WELLESLEY

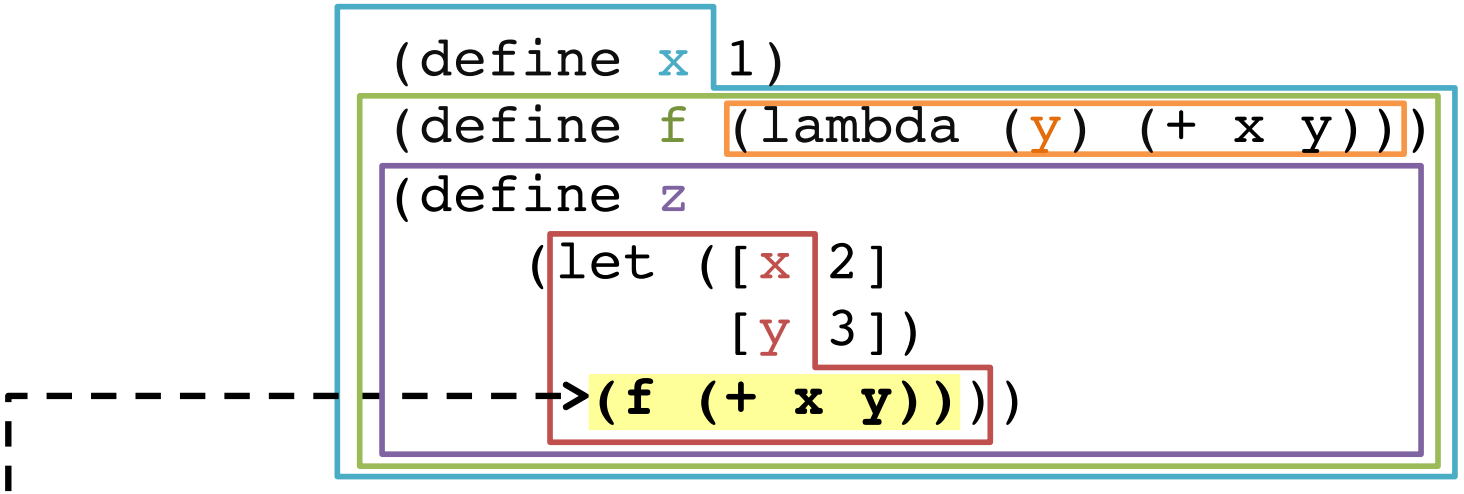
Lexical Scope and Function Closures

+closures.rkt

Topics

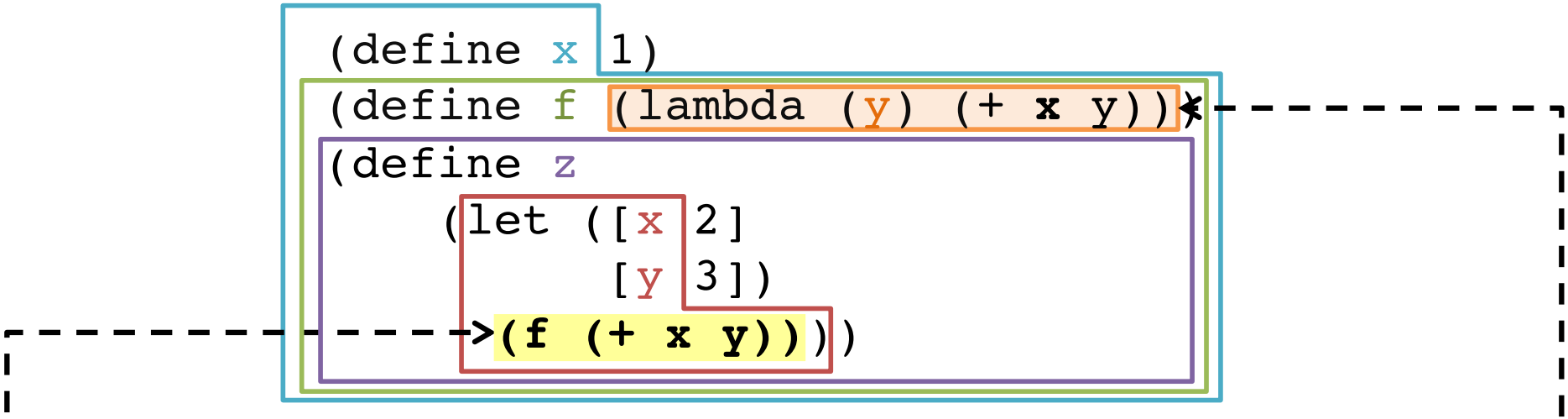
- Lexical vs dynamic scope
- Closures implement lexical scope.
- Design considerations: why lexical scope?
- Relevant design dimensions

A question of scope (warmup)



What is the argument value passed to this function application?

