Programming Languages

Lexical Scope and Function Closures

Adapted from Dan Grossman's PL class,
U. of Washington

Very important concept

- We know that the body of a function can refer to non-local variables
  - i.e., variables that are not explicitly defined in that function or passed in as arguments
- So how does a language know where to find values of non-local variables?

Look where the function was defined

(not where it was called)

- There are lots of good reasons for this semantics
  - Discussed after explaining what the semantics is
- For HW, exams, and competent programming, you must "get this"
- This concept is called *lexical scope (sometimes also called static scope)*

Example

```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```

- Line 2 defines a function that, when called, evaluates body **`(+ x y)`** in environment where **`x`** maps to **`1`** and **`y`** maps to the argument
- Call on line 4:
  - Creates a *new* environment where x maps to 2.
  - Looks up **`f`** to get the function defined on line 2.
  - Evaluates **`(+ x y)`** in the new environment, producing **`6`**
  - Calls the function, which evaluates the body in the old environment, producing **`7`**

Closures

How can functions be evaluated in old environments?

- The language implementation keeps them around as necessary

Can define the semantics of functions as follows:

- A function value has two parts
  - The code (obviously)
  - The environment that was current when the function was defined
- This value is called a *function closure* or just *closure*.
- When a function **`f`** is called, f's code is evaluated in the environment pointed to by **`f`**'s environment pointer.
  - (The environment is first extended with extra bindings for the values of **`f`**'s arguments.)

Example

```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```

- Line 2 creates a closure and binds **`f`** to it:
  - Code: "take argument **`y`** and have body **`(+ x y)`**"
  - Environment: "**`x`** maps to **`1`**"
    - (Plus whatever else has been previously defined, including **`f`** for recursion)

What's happening behind the scenes

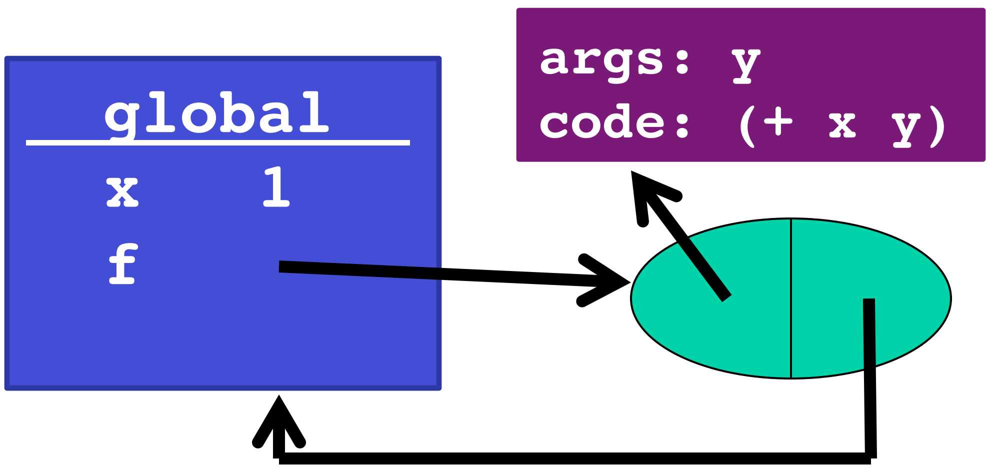
- An environment is stored using *frames*.
- A *frame* is a table that maps variables to values; a frame also may have a single pointer to another frame.
- When a variable is asked to be looked up in an "environment," the lookup always starts in some frame.
- If the variable is not found in that frame, the search continues wherever the frame points to (another frame).
- If the search ever gets to a frame without a pointer to another frame (usually this is the "global" or "top-level" frame), we report an error that the variable is undefined.

```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```

