



Lexical Scope and Function Closures

Topics

- Lexical vs dynamic scope
- Closures implement lexical scope.
- Design considerations: why lexical scope?
- Relevant design dimensions

A question of scope (warmup)

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

What is the argument value passed to
this function application?

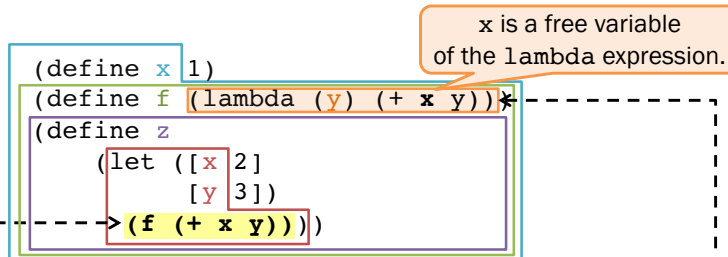
A question of scope

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

What is the value of x when this function body
is evaluated for this function application?

A question of *free variables*

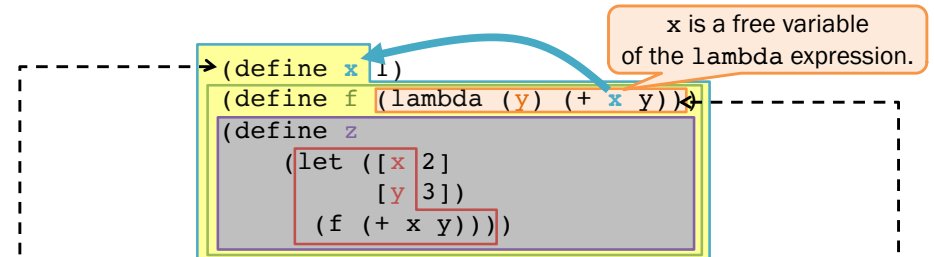
A variable, x , is **free** in an expression, e , if x is referenced in e outside the scope of any binding of x within e .



To what bindings do free variables of a function refer when the function is applied?

Answer 1: *lexical (static) scope*

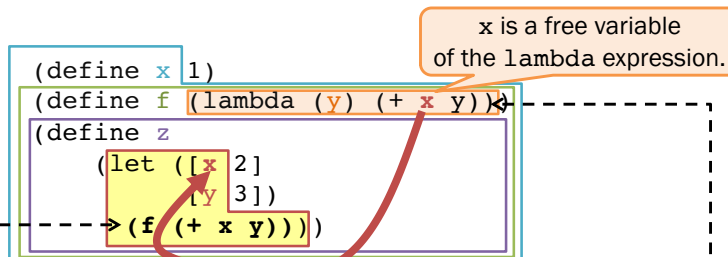
A variable, x , is **free** in an expression, e , if x is referenced in e outside the scope of any binding of x within e .



Free variables of a function refer to bindings in the environment where the function is **defined**, regardless of where it is applied.

Answer 2: *dynamic scope*

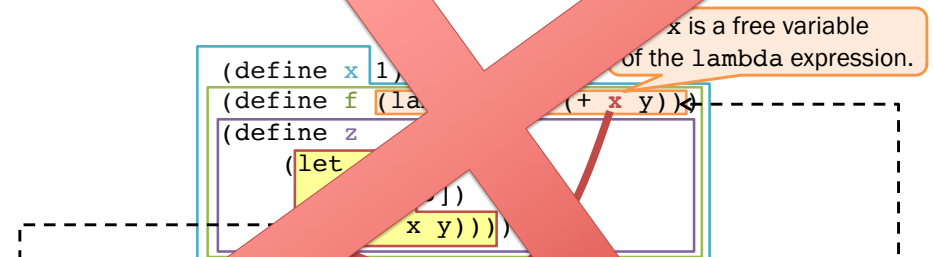
A variable, x , is **free** in an expression, e , if x is referenced in e outside the scope of any binding of x within e .