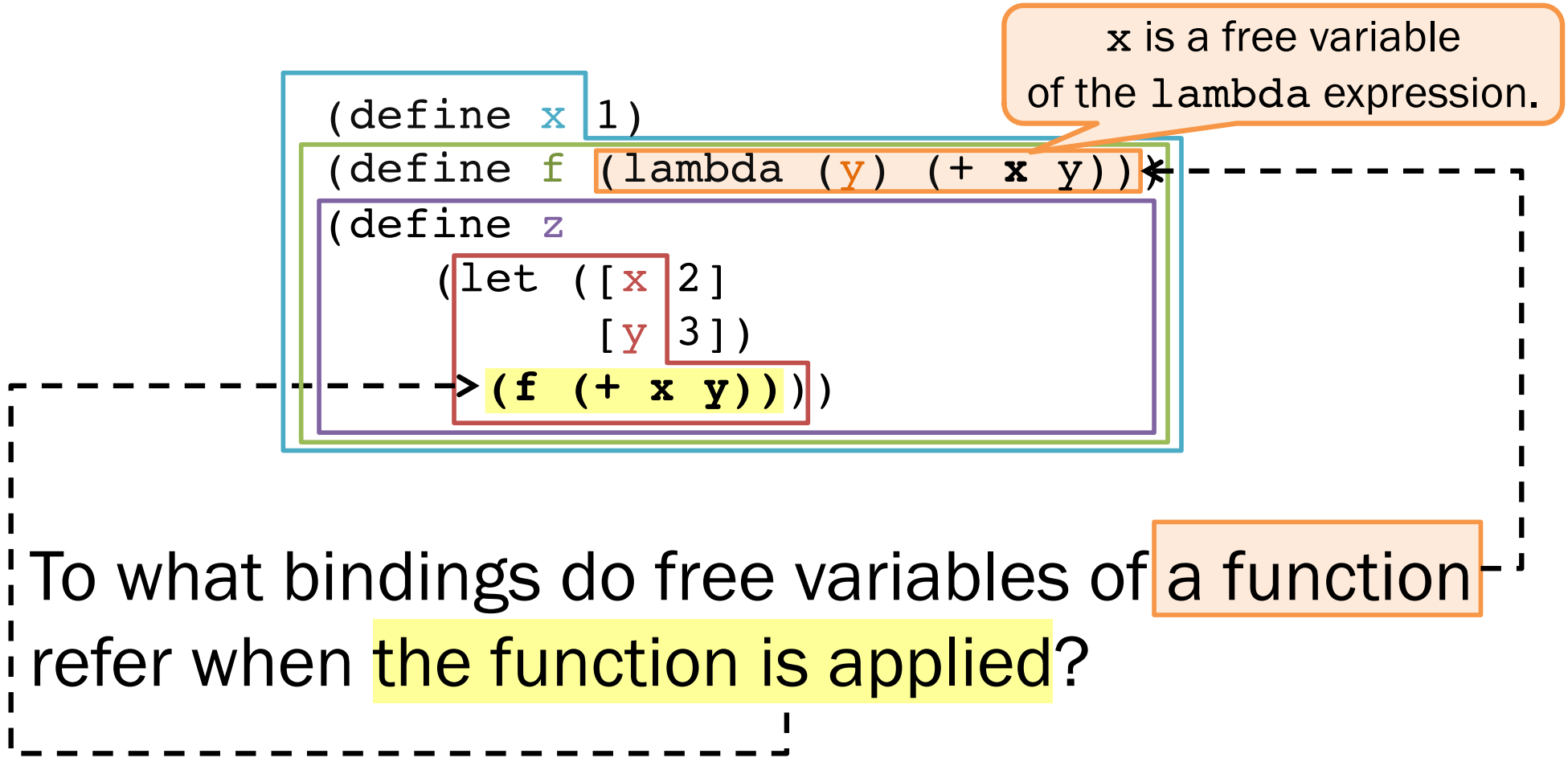
A question of scope



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

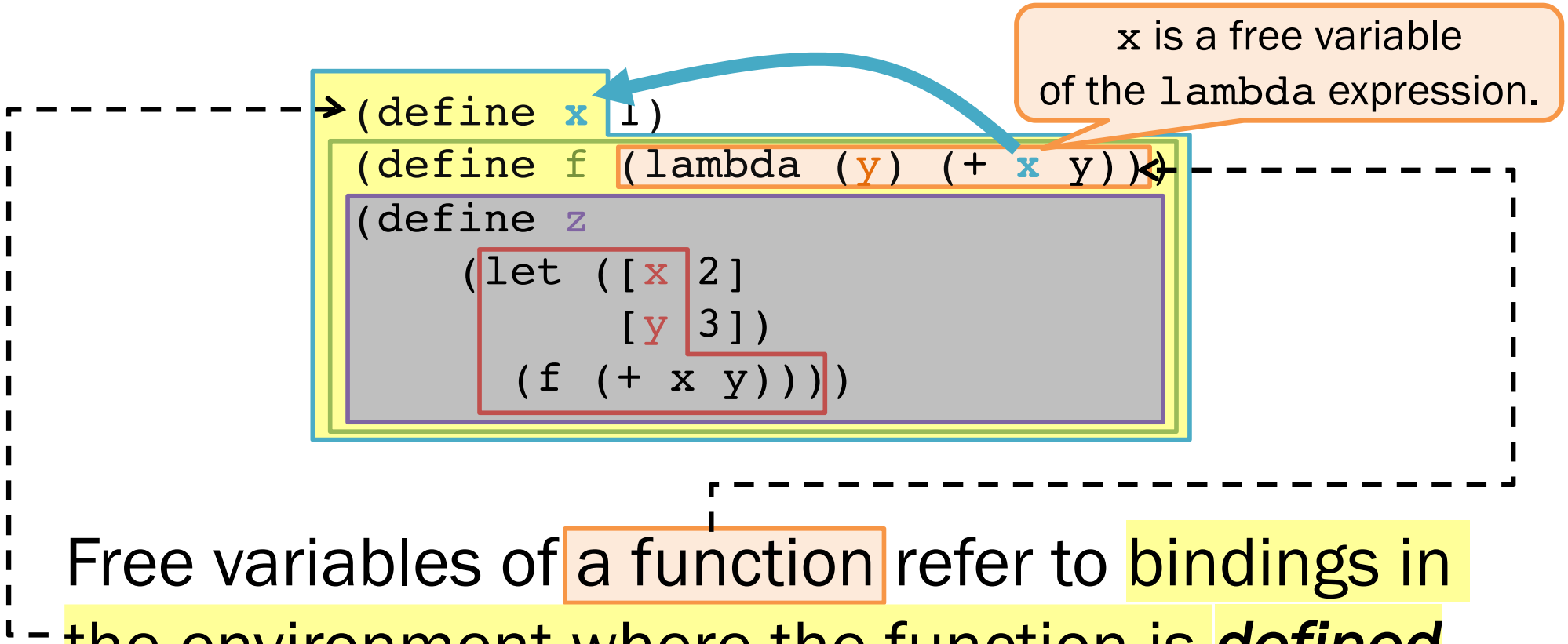
A question of *free variables*

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`.



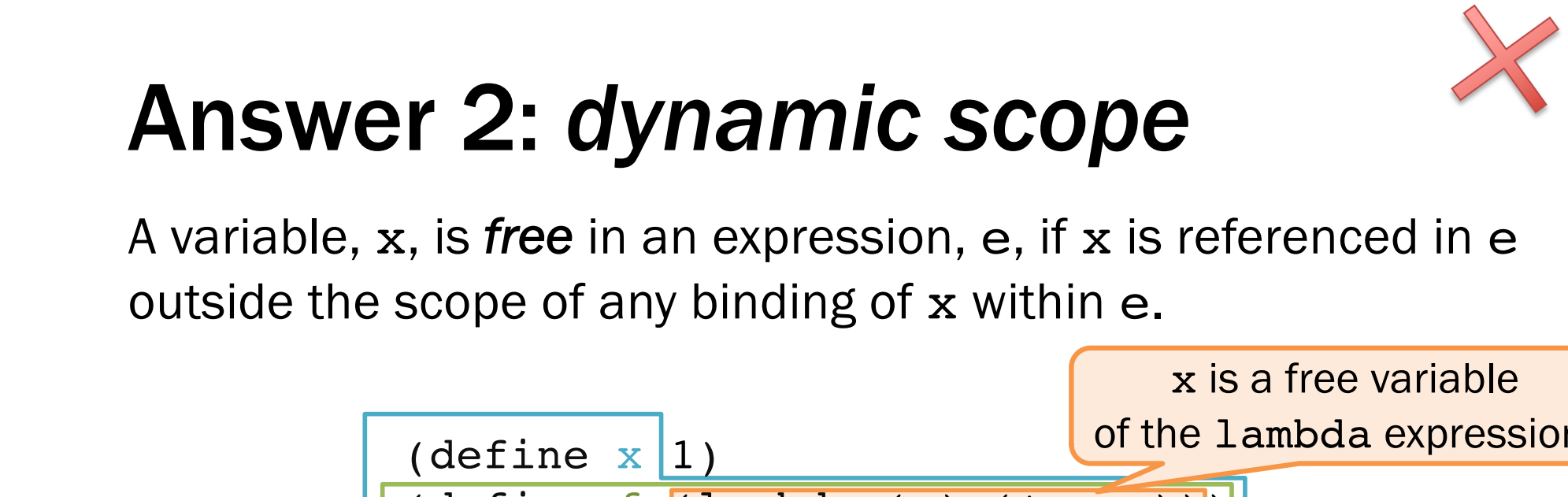
Answer 1: *lexical (static)* scope

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`.

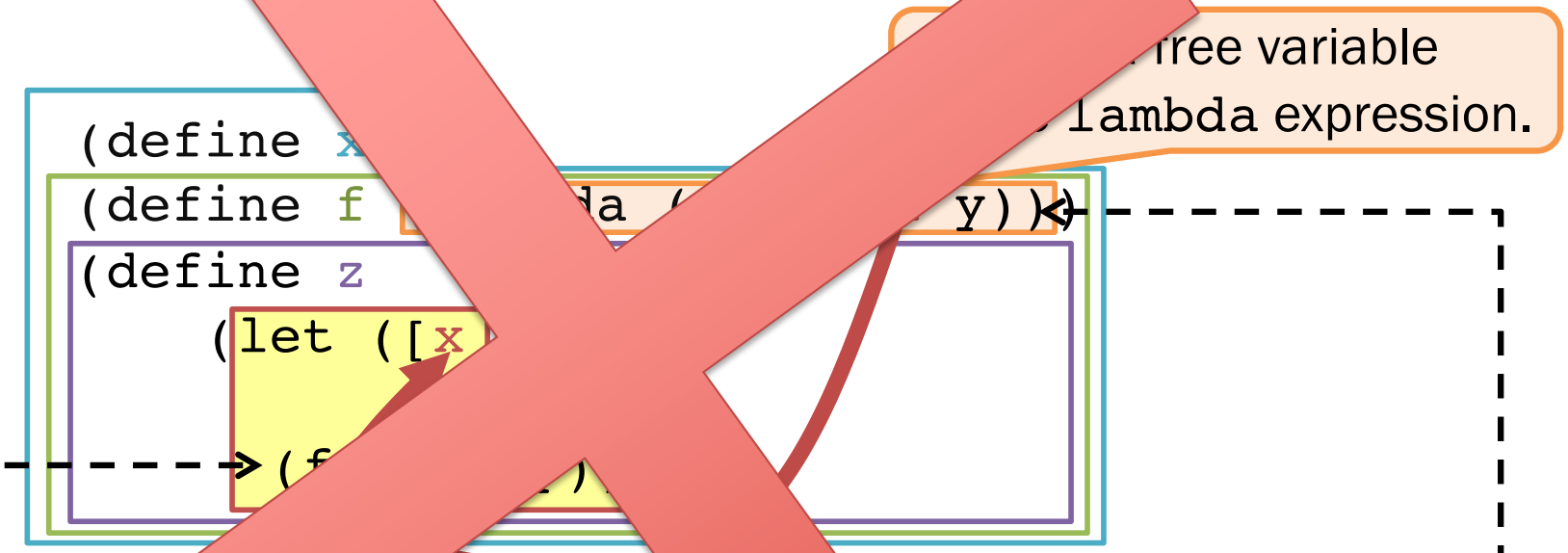
`x` is a free variable of the `lambda` expression.

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

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 [illegible] an expression, `e`, if `x` is referenced in `e` outside the scope [illegible] binding of `x` within `e`.



Free vari[illegible] of a function [illegible] to bindings in the environment where the fun[illegible] 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: `(lambda (x1 … xn) e)`

- parameters: `x1` through `xn` are identifiers
- body: `e` is any expression

Evaluation:

1. The result is a ***function closure***, ⟨`E, (lambda (x1 … xn) e)`⟩, holding the current environment, `E`, and the function.

[closure]

$$\texttt{E} \vdash \texttt{(lambda (x1 … xn) e)} \downarrow \langle \texttt{E, (lambda (x1 … xn) e)} \rangle$$

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: `(e0 e1 ... en)`

Evaluation:

1. Under the current dynamic environment, `E`, evaluate `e0` through `en` to values `v0, …, vn.`
2. If `v0` is a function closure of n arguments, ⟨`E'`, `(lambda (x1 … xn) e)`⟩ then

   The result is the result of evaluating the closure body, `e`, under the closure environment, `E'`, extended with argument bindings: `x1` ⟼ `v1, …, xn` ⟼ `vn`.

