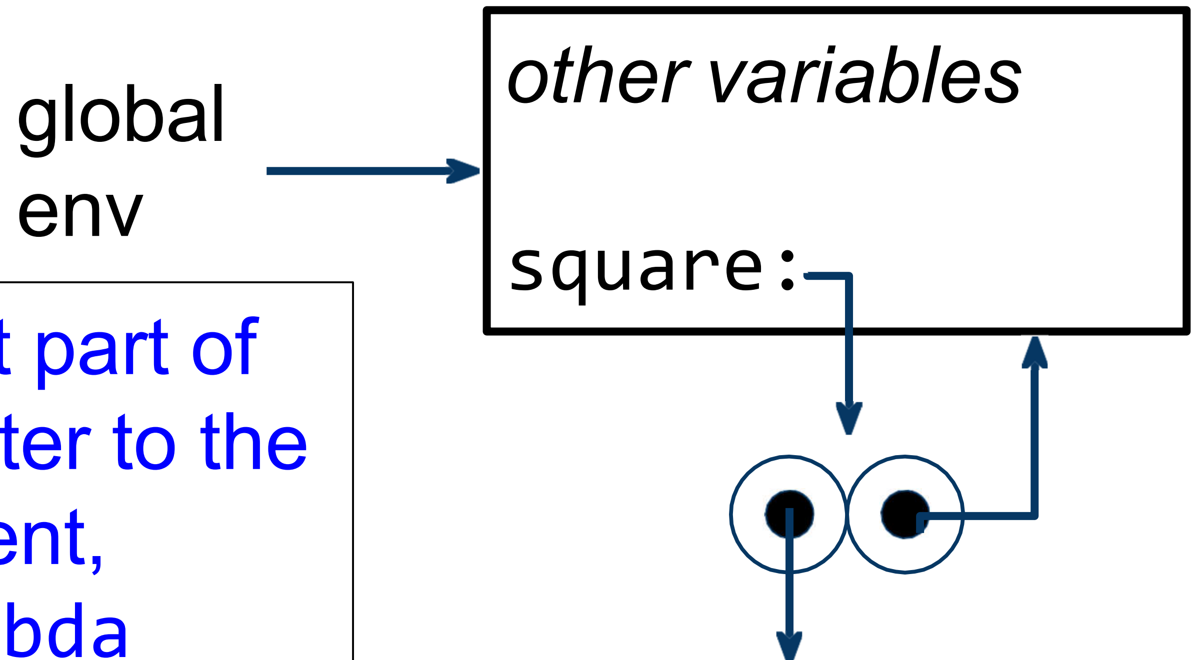
Diagrams follow style used in *SICP*




- The code part of the pair specifies that the procedure has one formal parameter, x, and body `(* x x)`