```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```

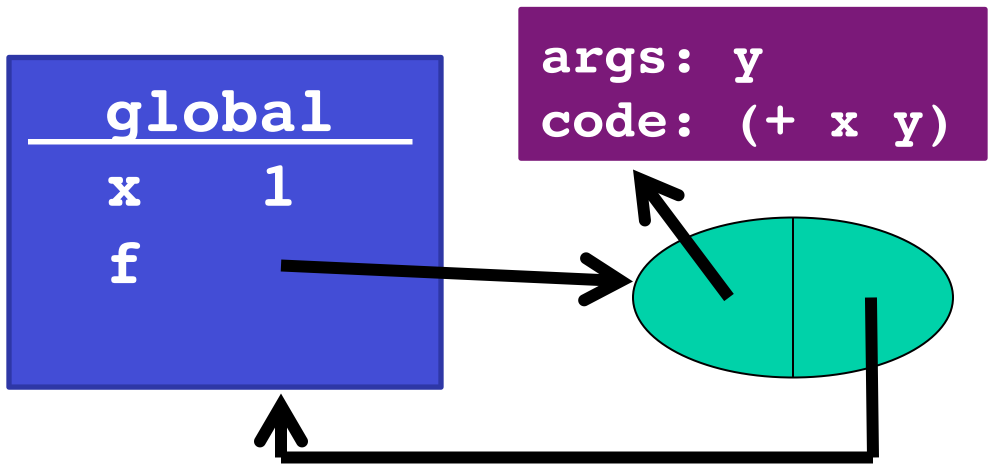
global	
x	1

```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```



Rules for frames and environments

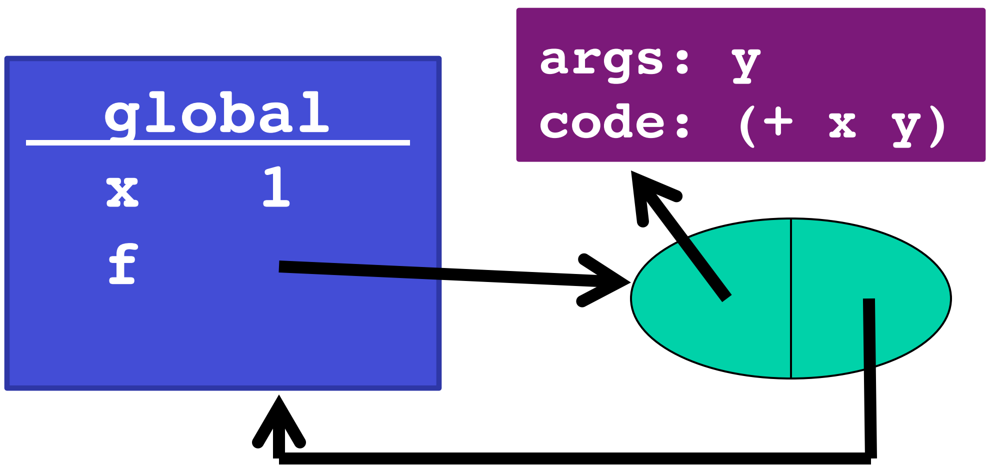
- Rule 1:
  - Every function **definition** (including anonymous function definitions) creates a closure where
    - the code part of the closure points to the function's code
    - the environment part of the closure points to the frame that was current when the function was defined (the frame we are currently using to look up variables)



