CSCI 275: Programming Abstractions

Lecture 07: Function Design - Part 2

Spring 2025

Stephen Checkoway, Oberlin College
Slides gratefully borrowed from Molly Q Feldman

Questions? Comments?

Goals for Today's Class

- Local variables: `let`
- Environments: how do we store bindings?
- [If time] Tail Recursion, or how to be efficient

Let

Storing Local Information

```
(let ([id1 s-exp1] [id2 s-exp2]…) body)
```

`let` enables us to create some new bindings that are visible only inside `body`

```
(let ([x 37]            ; binds x to 37
      [y (foo 42)])     ; binds y to the result of (foo 42)
  (if (< x y)
      (bar x)
      (bar y)))
```

`x` and `y` are only bound inside the body of the let expression

That is, the *scope* of the identifiers bound by `let` is `body`

What happens when you want a binding in terms of an existing one?

When writing programs, it's not uncommon to define some local variables in terms of other local variables

```
(define (all-larger? lst)
  (let ([head (first lst)]
        [streamlined
          (filter (λ (x) (> x head)) (rest lst))])
      (pair? streamlined)))
```

Given a list, is everything after the first element larger than the first element?

This doesn't work; we can't use `head` in the definition of `streamlined`

The Fix? Sequential let

`(let* ([id1 s-exp1] [id2 s-exp2]…) body)`

Later s-exps can use earlier ids!

Example:

```
(let* ([x 5]
       [y (foo x)]
       [z (+ x y)])
  (bar z y))
```

Environments

How we know what `x` means?

Recall that when Racket evaluates a variable, the result is the value that the variable is bound to

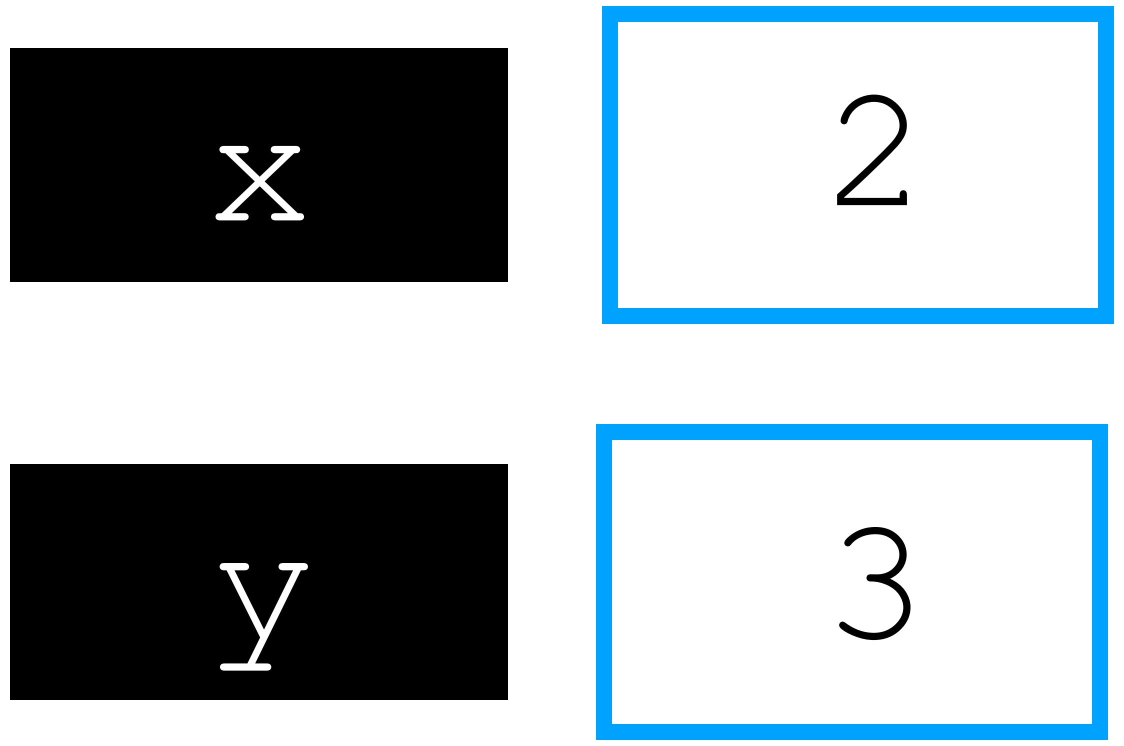
If we have `(define x 10)`, then evaluating `x` gives us the value `10`

If we have `(define (foo x) (- x y))`, then evaluating `foo` gives us the procedure `(lambda (x) (- x y))`, along with a way to get the value of `y` (which is hopefully defined!)