# Evaluating a Procedure

```
(define (square x)
  (* x x))
```



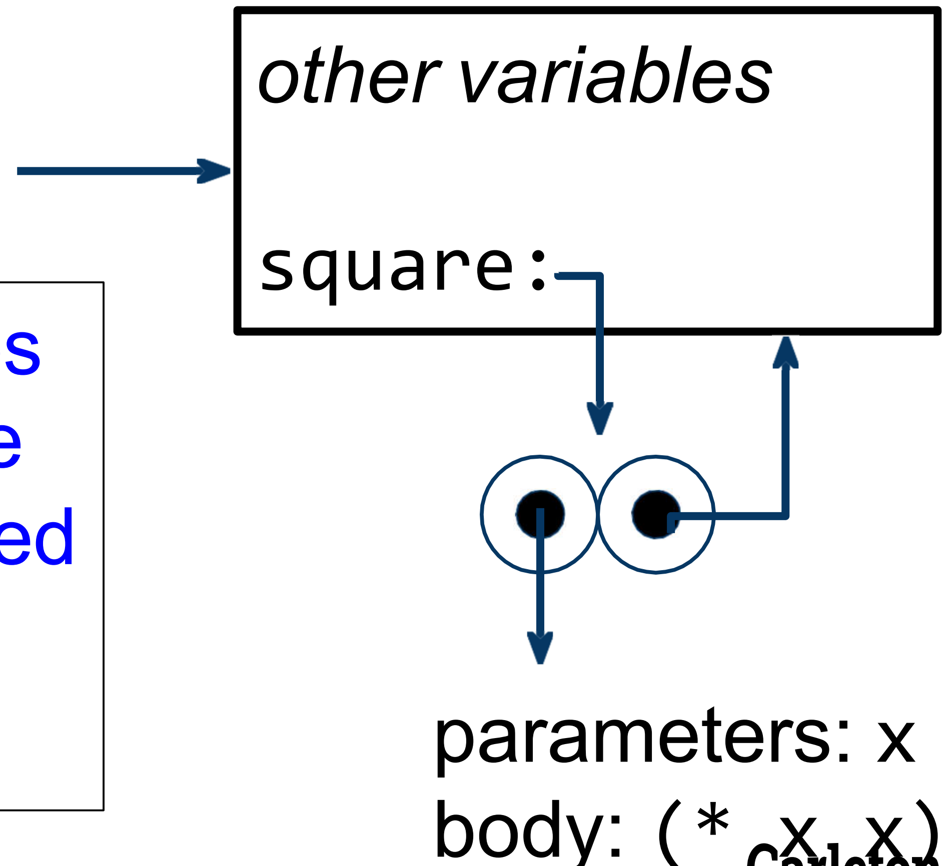

parameters: x
body: (* x x)

- The environment part of the pair is a pointer to the global environment, because the `lambda` expression was evaluated in that environment

Carleton University

# Evaluating a Procedure

```
(define (square x)
  (* x x))
```




- A binding that associates variable `square` with the procedure object is added to the global environment's frame

# Applying a Procedure

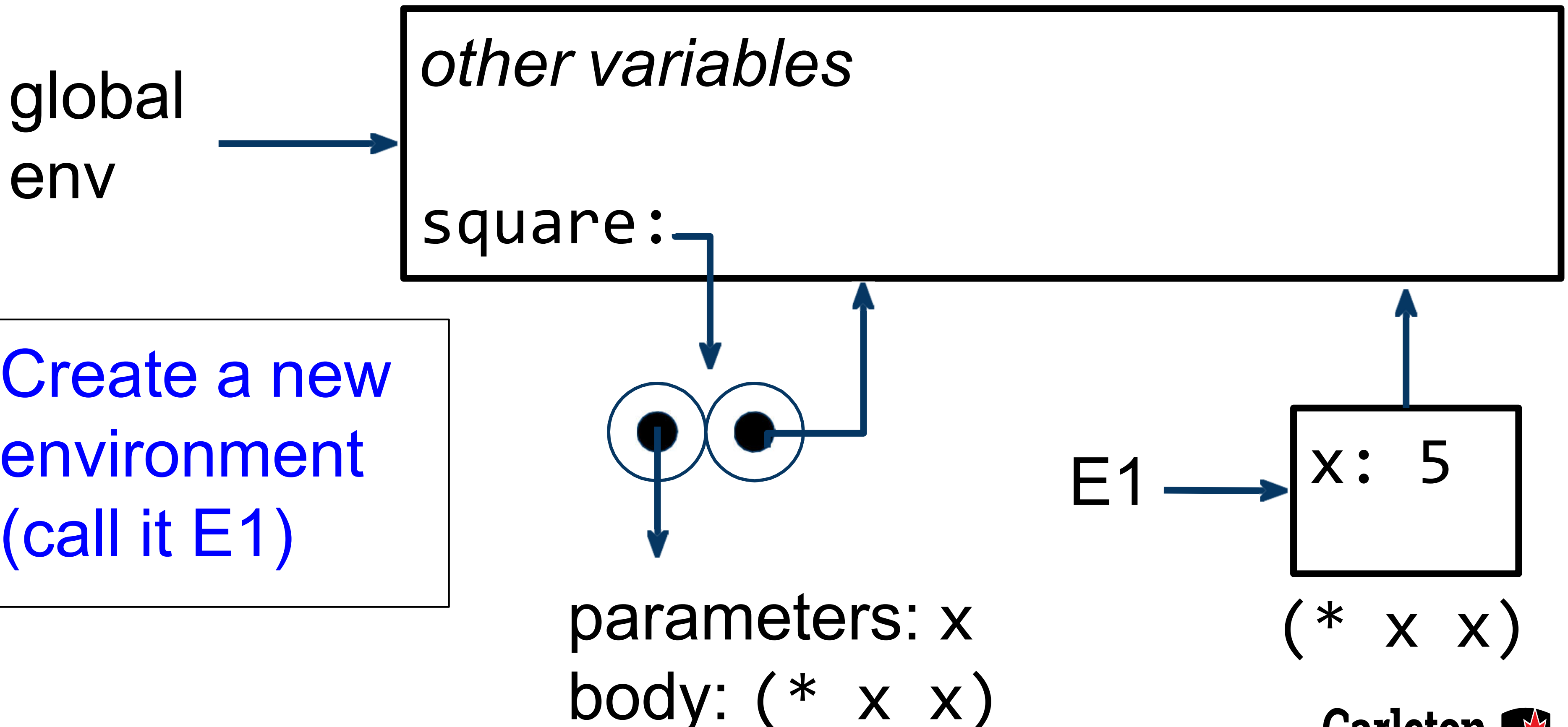
- To apply a procedure to arguments
- Step 1: create a new environment
  - create a new frame with the procedure's formal parameters bound to the values of the arguments
  - the enclosing environment of the new frame is the environment pointed to by the procedure object