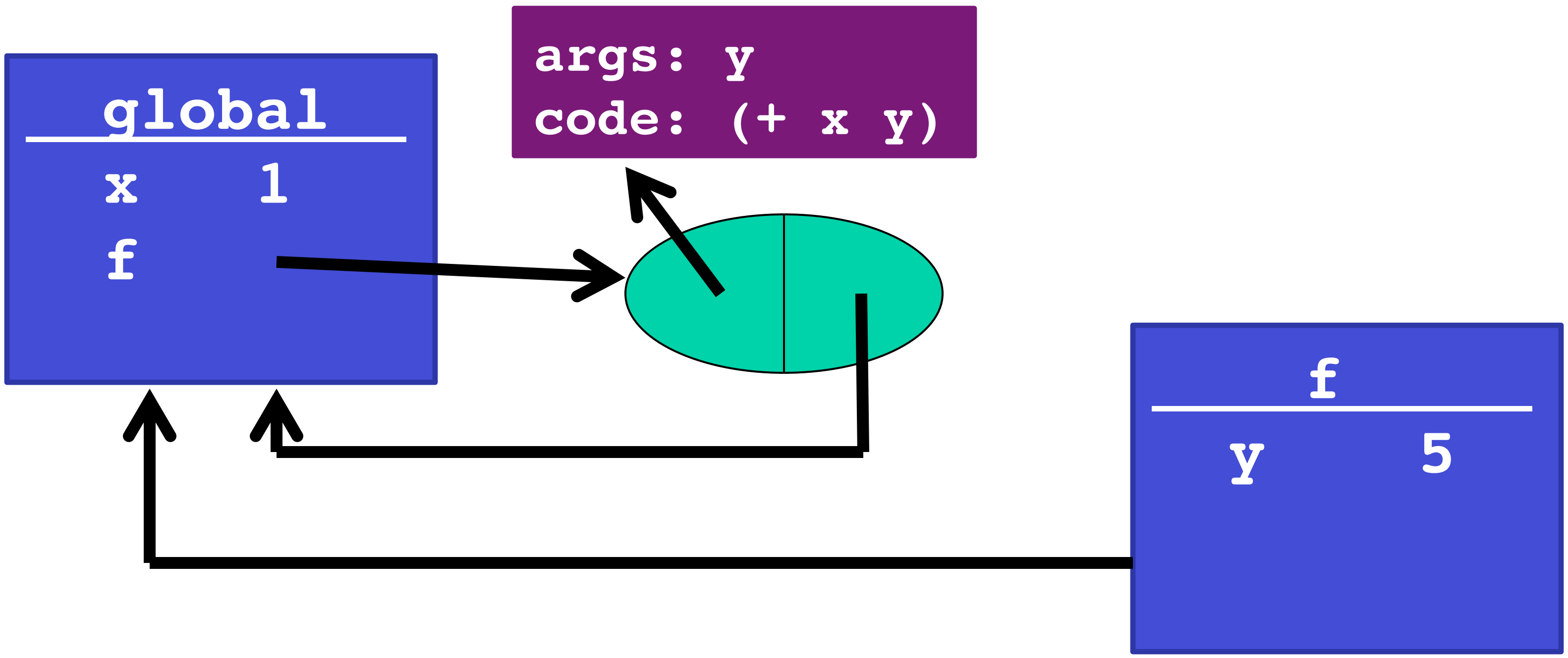
Rules for frames and environments

- Rule 2:
  - Every function **call** creates a new frame consisting of the following:
    - the new frame's table has bindings for all of the function's arguments and their corresponding values
    - the new frame's pointer points to the same environment that f's environment pointer points to.

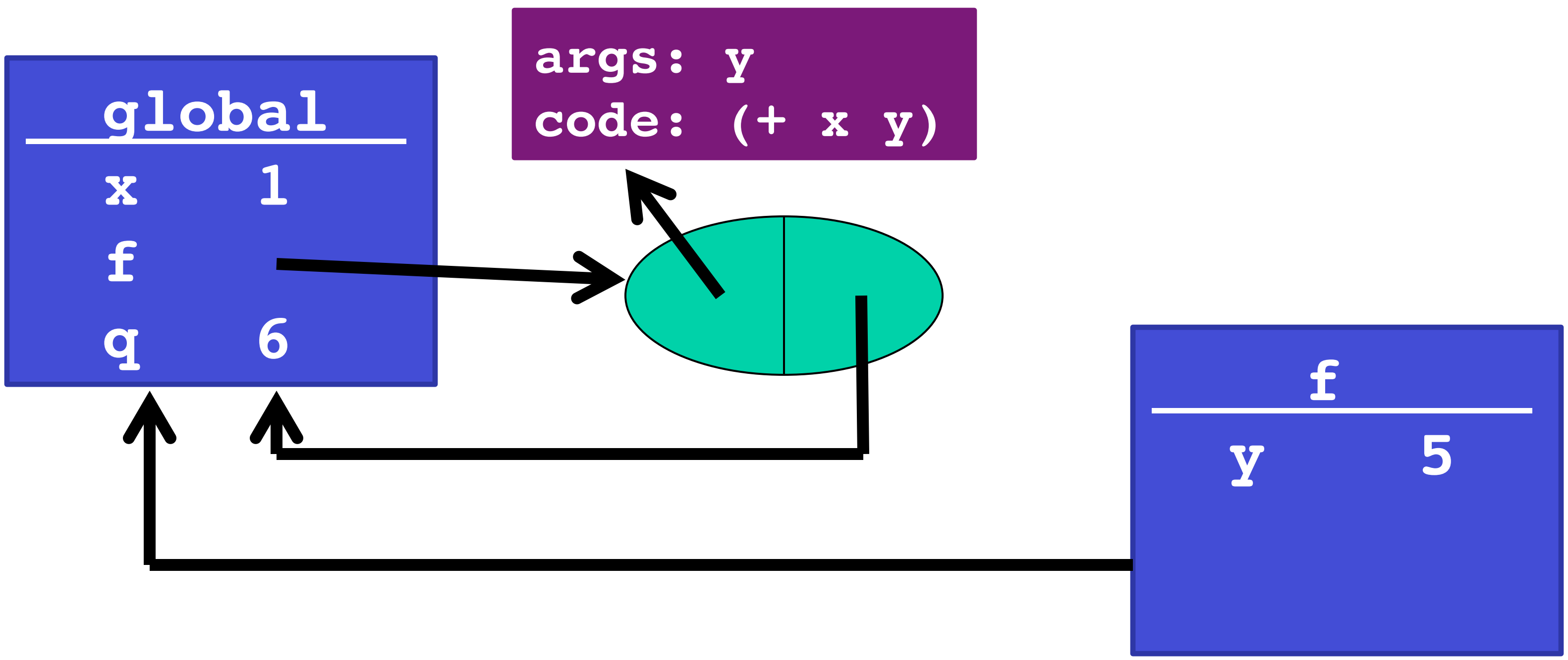
```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define q (f 5))   ; changed this line
```