Racket needs a way to look up values that correspond to variables: an **environment**

Environments: Examples

```
(let ([x 2]
      [y 3])
  (+ x y))
```



When we execute the following, what is the result?

```
(let ([x 2]
      [y 3])
  (let ([x 4])
    (+ x y)))
```

A. 6

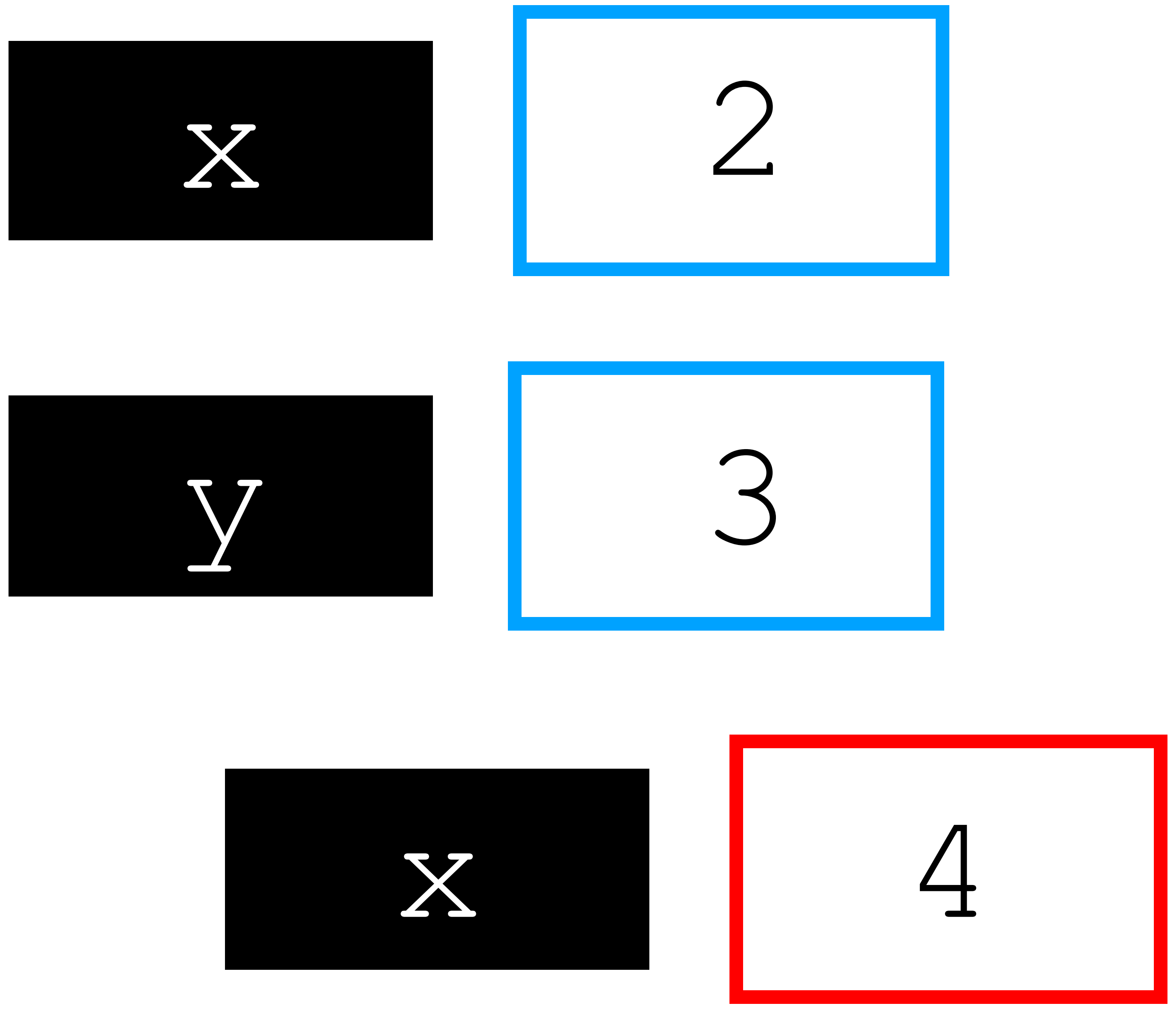
B. 9

C. 7

D. Something else

Environments: Examples

```
(let ([x 2]
      [y 3])
  (let ([x 4])
    (+ x y)))
```