# Applying a Procedure

- Step 2: evaluate the procedure body with respect to the new environment
- Example: evaluate `(square 5)`

# Applying a Procedure

```
(square 5)
```

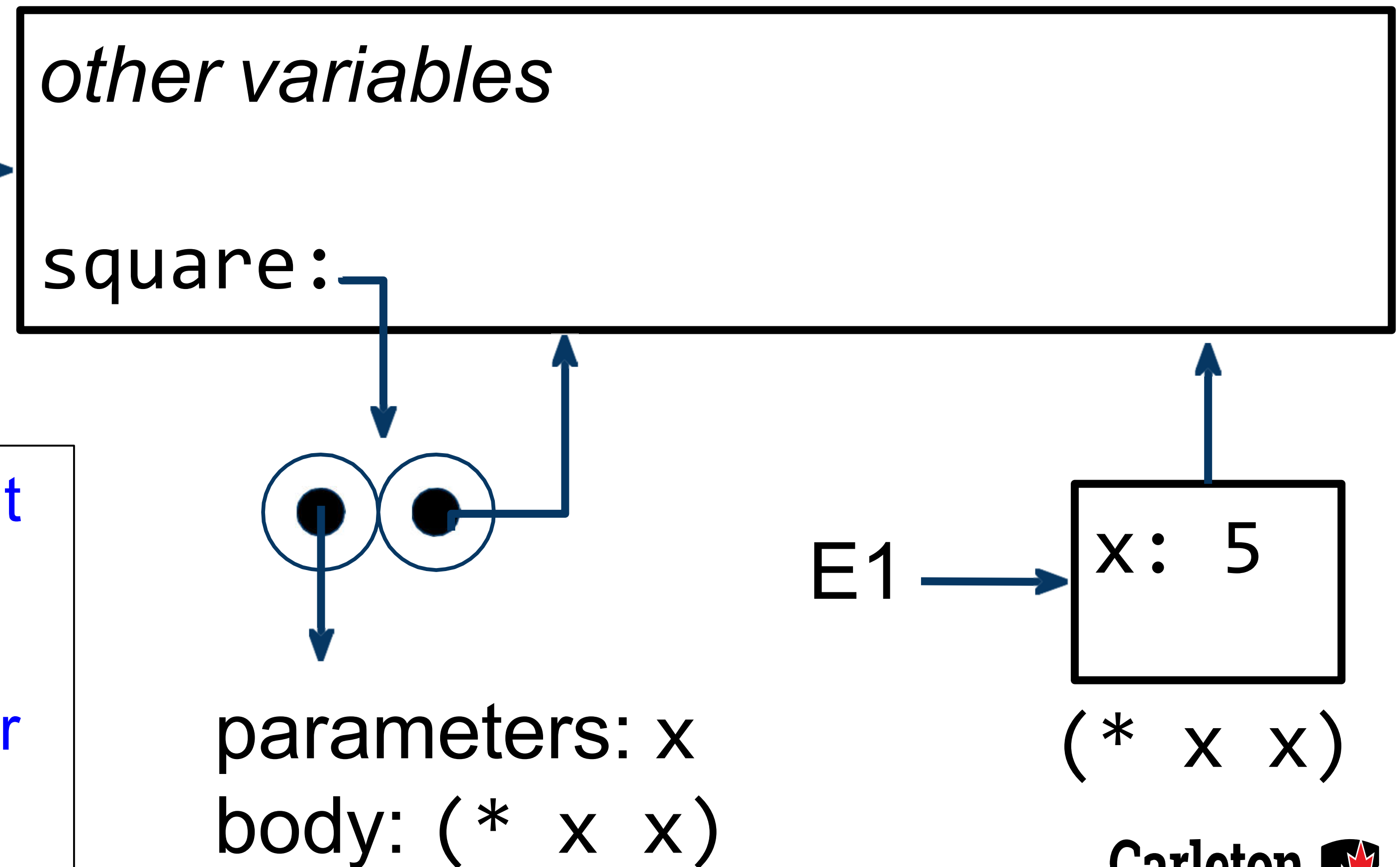