```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define q (f 5))   ; changed this line
```



```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define q (f 5))   ; changed this line
```



So what?

Now you know the rules. Next steps:

- (Silly) examples to demonstrate how the rule works for higher-order functions
- Why the other natural rule, *dynamic scope*, is a bad idea
- Powerful idioms with higher-order functions that use this rule
  - This lecture: Passing functions to functions like **`filter`**
  - Next lecture: Several more idioms

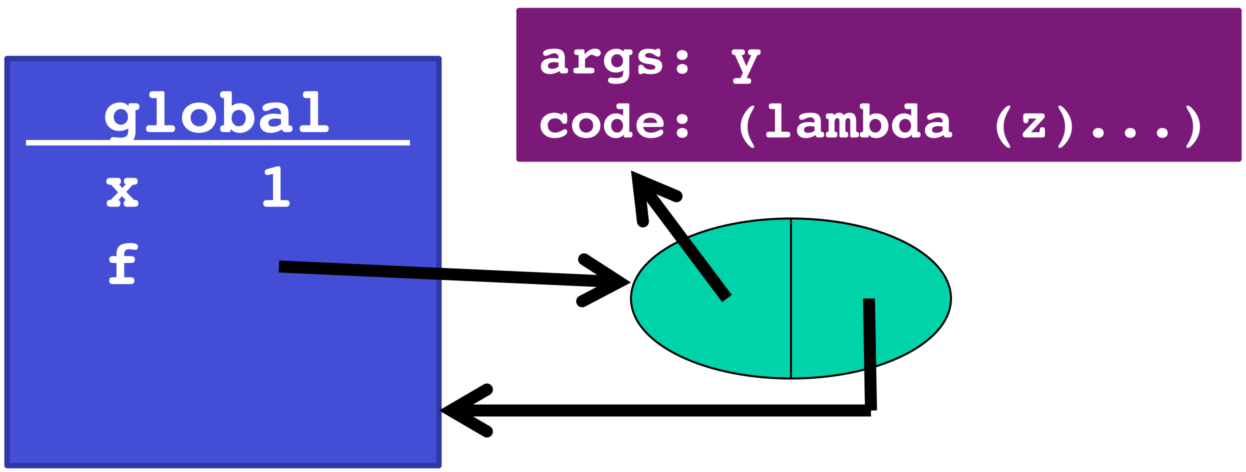
Example: Returning a function