Free variables of a function refer to bindings in the environment where the function is **applied**, regardless of where it is defined.

Answer 2: *dynamic scope*

A variable, x , is **free** in an expression, e , if x is referenced in e outside the scope of any binding of x within e .



Free variables of a function refer to bindings in the environment where the function is **applied**, regardless of where it is defined.

Closures implement lexical scope.

Closures allow functions to use any binding in the environment where the function is defined, regardless of where it is applied.

Anonymous function definition expressions

Syntax: $(\text{lambda } (x_1 \dots x_n) e)$

- parameters: x_1 through x_n are identifiers
- body: e is any expression



Evaluation:

1. The result is a **function closure**, $\langle E, (\text{lambda } (x_1 \dots x_n) e) \rangle$, holding **the current environment, E** , and the function.

[closure]

$$E \vdash (\text{lambda } (x_1 \dots x_n) e) \downarrow \langle E, (\text{lambda } (x_1 \dots x_n) e) \rangle$$

Note:

- An anonymous function definition is an expression.
- A function closure is a new kind of value. Closures are not expressions.
- This is a **definition**, not a call. The body, e , is **not** evaluated now.
- lambda from the λ -calculus.

Function application (call)

Syntax: $(e_0 e_1 \dots e_n)$



Evaluation:

1. Under the current dynamic environment, E , evaluate e_0 through e_n to values v_0, \dots, v_n .
2. If v_0 is a function closure of n arguments, $\langle E', (\text{lambda } (x_1 \dots x_n) e) \rangle$ then
The result is the result of evaluating the closure body, e , **under the closure environment, E'** , extended with argument bindings:
 $x_1 \mapsto v_1, \dots, x_n \mapsto v_n$.

Otherwise, there is a type error.

Function application (call)

Syntax: $(e_0 e_1 \dots e_n)$



Evaluation:

$$E \vdash e_0 \downarrow \langle E', (\text{lambda } (x_1 \dots x_n) e) \rangle$$

$$E \vdash e_1 \downarrow v_1$$

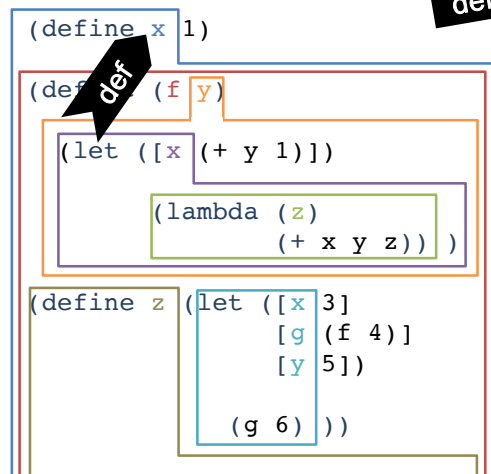
...

$$E \vdash e_n \downarrow v_n$$

$$x_1 \mapsto v_1, \dots, x_n \mapsto v_n, E' \vdash e \downarrow v$$

$$E \vdash (e_0 e_1 \dots e_n) \downarrow v \quad \text{[apply]}$$

Example: returning a function



def

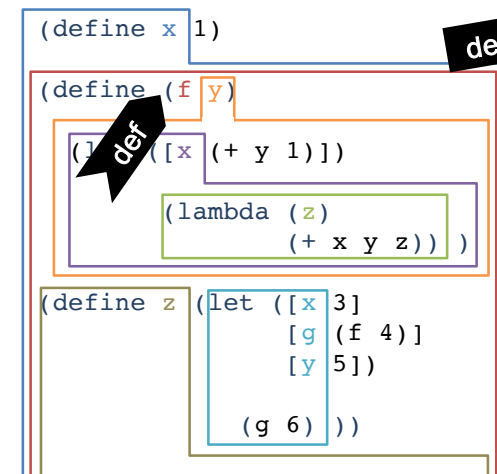
env pointer
shows env structure, by pointing to
"rest of environment"
binding
maps variable name to value