- Create a new environment (call it E1)

# Applying a Procedure

```
(square 5)
```

global env

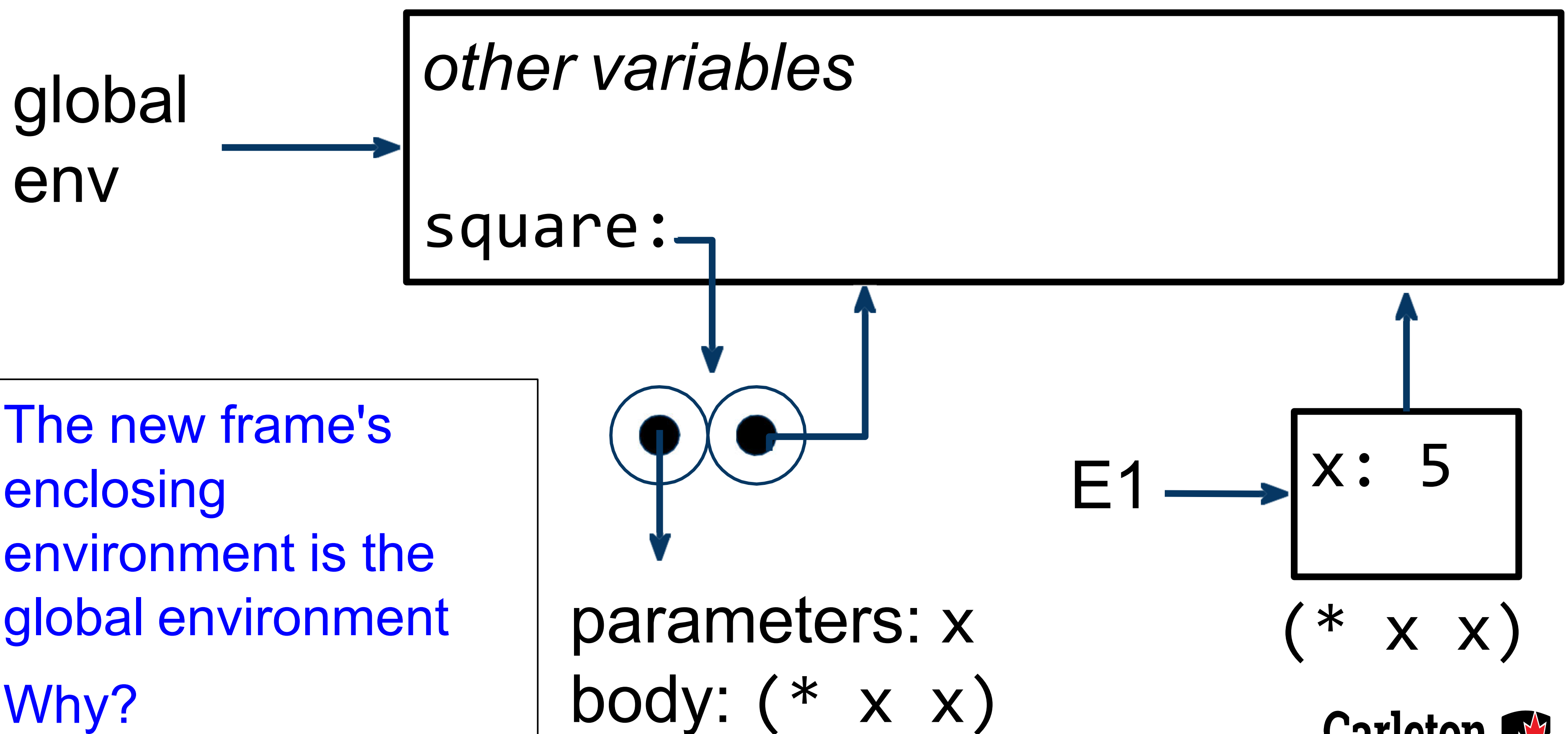



- This environment begins in a new frame in which formal parameter x is bound to 5

