

Lexical Scope and Function Closures

adapted from materials by Dan Grossman at the University of Washington

Free variables

Variables used but not bound within function bodies.

```
(define x 1)
(define f (lambda (y) (+ x y)))
(define z
  (let ([x 2]
        [y 3])
    (f (+ x y))))
```

x is a free variable in the definition of f.

Big question:

What is the value of **x** when we evaluate the body of `(lambda (y) (+ x y))` here?

Lexical Scope

Function bodies can use any binding in scope *where the function was defined.* (not where it was called)

HUGELY important concept.

251 considers:

- Semantics (what?)
- Clarity/usability (why?)
- Implementation (how?)

Visualize in DrRacket, draw environments.

Example

Demonstrates lexical scope without higher-order functions:

defines a function that, when called, evaluates body `(+ x y)` in an environment where **x** is bound to 1 and **y** is bound to the argument

```
(define x 1)
(define f (lambda (y) (+ x y)))
(define z
  (let ([x 2]
        [y 3])
    (f (+ x y))))
```

1. Looks up **f** in current environment, finding this.
2. Evaluates `(+ x y)` in **current** environment, producing 5.
3. Calls the function with argument 5:
 - Evaluates the body in the **old** environment, producing 6.

Closures

revising our definition of functions

A **function definition expression** evaluates to a **function closure** value.

A **function closure** has **two parts**:

- **code** of function
- **environment** where the function was defined

Not a cons cell.
Cannot access pieces.

A **function call expression**:

- Evaluates the code of a function closure
- In the environment of the function closure

Example

Demonstrates lexical scope without higher-order functions:

```
(define x 1)
(define f (lambda (y) (+ x y)))
(define z
  (let ([x 2]
        [y 3])
    (f (+ x y))))
```

Creates a closure and binds **f** to it:
Code: `(lambda (y) (+ x y))`
Environment: `f → this closure, x → 1`

1. Looks up **f** in current environment, finding this closure.
2. Evaluates `(+ x y)` in **current** environment, producing 5.
3. Evaluates the closure's function body `(+ x y)` in the closure's environment (`f → the closure, x → 1`), extended with `y → 5`, producing 6.

The Rule: Lexical Scope

A **function body** is evaluated in the **environment where the function was defined** (created), extended with bindings for the arguments.

Next:

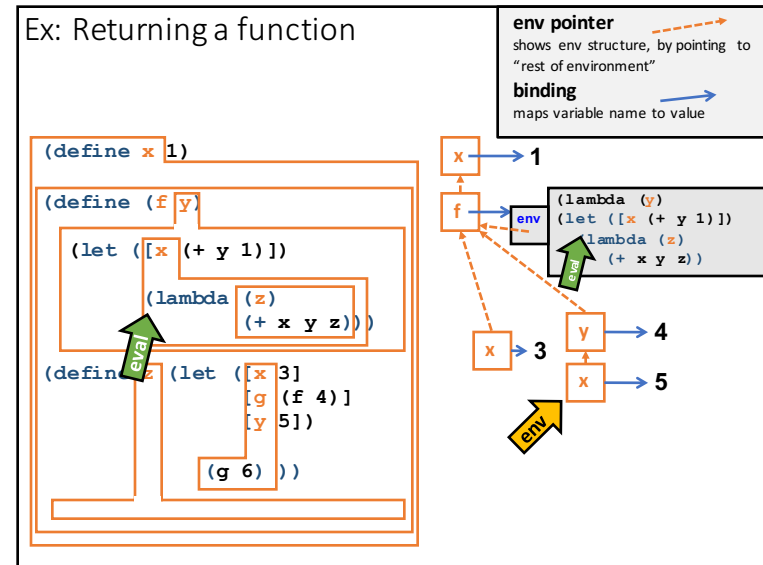
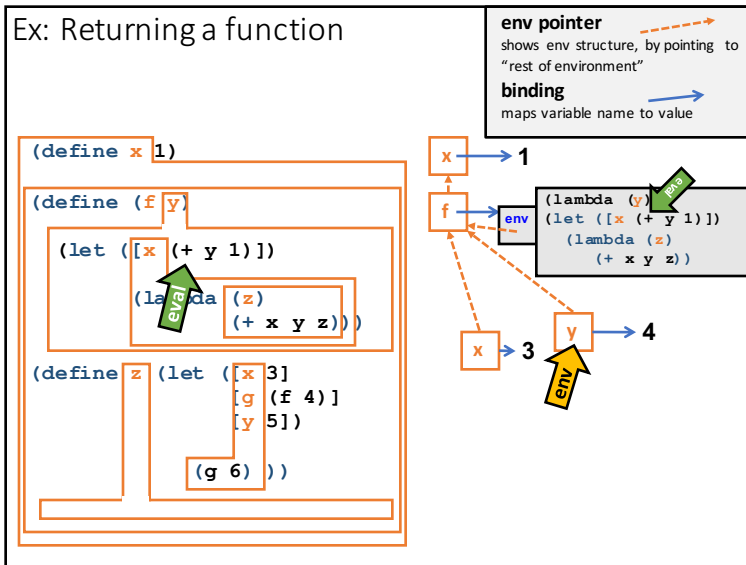
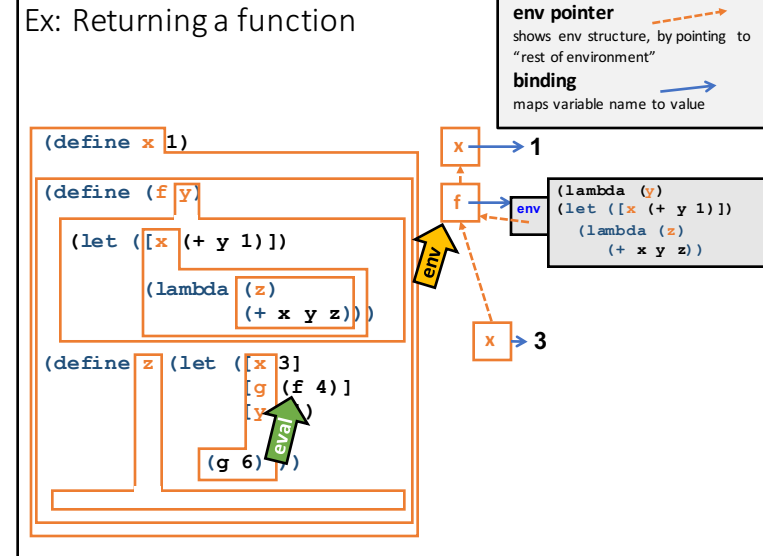
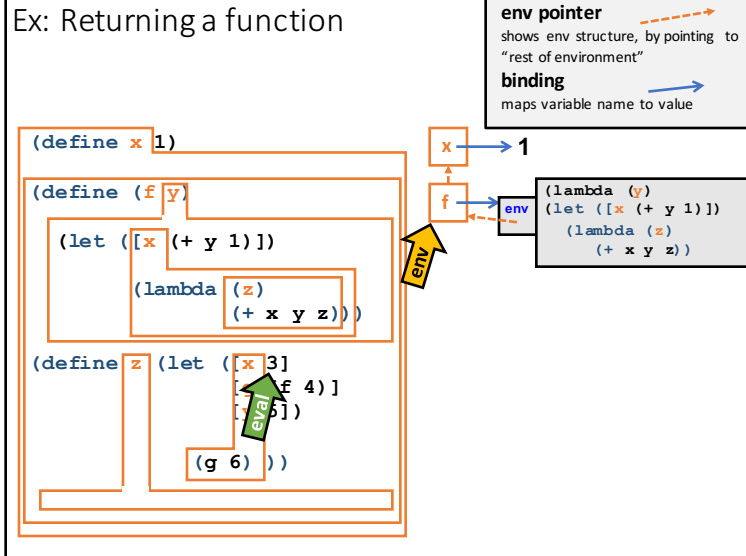
- Even taking / returning functions with higher-order functions!
- Makes first-class functions much more powerful.
 - Even if counterintuitive at first.
- Why alternative is problematic.