When we execute the following, what is the result?

```
(let ([x 2]
      [y 3])
  (let ([f (lambda (x) (+ x y))])
    (f 5)))
```

A. 8

B. 7

C. 5

D. Something else

DrRacket shows variable bindings

Mouse over an identifier in DrRacket

```
(let ([x 2] [y 3])
  (let ([f (lambda (x) (+ x y))])
    (f 5)))
```

```
(let ([x 2] [y 3])
  (let ([f (lambda (x) (+ x y))])
    (f 5)))
```

Environment Operations

Two basic operations on environments:

1. **Look something up**
   - What is the binding of `x` right now?

2. **Extend an existing environment with new bindings**
   - Creates a new environment containing both the existing and the new bindings

Look Up in Environments

Look up the value to which a symbol is bound:

```
(let ([x 3])
  (let ([x 4])
    (+ x 5)))
```

should return 9 since the innermost binding of x is 4

Extending Environments: Let

Consider

```
(let ([x (+ 3 4)]
      [y 5]
      [z (foo 8)])
  body)
```

We have three symbols `x`, `y`, and `z` and three values, 7, 5, and whatever the result of `(foo 8)` is, let's say it's 12

If `E` is the environment of the whole `let` expression, then the body should be evaluated in the environment

`E[x ↦ 7, y ↦ 5, z ↦ 12]`

Closures

The expression of `(lambda parameters body…)` evaluates to a *closure* consisting of

- The parameter list (a list of identifiers)
- The body as un-evaluated expressions (often just one expression)
- The environment (the mapping of identifiers to values) **at the time the lambda expression is evaluated** not the time the closure is called

Environments & Procedure Calls

```
(define A 10)
(define add-a
  (lambda (x)
    (+ x A)))
```

Calling the closure extends the closure's environment with its parameters bound to the arguments

```
(add-a 20)
```

When called, the closure's body is evaluated with this new environment

Environment of the closure

A	10

Keep it around! Part of what the closure contains!

Environment of the call

A	10
x	20

Even More Let

A realistic example

Let's write a procedure `(split-by pred lst)` that splits lst into two lists, the first contains all of the elements that match pred, the second contains all the elements that do not match pred

```
(split-by even? (range 10)) => '((0 2 4 6 8) (1 3 5 7 9))

(split-by (lambda (x) (< x 3)) (range 5)) =>
               '((0 1 2) (3 4))
```

Recursion

Often, we're going to want to define a recursive procedure in a `let`. For example,

```
(define (count-bigger-than-first lst)
  (let* ([head (first lst)]
         [count (λ (lst)
                  (cond [(empty? lst) 0]
                        [(> (first lst) head)
                         (+ 1 (count (rest lst)))]
                        [else (count (rest lst))]))])
    (count (rest lst))))
```

Unfortunately, we can't use `count` in the definition of `count`

Recursive let

`(letrec ([id1 s-exp1] [id2 s-exp2]…) body)`

All of the s-exps can refer to *all* of the ids

This is used to make recursive procedures

```
(define (count-bigger-than-first lst)
  (letrec ([head (first lst)]
           [count (λ (lst)
                    (cond [(empty? lst) 0]
                          [(> (first lst) head)
                           (+ 1 (count (rest lst)))]
                          [else (count (rest lst))]))])
    (count (rest lst))))
```

Can’t we just always use letrec then?

Nope, a subtle point: the values of the identifiers we're binding can't be used in the bindings

Invalid (the value of `x` is used to define `y`)

```
(letrec ([x 1]
          [y (+ x 1)])
  y)
```

Valid (the *value* of `x` isn't used to define `y`, it's only used when `y` is *called*)

```
(letrec ([x 1]
          [y (lambda () (+ x 1))])
  (y))
```

Next Up

HW2 due at 11:59pm Friday – first commit due **tonight**