Carleton University

# Applying a Procedure

```
(square 5)
```



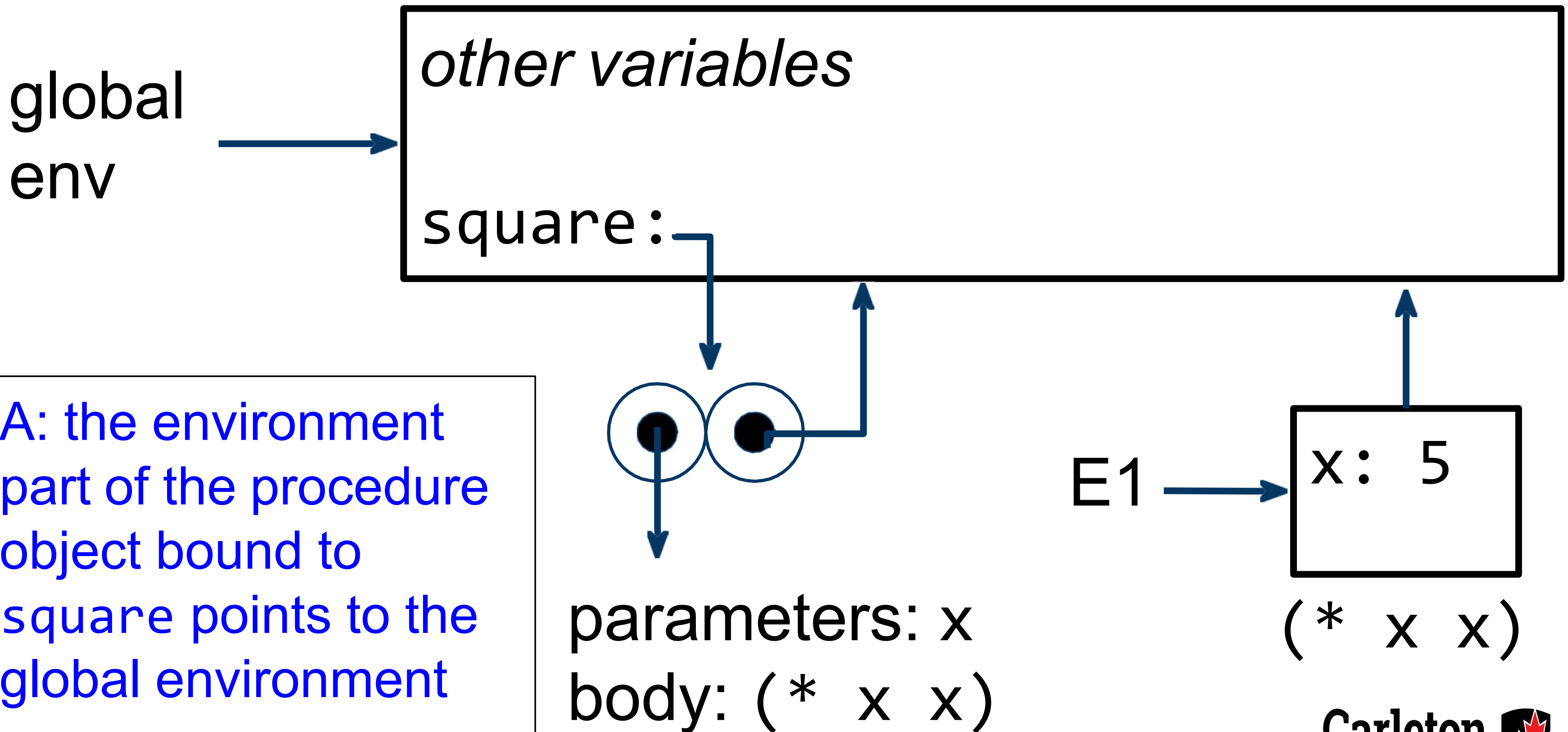

- The new frame's enclosing environment is the global environment
- Why?