   Otherwise, there is a type error.

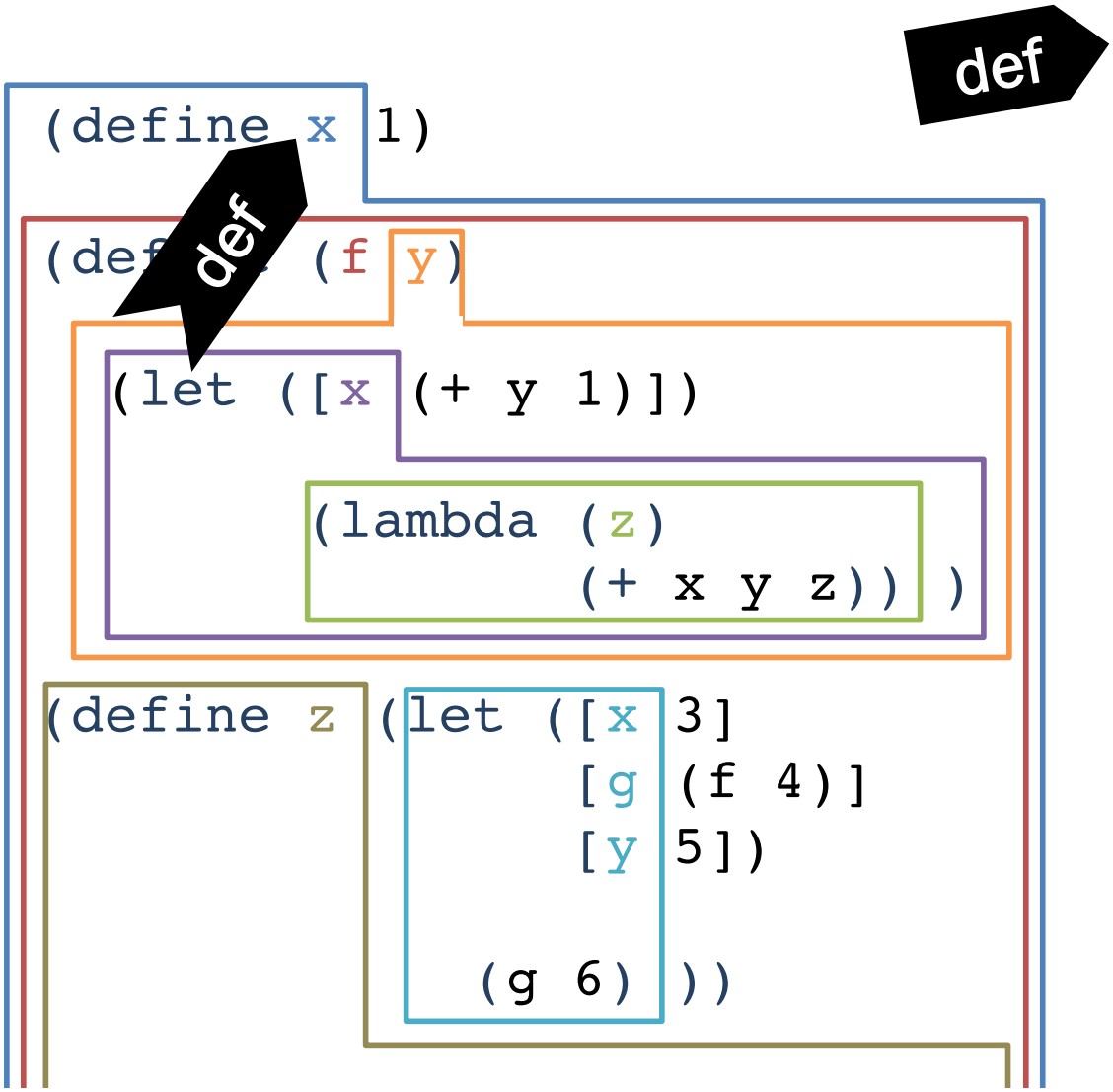
Function application (call)

Syntax: `(e0 e1 … en)`

Evaluation:

$$\frac{\begin{array}{l} E \vdash e0 \downarrow \langle E', (\text{lambda } (x1 \ldots xn)\ e)\rangle \\ E \vdash e1 \downarrow v1 \\ \quad\quad \ldots \\ E \vdash en \downarrow vn \\ x1 \longmapsto v1, \ldots, xn \longmapsto vn, E' \vdash e \downarrow v \end{array}}{E \vdash (e0\ e1\ \ldots\ en) \downarrow v}\ [\text{apply}]$$

Example: returning a function



env pointer
shows env structure, by pointing to "rest of environment"

binding
maps variable name to value

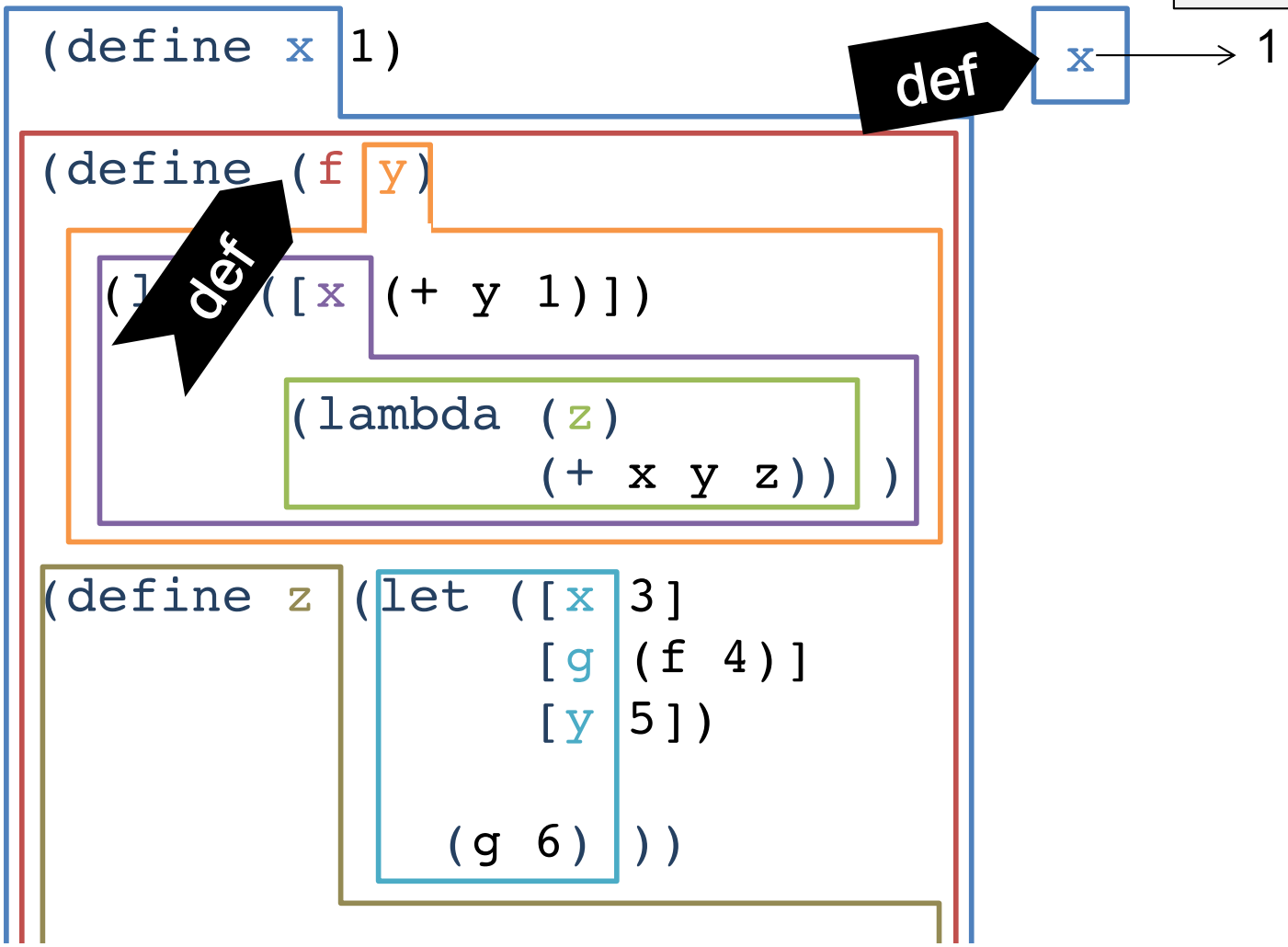
Current evaluation step:

Current environment:

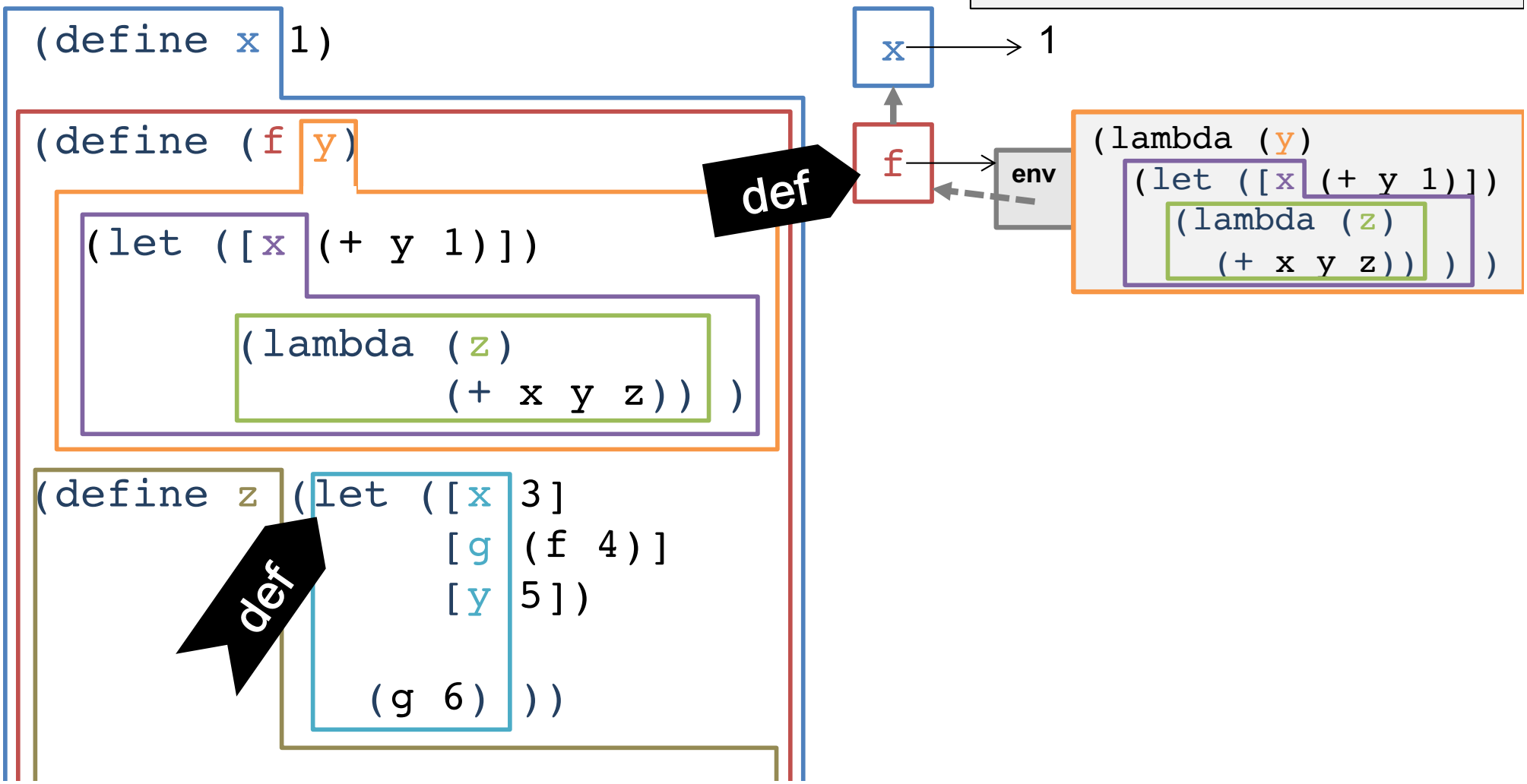
Example: returning a function

env pointer
shows env structure, by pointing to "rest of environment"