Current evaluation step:

Current environment:



Example: returning a function

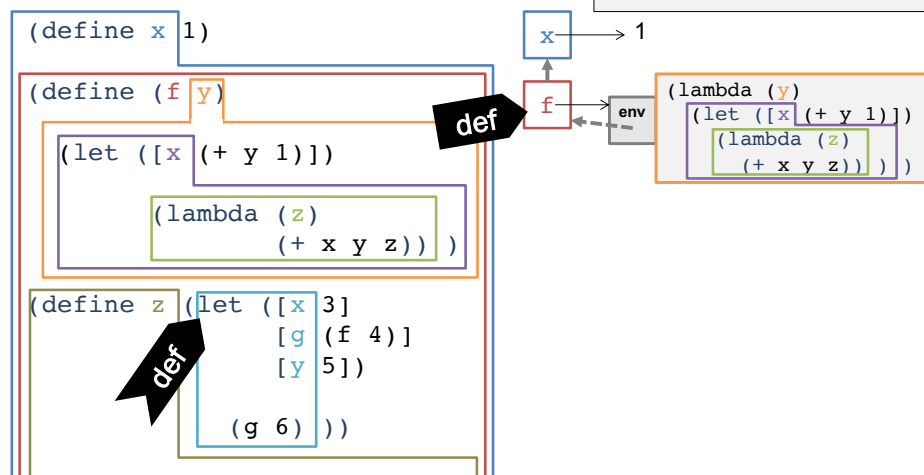


def



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

Example: returning a function

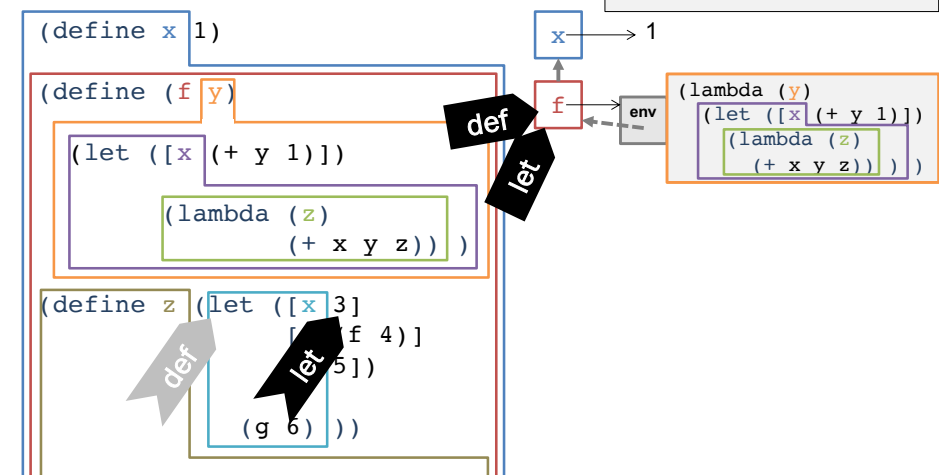


def