Carleton University

# Applying a Procedure

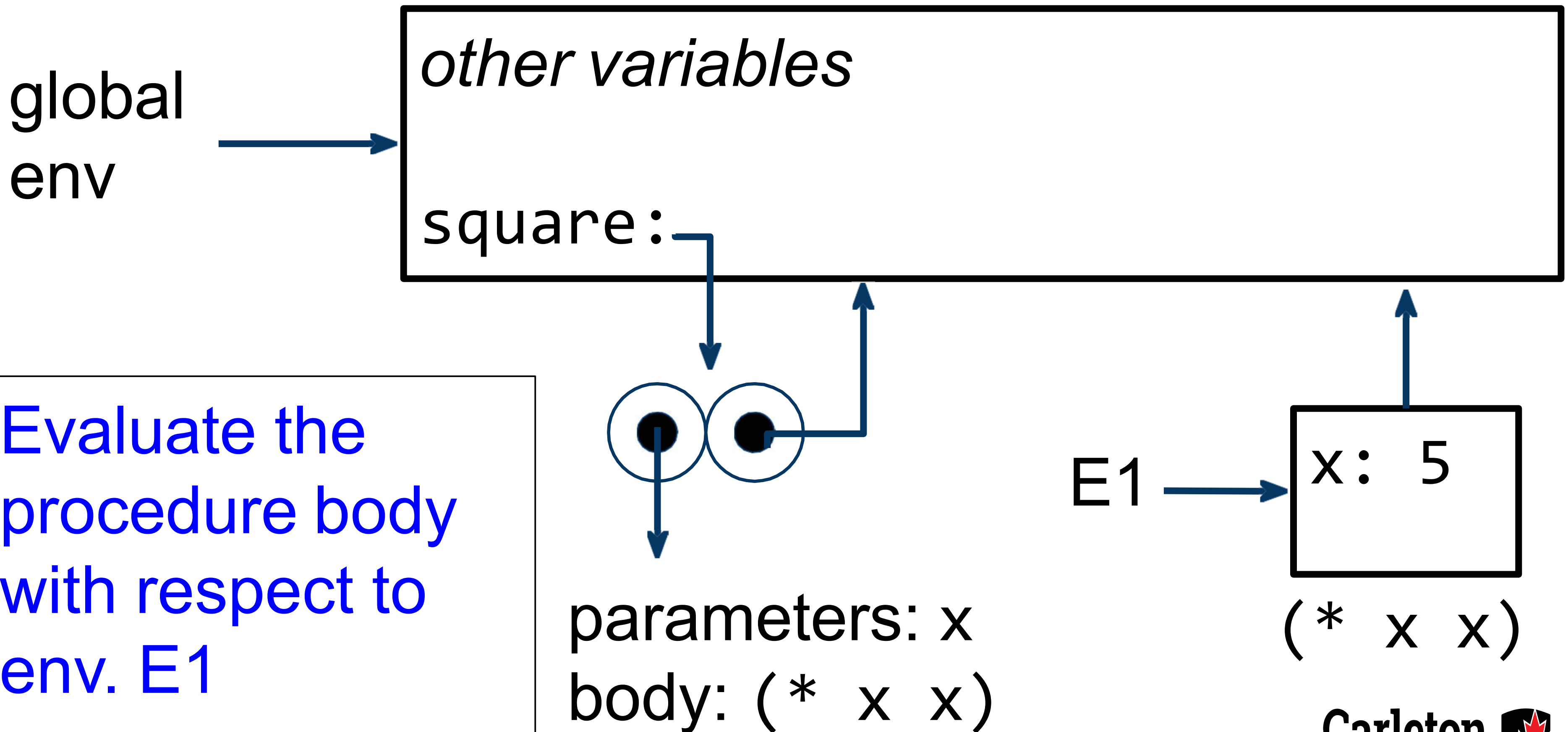
`(square 5)`




- A: the environment part of the procedure object bound to `square` points to the global environment