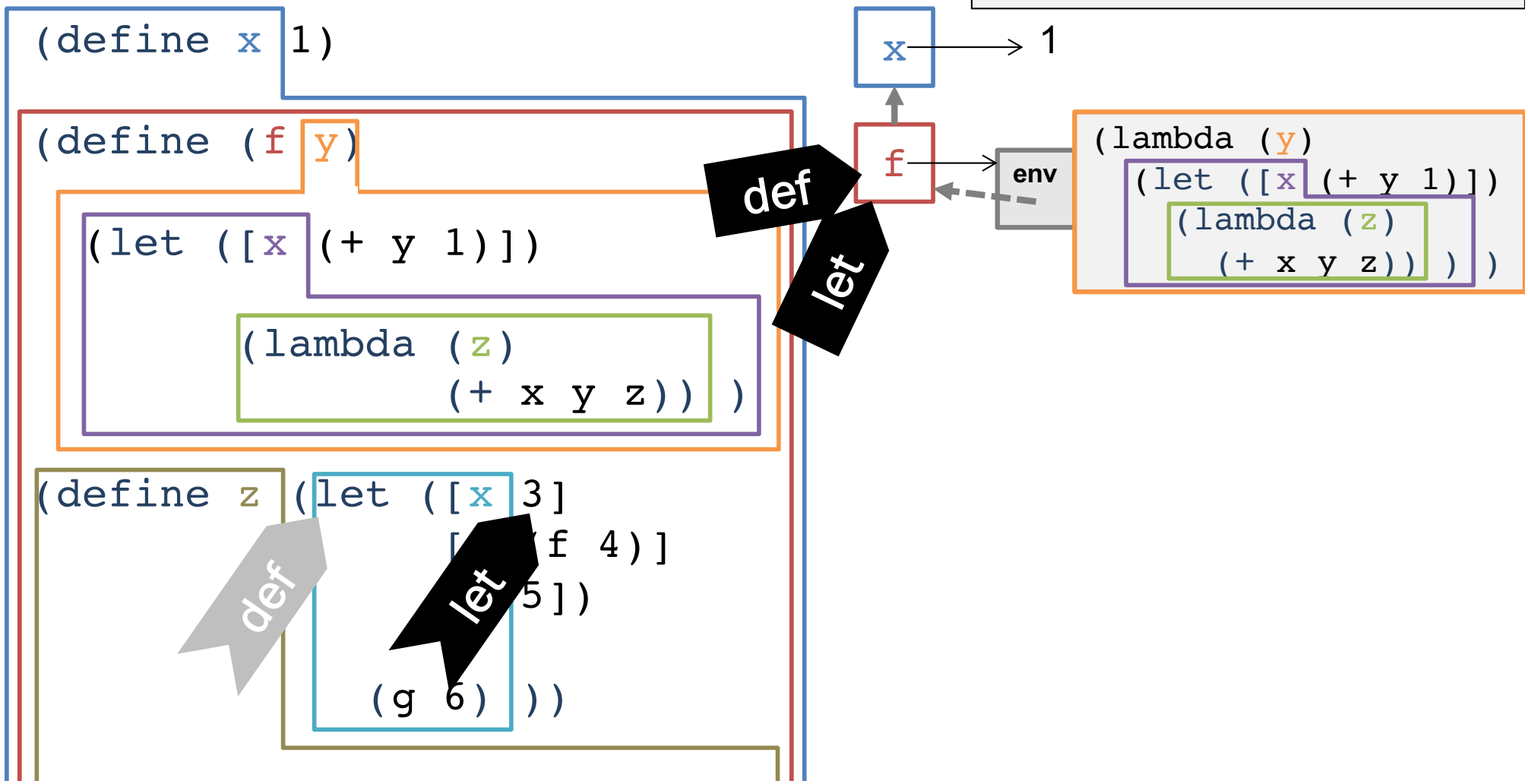
binding
maps variable name to value



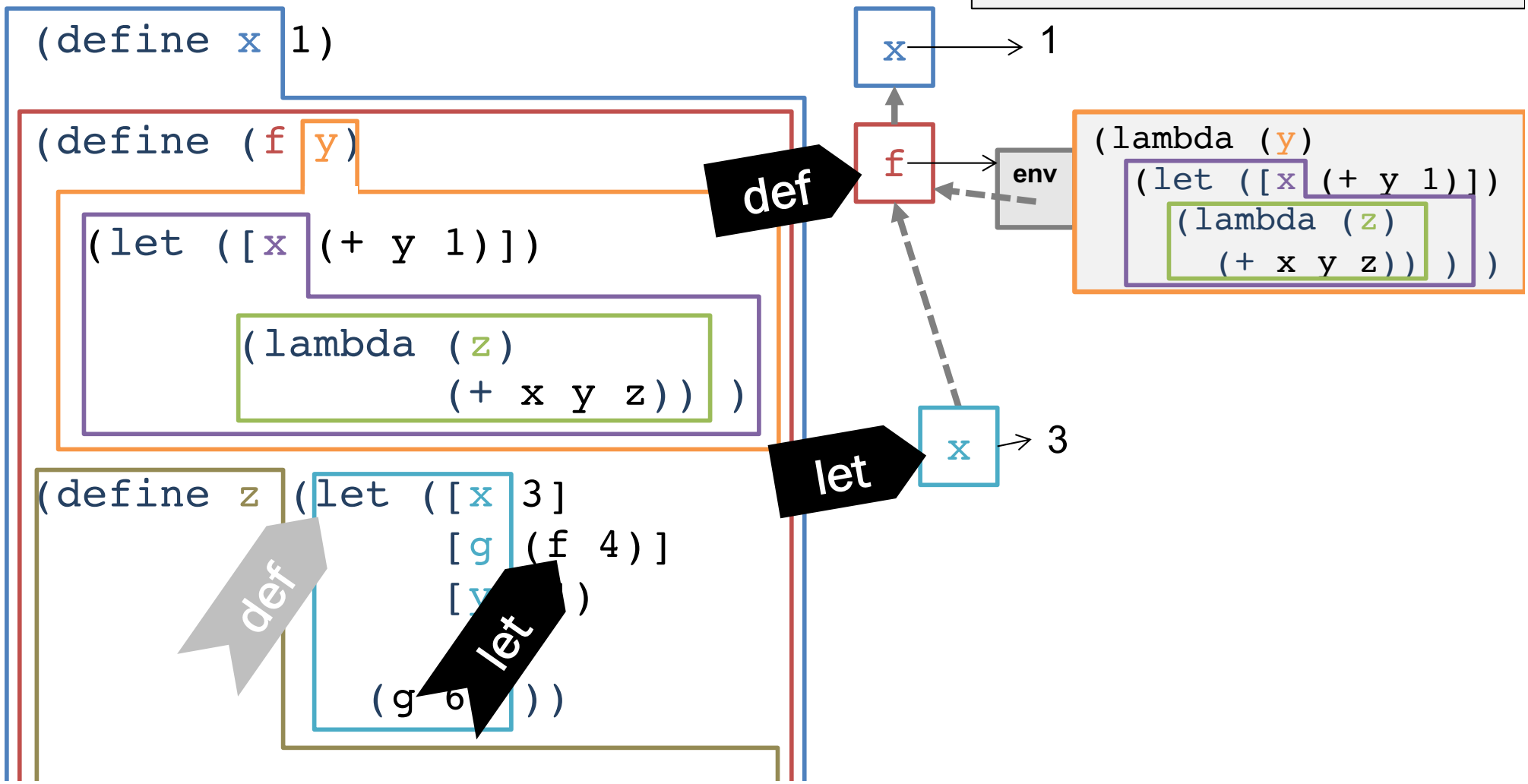
Example: returning a function



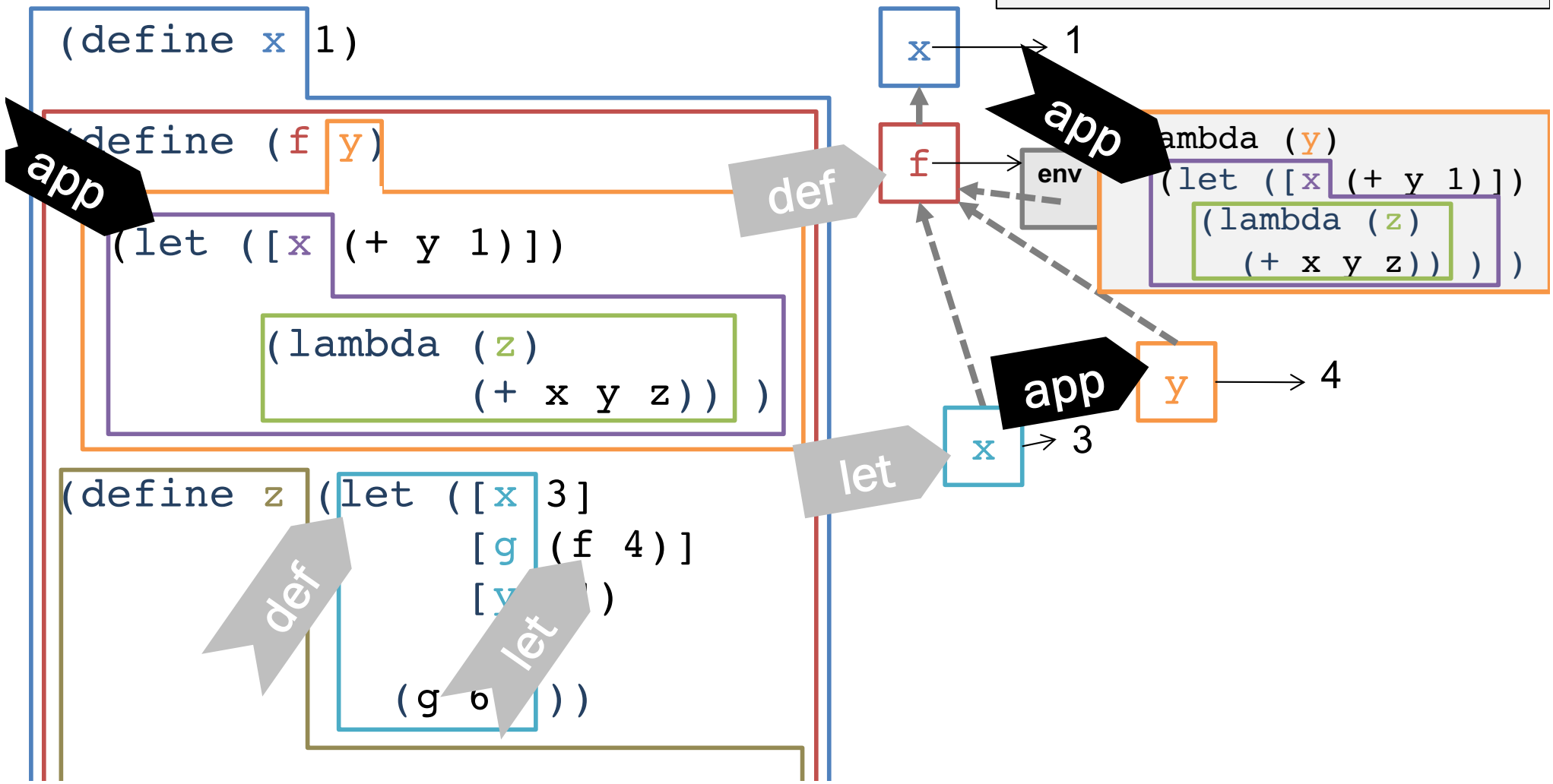
Example: returning a function



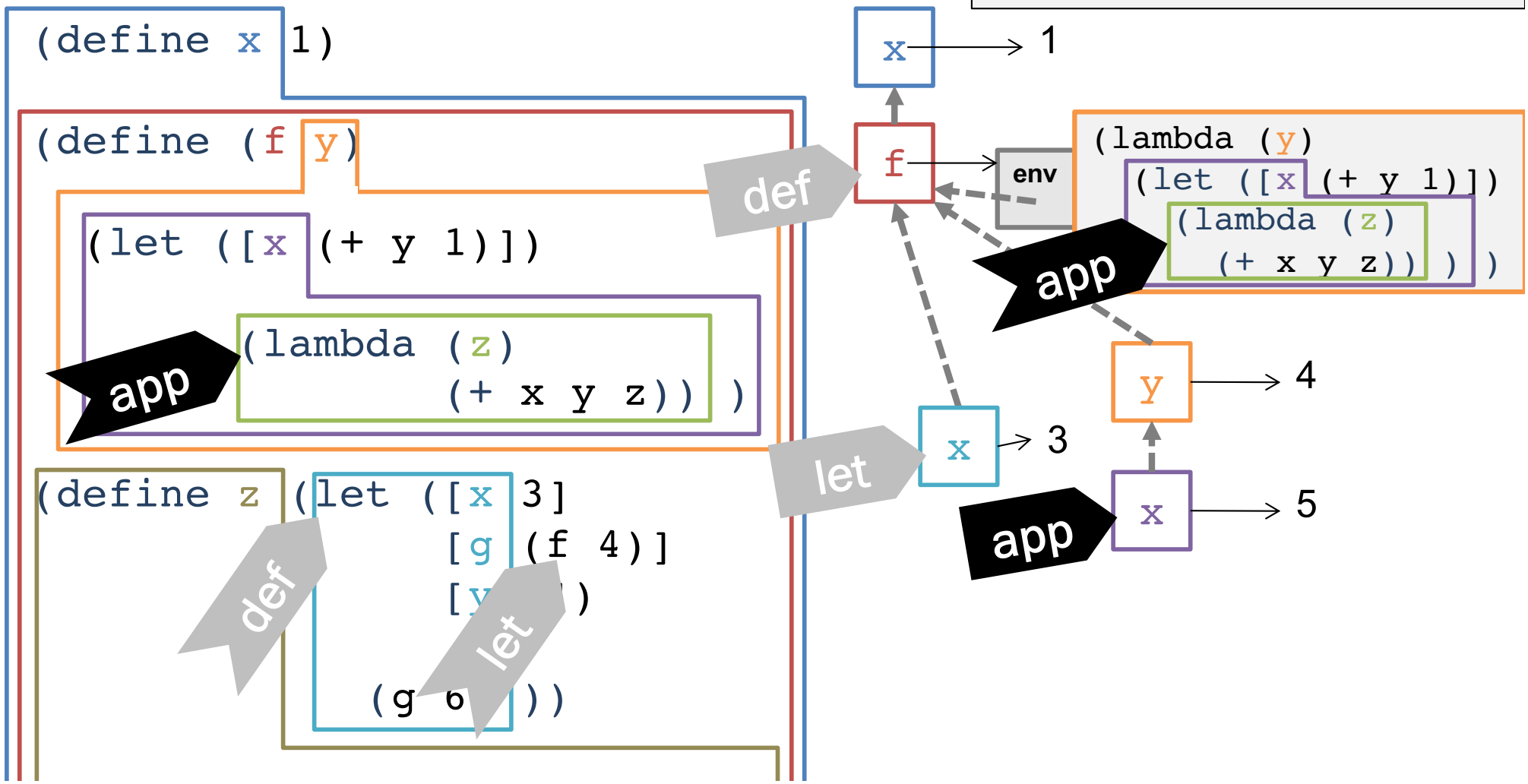
Example: returning a function



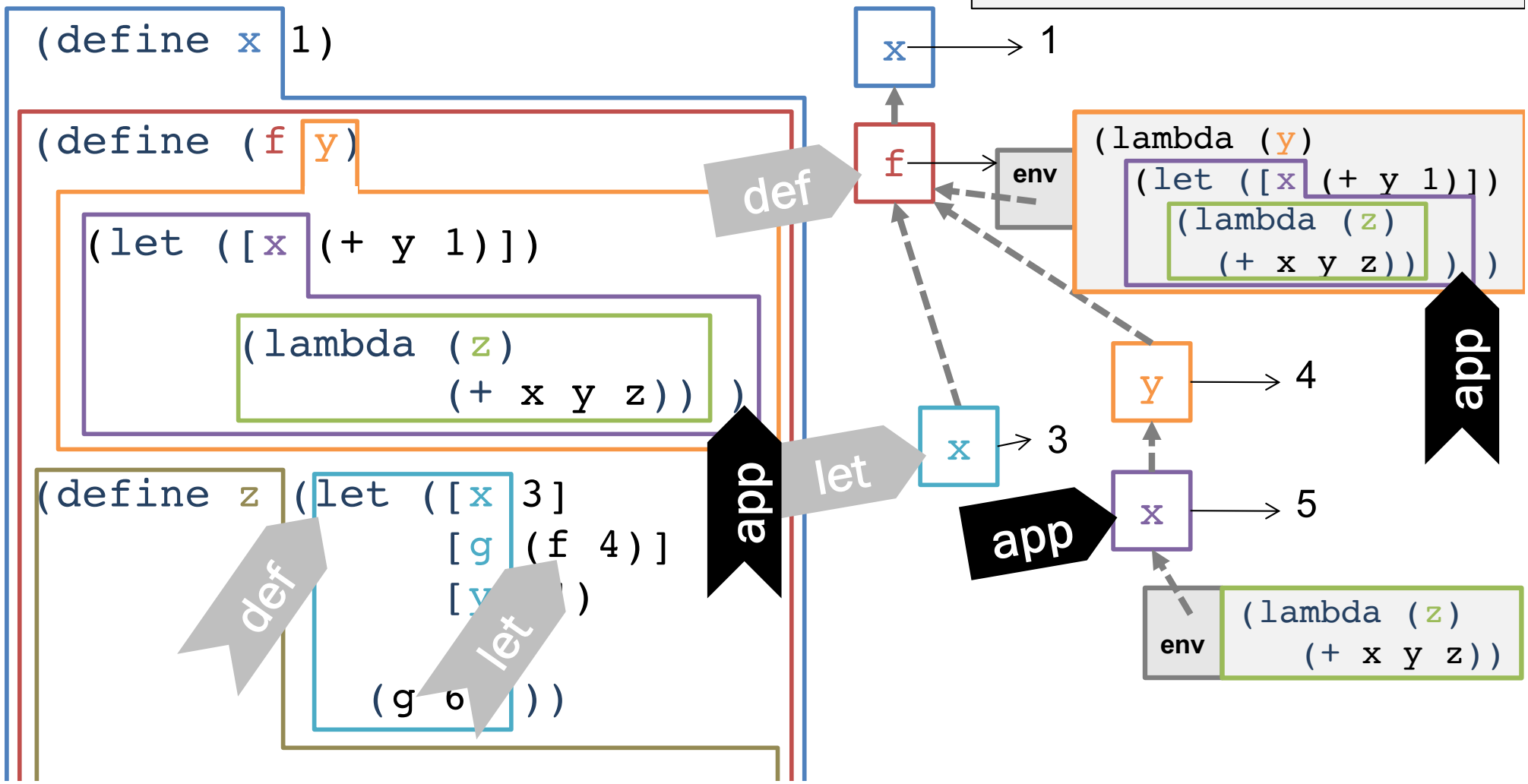