```
1   (define x 1)
2   (define (f y) (lambda (z) (+ x y z)))
3   (define g (f 4))
4   (define z (g 6))
```

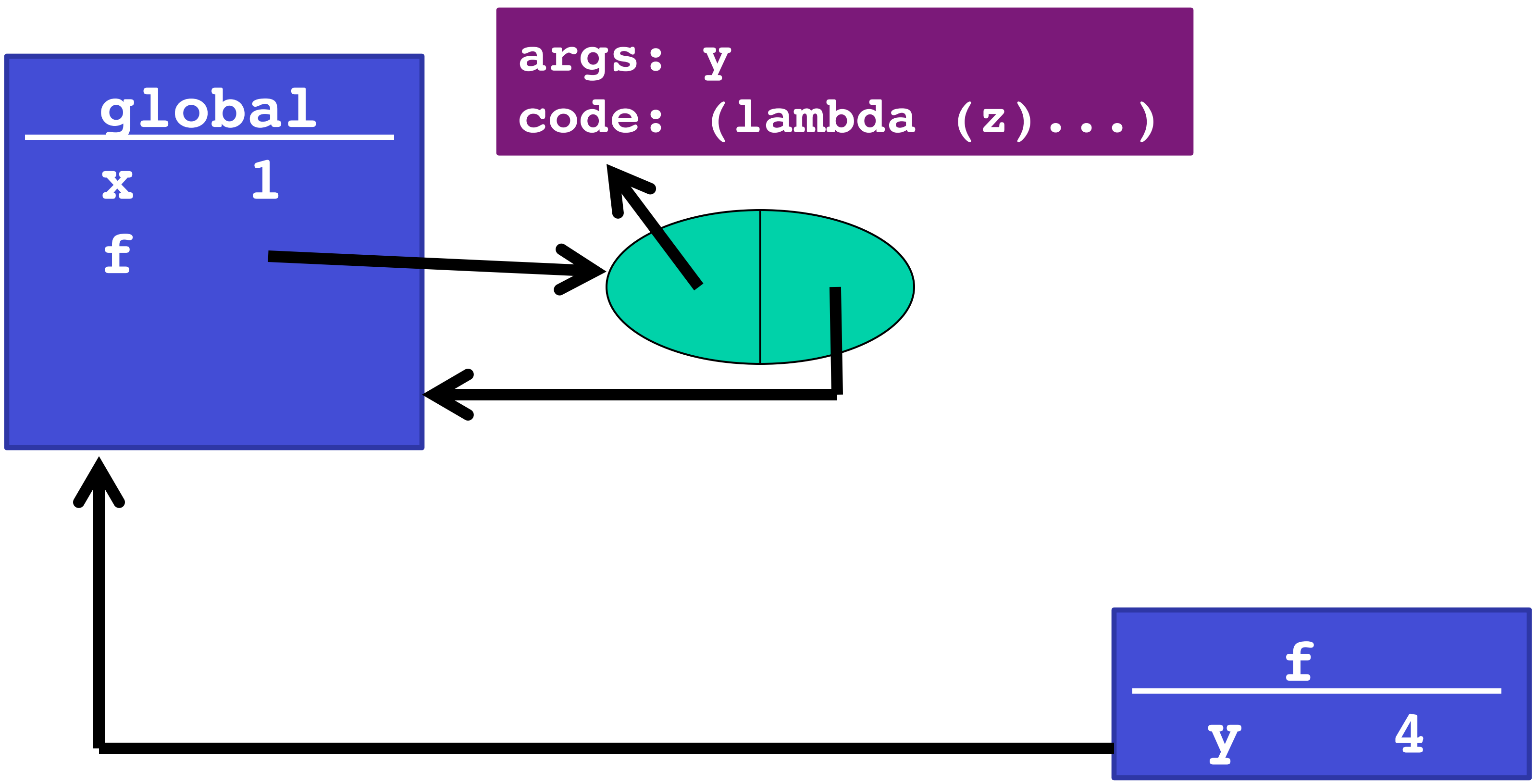
- Trust the rules:
  - Evaluating line 2 binds f to a closure.
  - Evaluating line 3 binds g to a closure as well.
    - New frame is created for the call to f.
  - Evaluating line 4 binds z to a number.
    - New frame is created for the call to g.