# Applying a Procedure

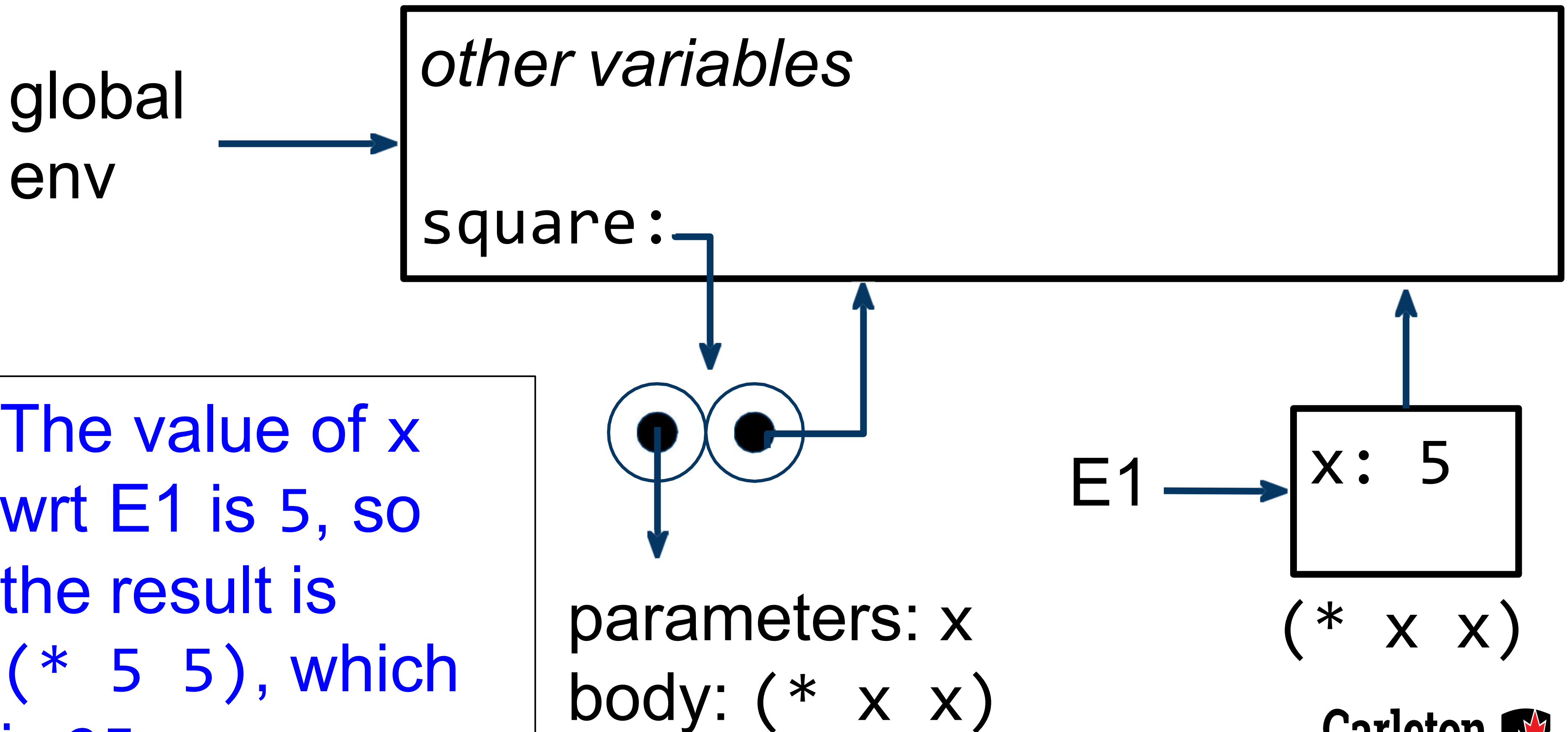
`(square 5)`




- Evaluate the procedure body with respect to env. E1

# Applying a Procedure

```
(square 5)
```




- The value of x wrt E1 is 5, so the result is (* 5 5), which is 25

Carleton University

# Env. Model: Summary of Rules (1)

- Environment model specifies what happens when a compound procedure is ***created***
    - evaluate a `lambda` expression relative to an environment
    - the resulting procedure object is a pair consisting of the `lambda` expression text and a pointer to the environment in which the procedure was created