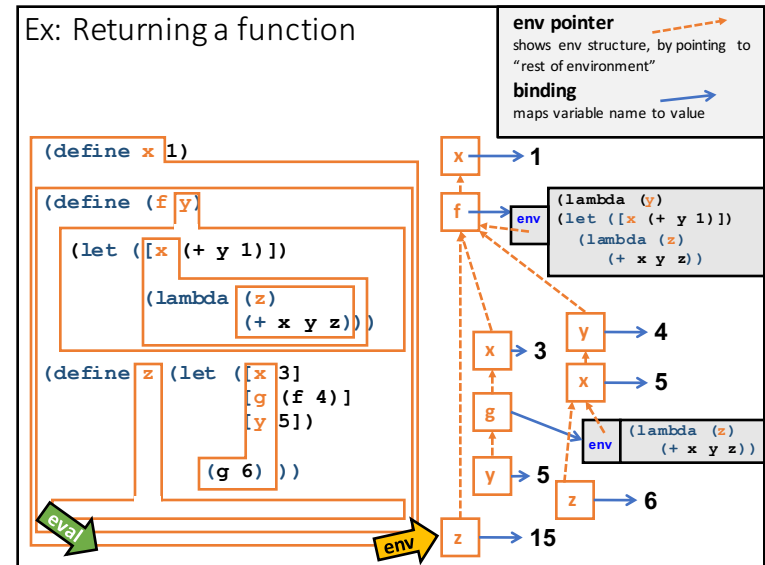
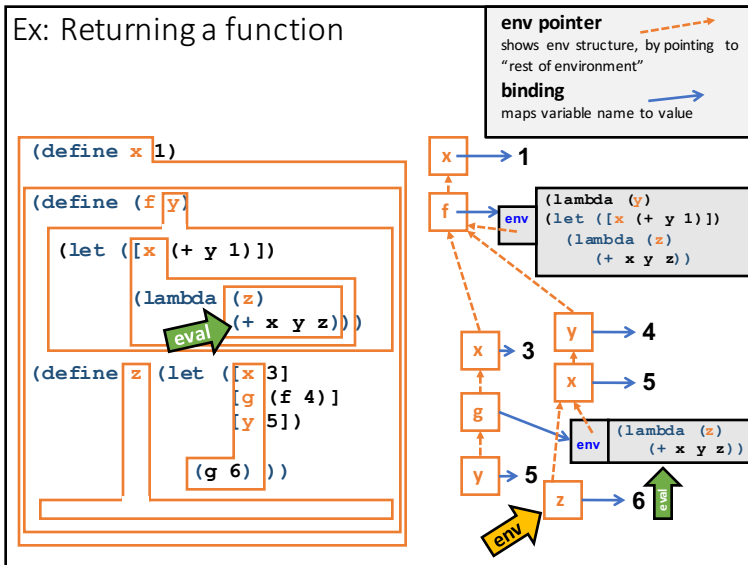
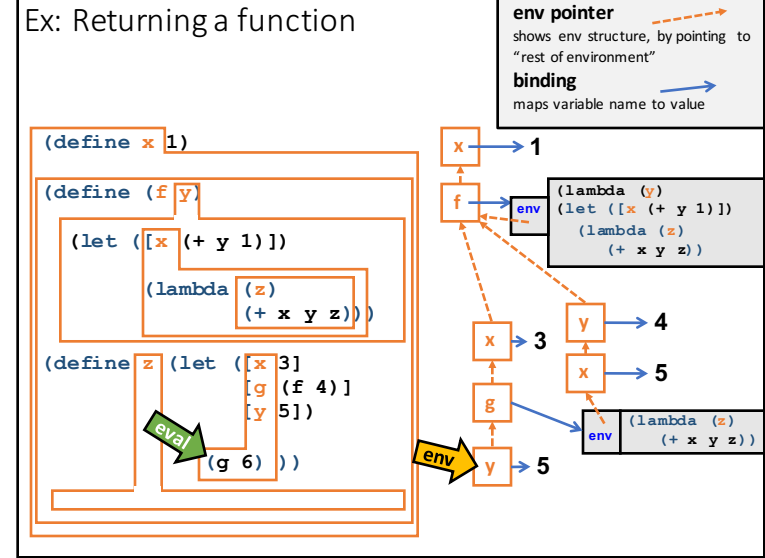
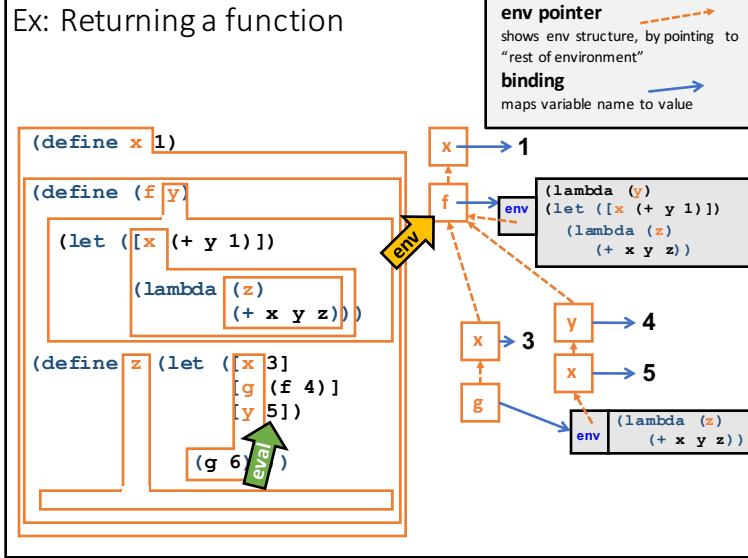
More examples in closures.rkt, notes. Draw...