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

Example: returning a function



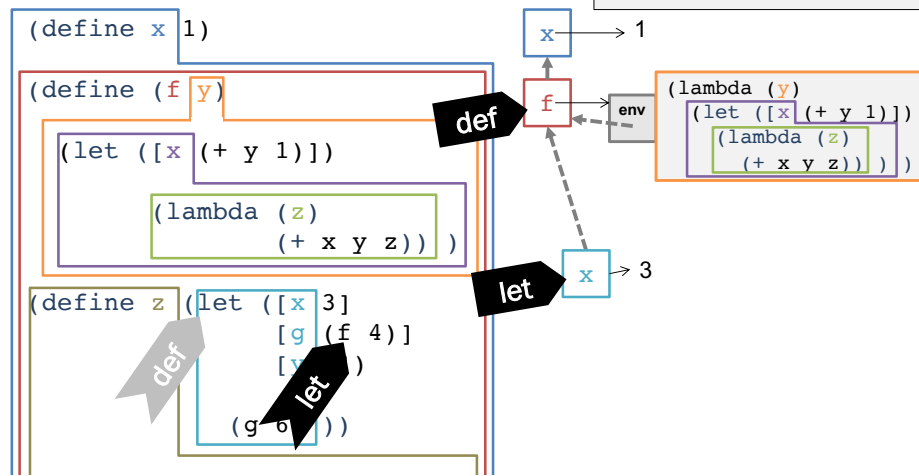
def

let

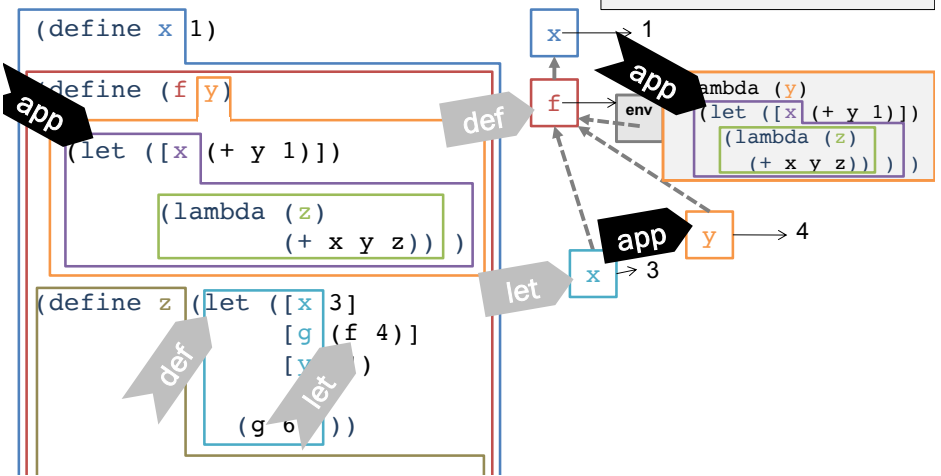


env pointer
shows env structure, by pointing to
"rest of environment"
binding
maps variable name to value