# Env. Model: Summary of Rules (2)

- Environment model specifies what happens when a compound procedure is ***applied*** to arguments
  - create a frame with the formal parameters bound to the arguments
  - the new frame has, as its enclosing environment, the environment part of the procedure object being applied
  - evaluate the body of the procedure in the context of the new environment

```
#lang racket

;; Last edit: Mar. 28, 2017, dlb


(define (racket-1)
  (newline)
  (display "Racket-1: ")
  (flush-output)
  (print (eval-1 (read)))
  (newline)
  (racket-1)
  )




;Contains new code for assignment
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (eval-1 exp)
  ; display expressions added. -- dlb
  (display "Executing eval-1, exp = ")
  (displayln exp)

  (cond ((constant? exp)
         (displayln "constant? is true")
         exp)

        ((symbol? exp)
         (displayln "symbol? is true")
         (eval exp))  ; use underlying Racket's EVAL

        ((quote-exp? exp) (quote-helper exp)) ;checks if too many
args


        ((if-exp? exp)
         (displayln "if-exp? is true")
         (if (eval-1 (cadr exp))   ; use underlying Racket's IF
             (eval-1 (caddr exp))
             (eval-1 (cadddr exp))))

        ((lambda-exp? exp)
         (displayln "lambda-exp? is true")
         exp)
        ((map-exp? exp)  ;part of ex6
         (map-1 (eval(cadr exp)) (eval (caddr exp))))



        ((and-exp? exp)   ;part of ex7
```

```
         (cond
           ((=(length exp)1) '#t)
           ((=(length exp) 2) (eval-1 (cadr exp)))
           ((equal? (cadr exp) '#t)(eval-1 (caddr exp)))
           ((equal? (cadr exp) '#f) '#f)
           ((equal? (eval-1(cadr exp)) '#f)'#f)
           ((equal? (eval-1(cadr exp)) '#t)(eval-1(caddr exp)))
           ))


        ((pair? exp)
         (displayln "pair? is true")
         (apply-1 (eval-1 (car exp))      ; eval the operator
                  (map eval-1 (cdr exp))))

        (else (error "bad expr: " exp))))



 (define (map-1 proc arg) ;map-1 so that racket doesnt use map ex6
   (map proc arg))


 (define (quote-helper arg) ;we can only have one argument for quote
(ex5)
   (if (> (length arg) 2) (error "Too many arguments" arg) (cadr
arg)))





;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Code that came with the file
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(define (apply-1 proc args)
  ; display expressions added. -- dlb
  (display "Executing apply-1, proc = ")
  (display proc)
  (display " args = ")
  (displayln args)

  (cond ((procedure? proc)  ; use underlying Racket's APPLY
         (displayln "procedure? is true")
         (apply proc args))

        ((lambda-exp? proc)
         (displayln "lambda-exp? is true")
```

```
         (let ([s (substitute (caddr proc)   ; the body
                              (cadr proc)     ; the formal parameters
                              args            ; the actual arguments
                              '())])          ; bound-vars, see below
           (begin
             (display "substitute returned ")
             (displayln s)
             (eval-1 s))))

        (else (error "bad proc: " proc))))





;; Some trivial helper procedures:

(define (constant? exp)
  (or (number? exp) (boolean? exp) (string? exp) (procedure? exp)))

(define (exp-checker type)
  (lambda (exp) (and (pair? exp) (eq? (car exp) type))))


;Contains new code for assignment
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define quote-exp? (exp-checker 'quote))
(define if-exp? (exp-checker 'if))
(define lambda-exp? (exp-checker 'lambda))
(define map-exp? (exp-checker 'map-1));helper for map-1
(define and-exp? (exp-checker 'and)); helper for and
;;;;;;;;;;;;;;;;;;;;;


(define (substitute exp params args bound)
  ; display expressions added. -- dlb
  (display "Executing substitute, exp = ")
  (display exp)
  (display " params = ")
  (display params)
  (display " args = ")
  (display args)
  (display " bound = ")
  (displayln bound)

  (cond ((constant? exp) exp)
        ((symbol? exp)
         (if (memq exp bound)
             exp
             (lookup exp params args)))
        ((quote-exp? exp) exp)
```

```
        ((lambda-exp? exp)
         (list 'lambda
               (cadr exp)
               (substitute (caddr exp) params args (append bound
(cadr exp)))))
        (else (map (lambda (subexp) (substitute subexp params args
bound))
                   exp))))

(define (lookup name params args)
  (cond ((null? params) name)
        ((eq? name (car params)) (maybe-quote (car args)))
        (else (lookup name (cdr params) (cdr args)))))

(define (maybe-quote value)
  (cond ((lambda-exp? value) value)
        ((constant? value) value)
        ((procedure? value) value)  ; real Racket primitive procedure
        (else (list 'quote value))))
```