Example: returning a function



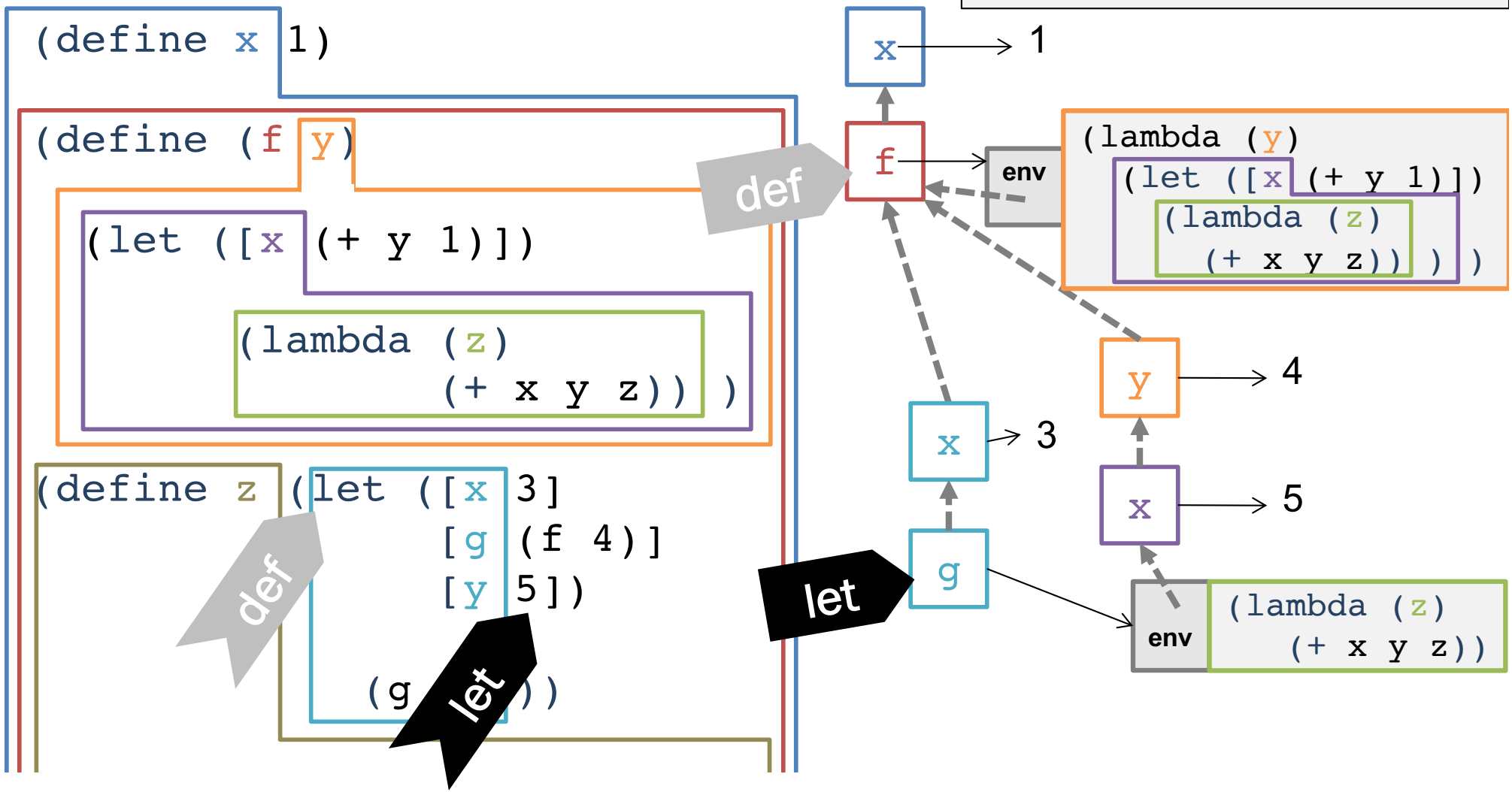
Example: returning a function



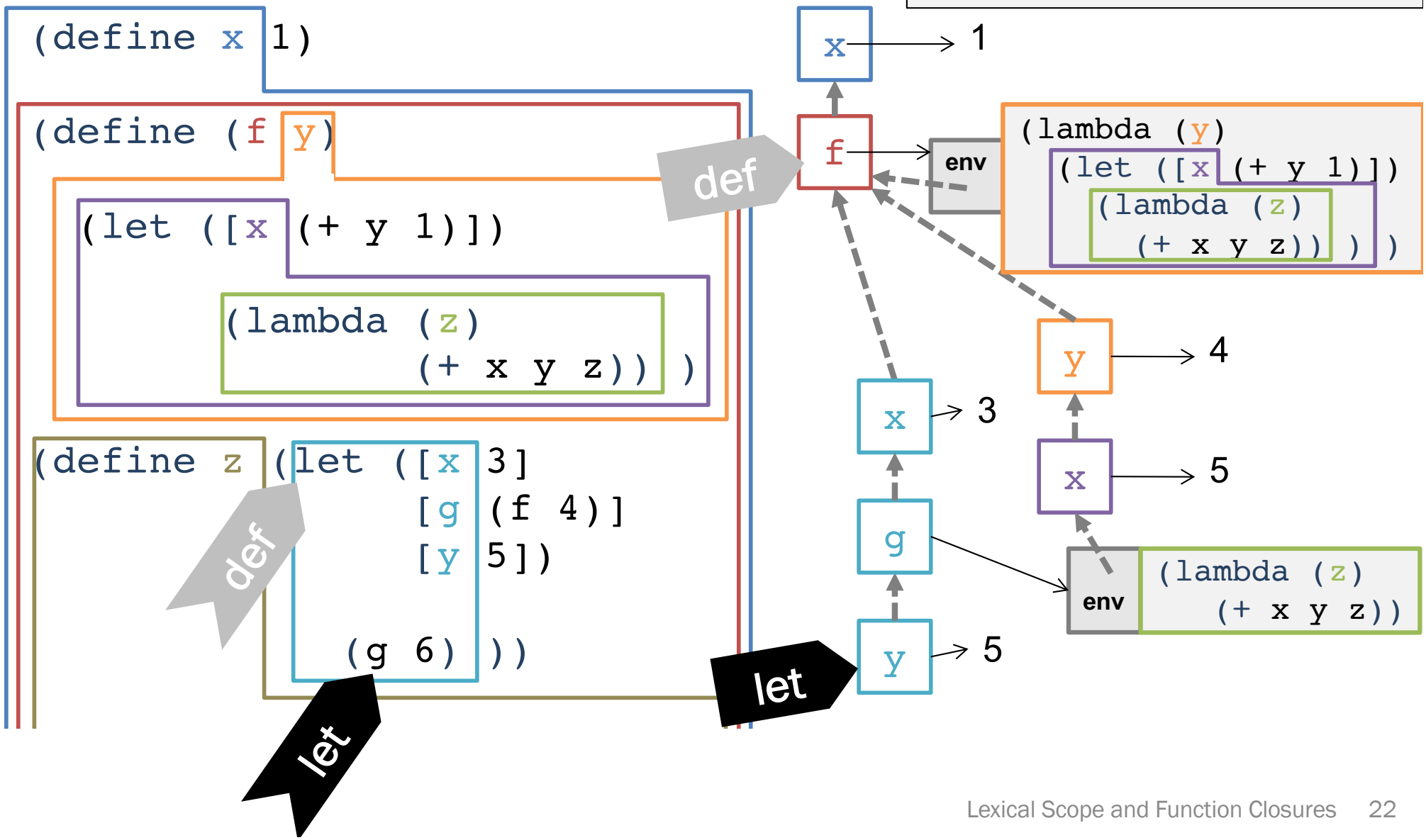
Example: returning a function



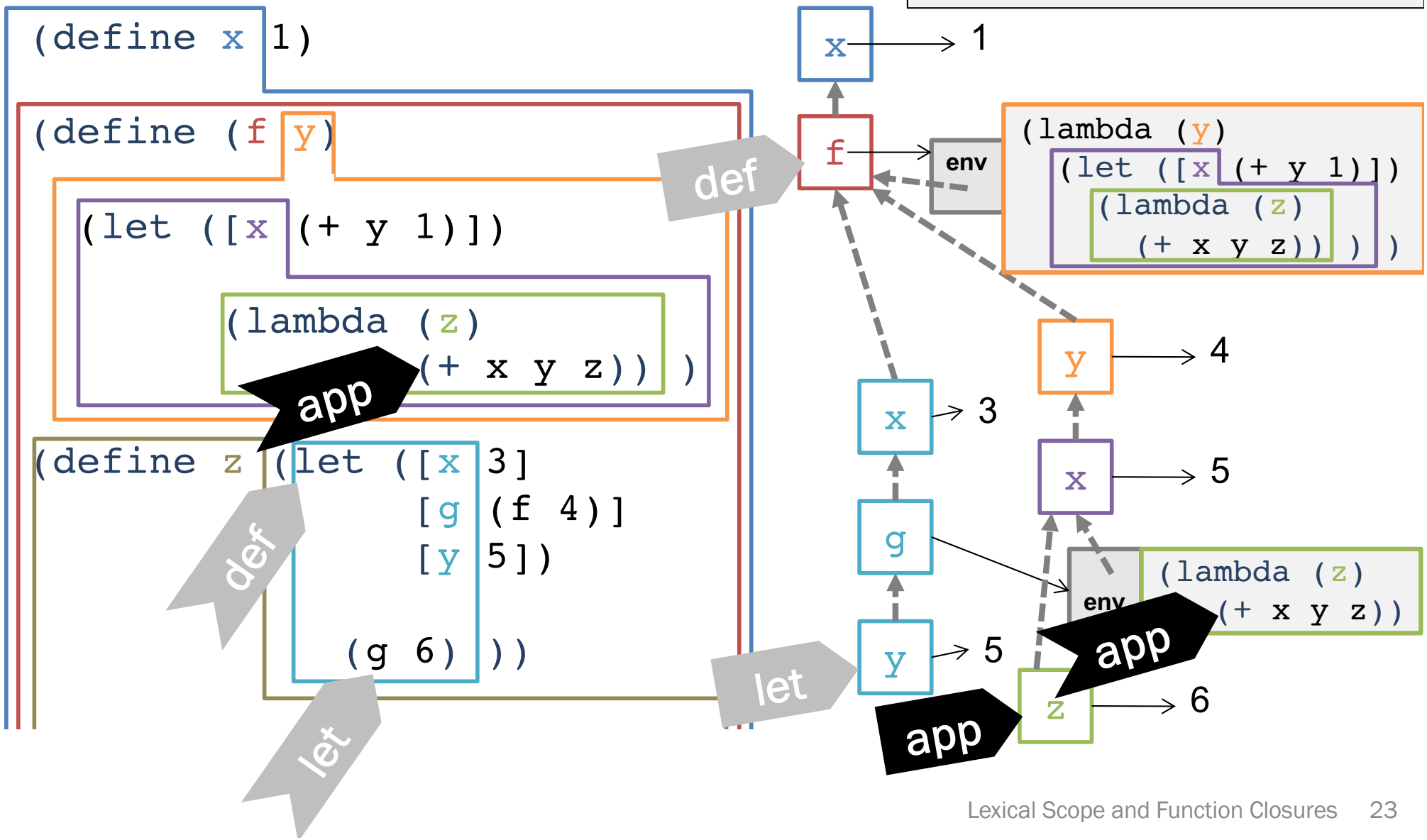
Example: returning a function



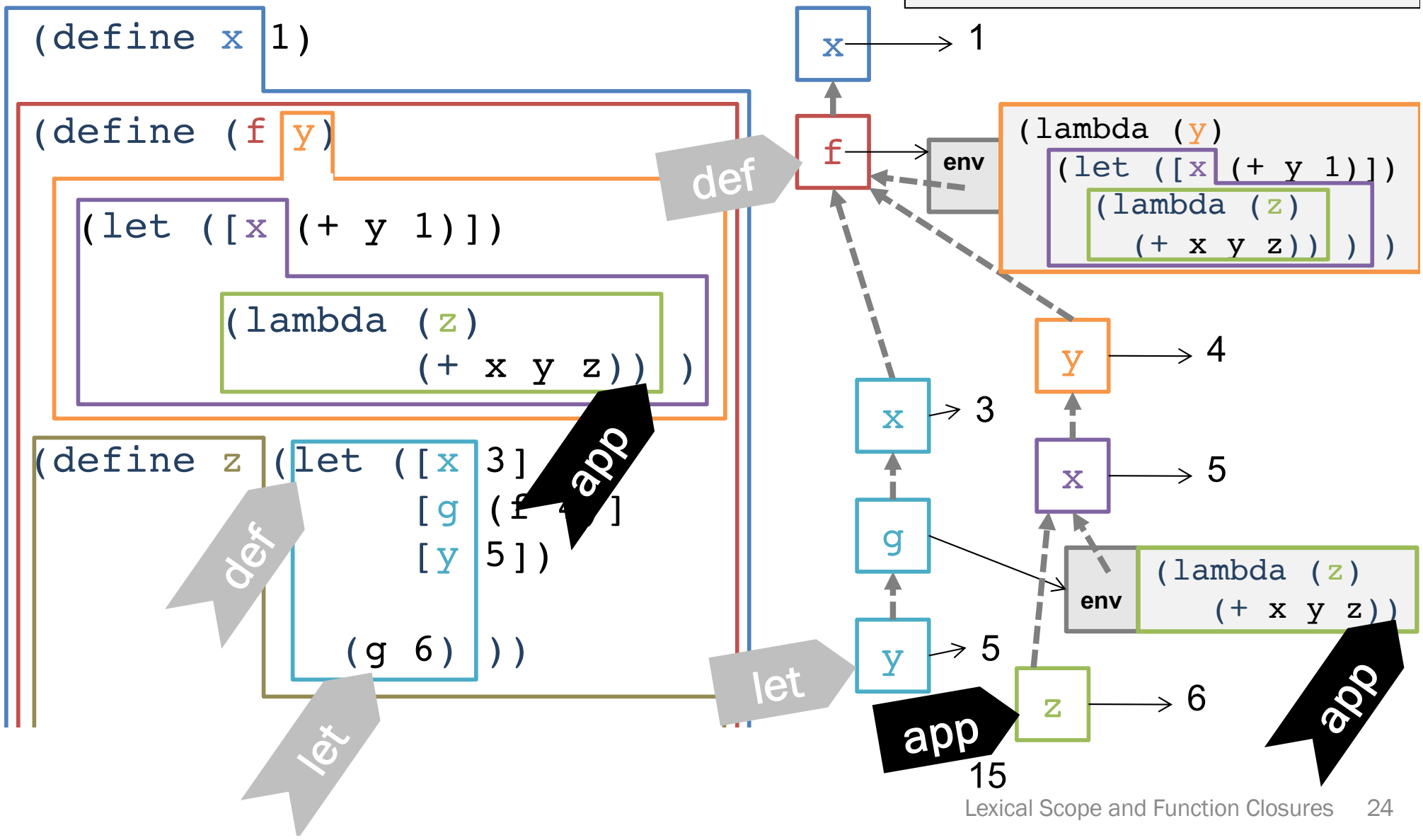
Example: returning a function



Example: returning a function