Ex: Returning a function

```
(define x 1)
(define (f y)
  (let ([x (+ y 1)])
    (lambda (z)
      (+ x y z))))
(define z (let ([x 3]
                [g (f 4)]
                [y 5])
  (g 6)))
```

env pointer shows env structure, by pointing to "rest of environment"
binding maps variable name to value





Why lexical scope?

Lexical scope: use environment where function is defined

Dynamic scope: use environment where function is called

History has shown that lexical scope is almost always better.

Here are some precise, technical reasons (not opinion).

Why lexical scope?

1. Function meaning does not depend on variable names.

Example: change body of `f` to replace `x` with `q`.

- Lexical scope: it cannot matter
- Dynamic scope: depends how result is used

```
(define (f y)
  (let ([x (+ y 1)])
    (lambda (z) (+ x y z))))
```

Example: remove unused variables.

- Dynamic scope: but maybe some `g` uses it (weird).
- ```
(define (f g)
 (let ([x 3])
 (g 2)))
```

## Why lexical scope?

### 2. Functions can be understood fully where defined.

Example: dynamic scope tries to add `#f`, unbound variable `y`, and `4`.

```
(define (f y)
 (let ([x (+ y 1)])
 (lambda (z) (+ x y z)))
(define x #f)
(define g (f 7))
(define a (g 4))
```

## Why lexical scope?

### 3. Closures automatically “remember” the data they need.

More examples, idioms later.

```
(define (greater-than-x x)
 (lambda (y) (> y x)))

(define (no-negs xs)
 (filter (greater-than-x -1) xs))

(define (all-greater xs n)
 (filter (lambda (x) (> x n)) xs))
```

## Dynamic scope?

- Lexical scope definitely the right default for variables.
  - Very common across modern languages
- Early LISP used dynamic scope.
  - even though inspiration (lambda calculus) has lexical scope
  - Later "fixed" by Scheme (Racket's parent) and other languages.
- Dynamic scope is very occasionally convenient:
  - Racket has a special way to do it.
  - Perl
  - Most languages are purely lexically scoped.

## When things evaluate

A function body is **not evaluated until** the function is called.

A function body is **evaluated every time** the function is called.

A binding evaluates its expression **when the binding is evaluated**, not every time the variable is used.

## Closures for avoiding recomputation

These functions filter lists of lists by length.

```
(define (all-shorter-than-1 lists mine)
 (filter (lambda (xs) (< (length xs) (length mine))) lists))
```

```
(define (all-shorter-than-2 lists mine)
 (let ([len (length mine)])
 (filter (lambda (xs) (< (length xs) len)) lists)))
```

How many times is the length function called?