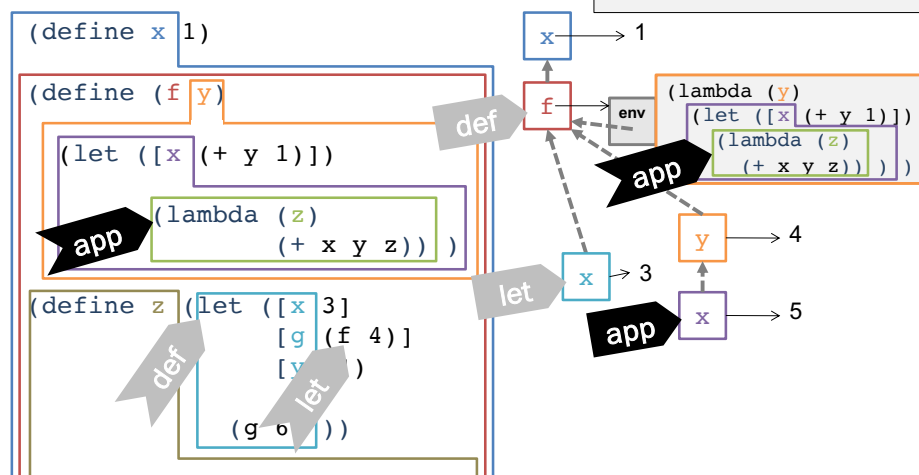
Example: returning a function



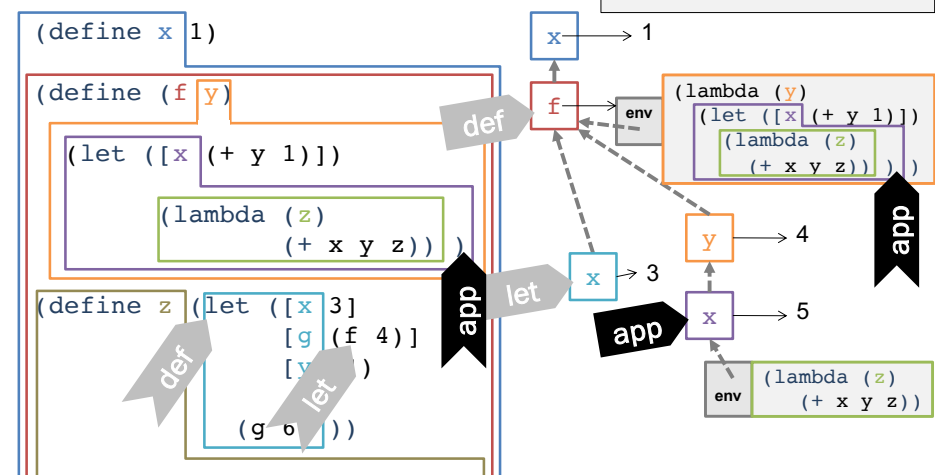
Example: returning a function



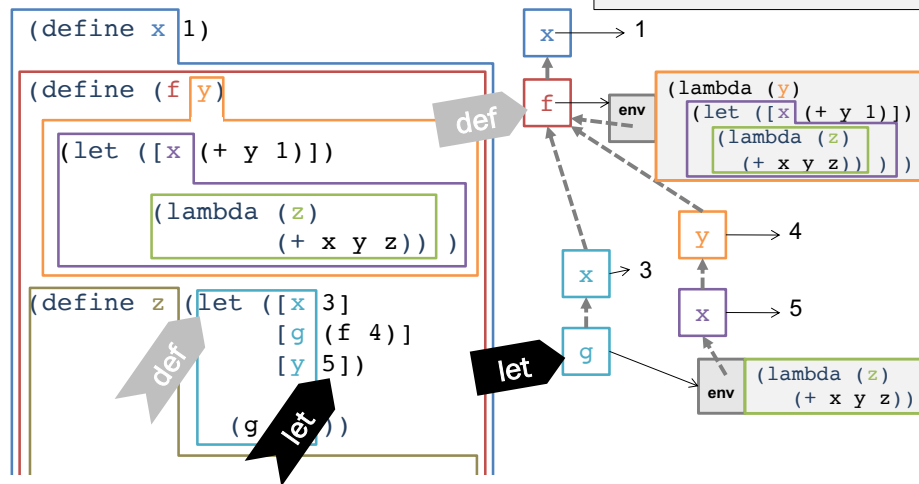
Example: returning a function



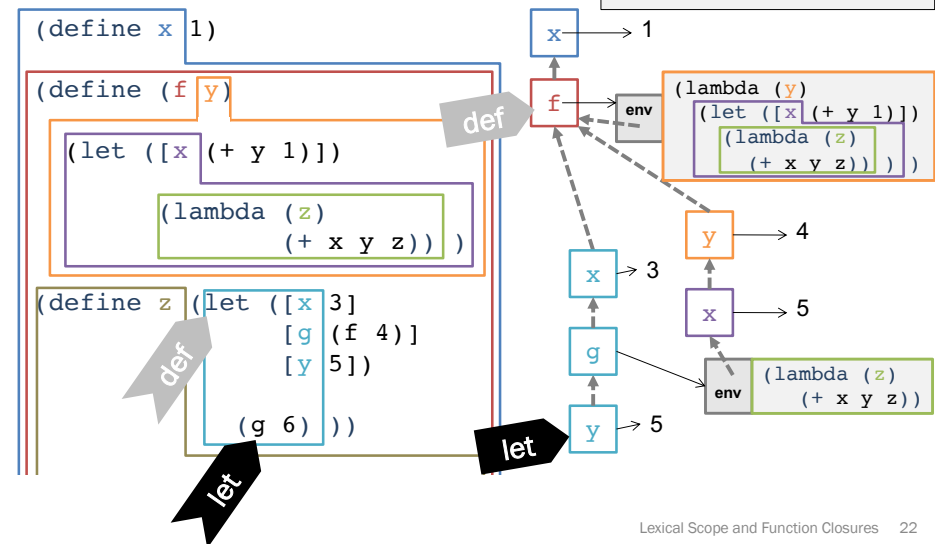
Example: returning a function



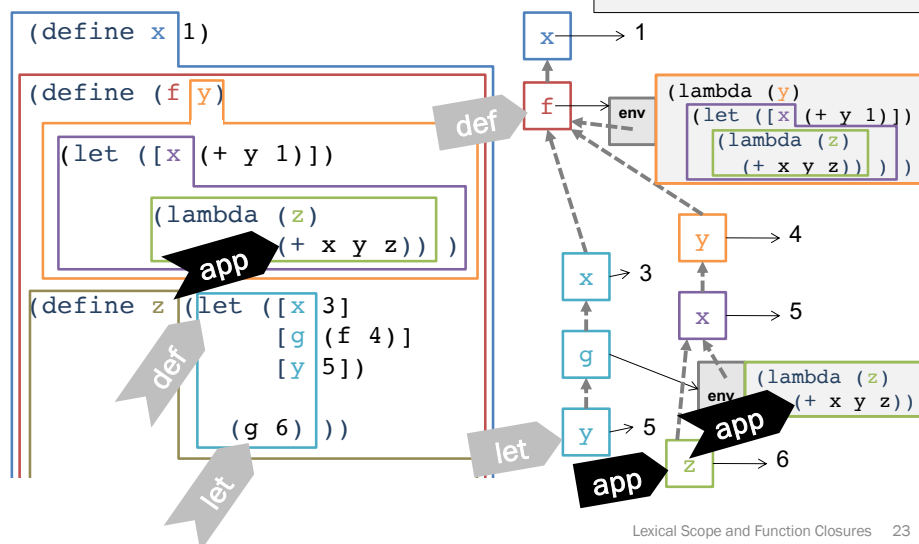
Example: returning a function



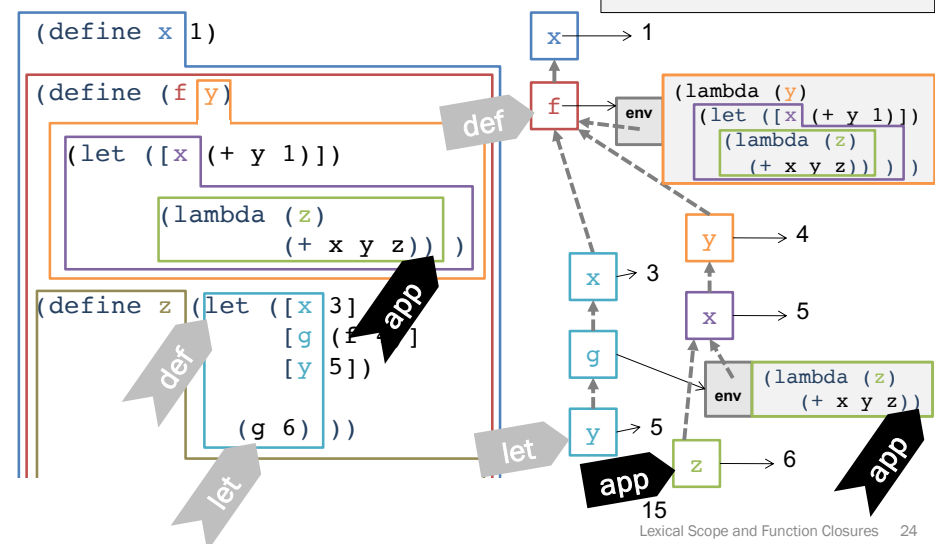
Example: returning a function



Example: returning a function

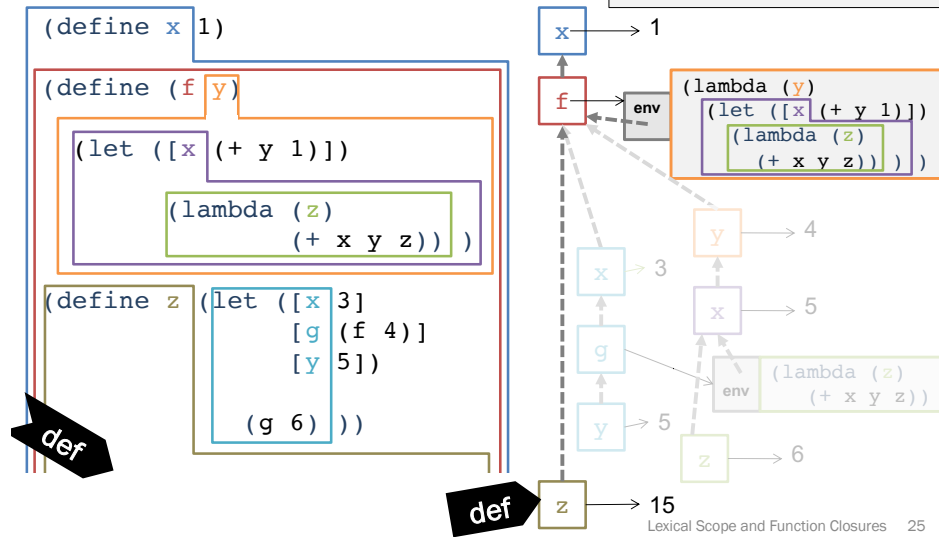


Example: returning a function



Example: returning a function

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



Debrief

1. Closures implement lexical scope.
2. Function bodies can use bindings from the environment where they were defined, not where they were applied.
3. The environment is not a stack.
 - Multiple environments (branches) may be live simultaneously.
 - CS 240's basic stack model will not suffice.
 - General case: heap-allocate the environment. GC will clean up for us!