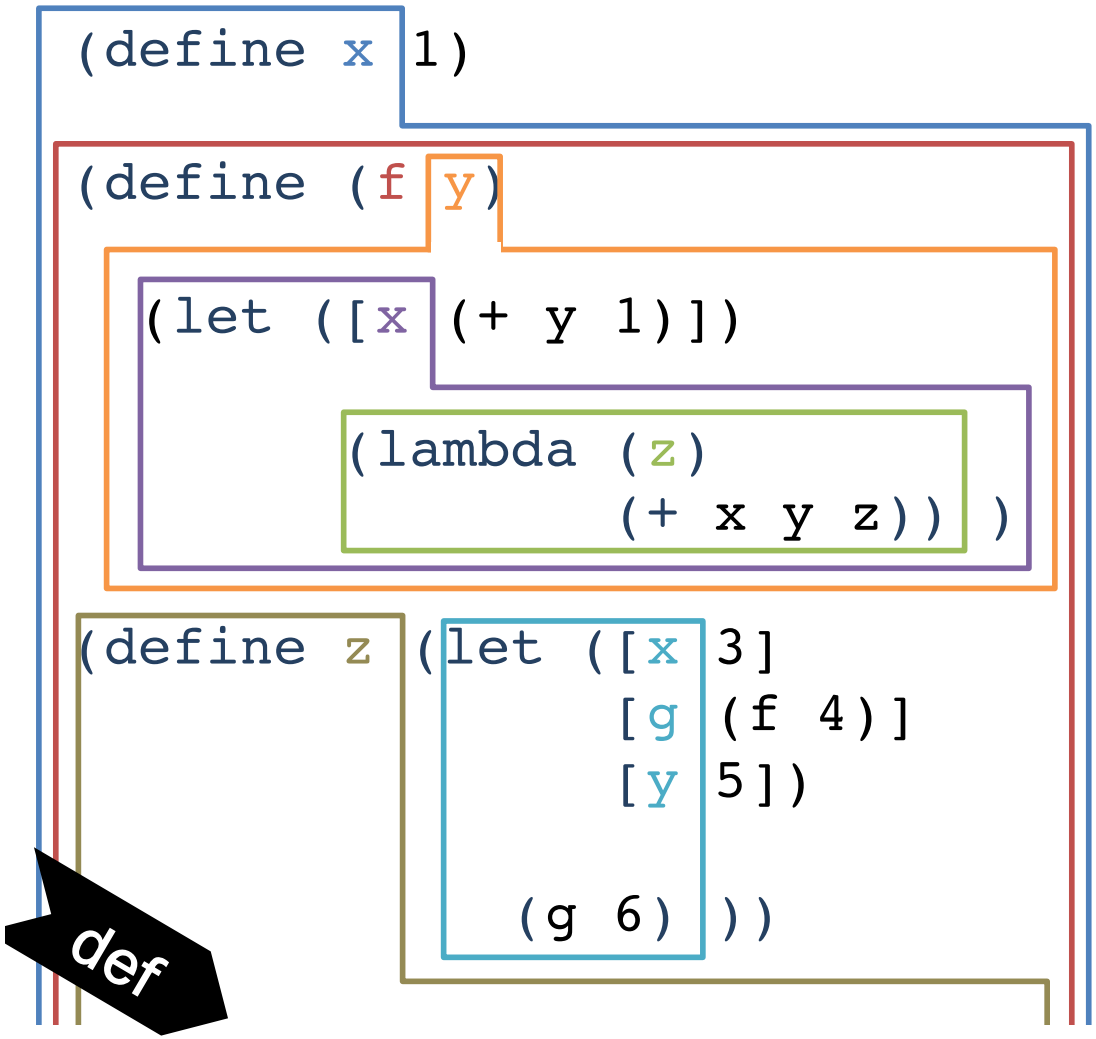
Example: returning a function

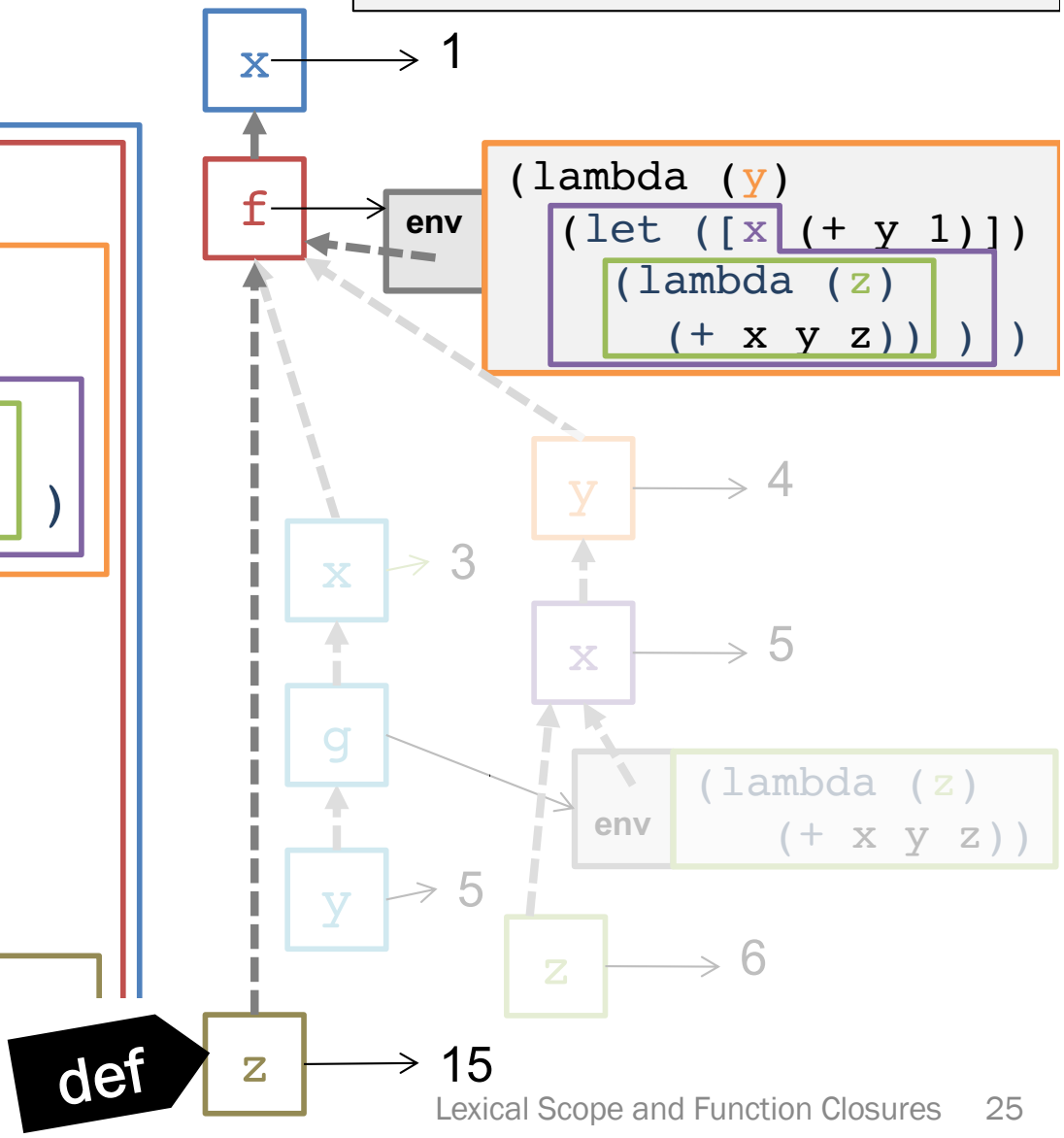


Example: returning a function

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:

- Under 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.

```
(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 during `all-shorter`…?

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 function call's **arguments are evaluated before the called function's body.**

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 definitions of the Racket language (subset):

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

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