PL design quiz

Java methods and C functions do not need closures because they _____.

- a. cannot refer to names defined outside the method/function
- b. are not first class values
- c. do not use lexical scope
- d. are not anonymous (i.e., they are named)

Which, if any, are correct? Why?

Why lexical scope?

Lexical scope: use environment where function is **defined**.

Dynamic scope: use environment where function is **applied**.

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 name choices.

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))))
```

(!) It is important in both cases that no other variable named `q` is used in `f`.

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.

There are no "hidden parameters."

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?

3a. 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))
```

Why lexical scope?

3b. Closures are a useful way to avoid 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?

Dynamic scope?

- Lexical scope is definitely the right default for variables.
 - Nearly all 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.
 - Emacs Lisp still uses dynamic scope.
- Dynamic scope is very occasionally convenient:
 - Racket has a special way to do it.
 - Perl has something similar.
 - Most languages are purely lexically scoped.
 - **Exception raise/handle, throw/catch is like dynamic scope.**

Remember 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.

As with lexical/dynamic scope, there are other options here that Racket does **not** use. We will consider some later.

Relevant PL design dimensions

in the Racket language:

- scope: lexical (static)
 - vs. dynamic
- parameter passing: pass-by-value (call-by-value)
 - vs. by-reference, by-name, by-need
- evaluation order: eager (strict)
 - vs. lazy

in our semantics of the Racket language:

- environments and closures
 - vs. substitution
- big-step operational semantics
 - vs. small-step

More on all of these dimensions (and alternatives) later!