```
(define x 1)
(define (f y) (lambda (z) (+ x y z)))
(define g (f 4))
(define z (g 6))
```

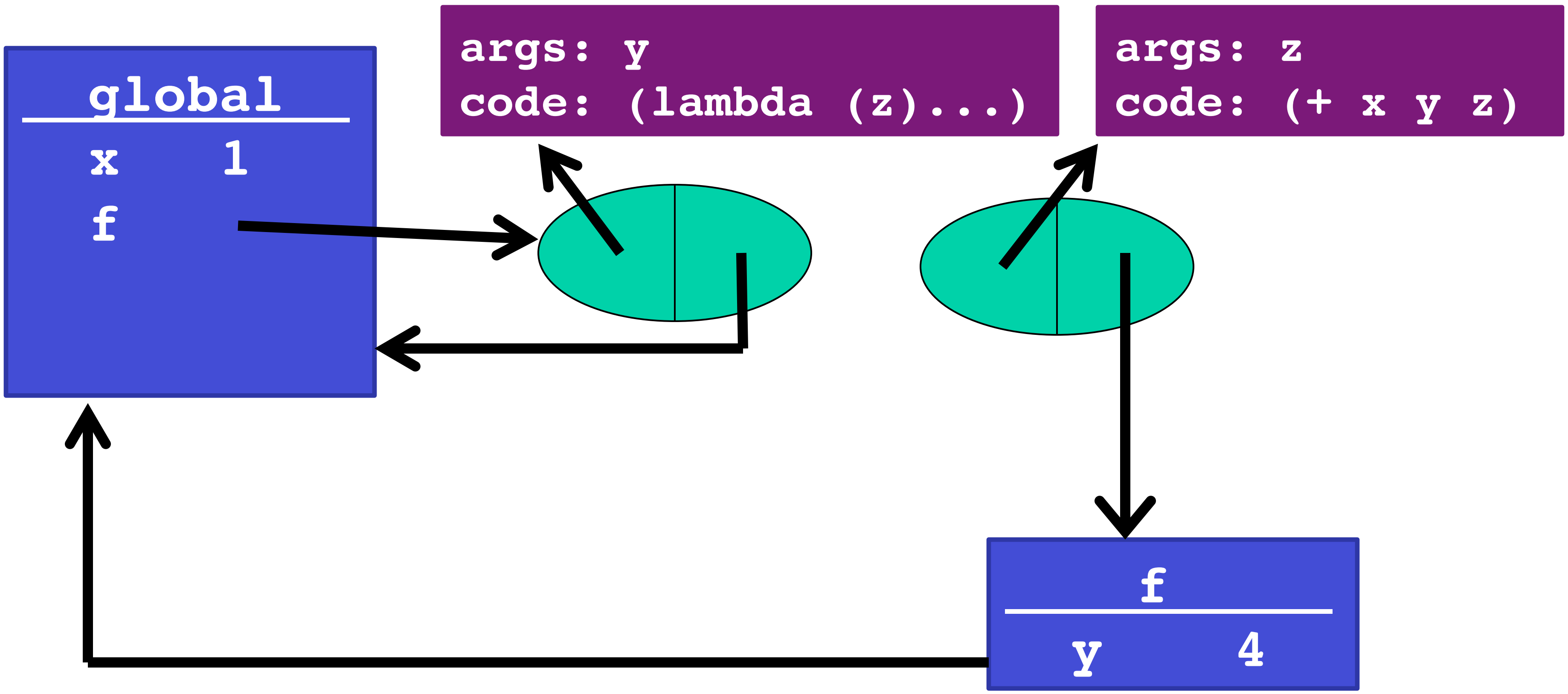
```
(define x 1)
(define (f y) (lambda (z) (+ x y z)))
(define g (f 4))
(define z (g 6))
```



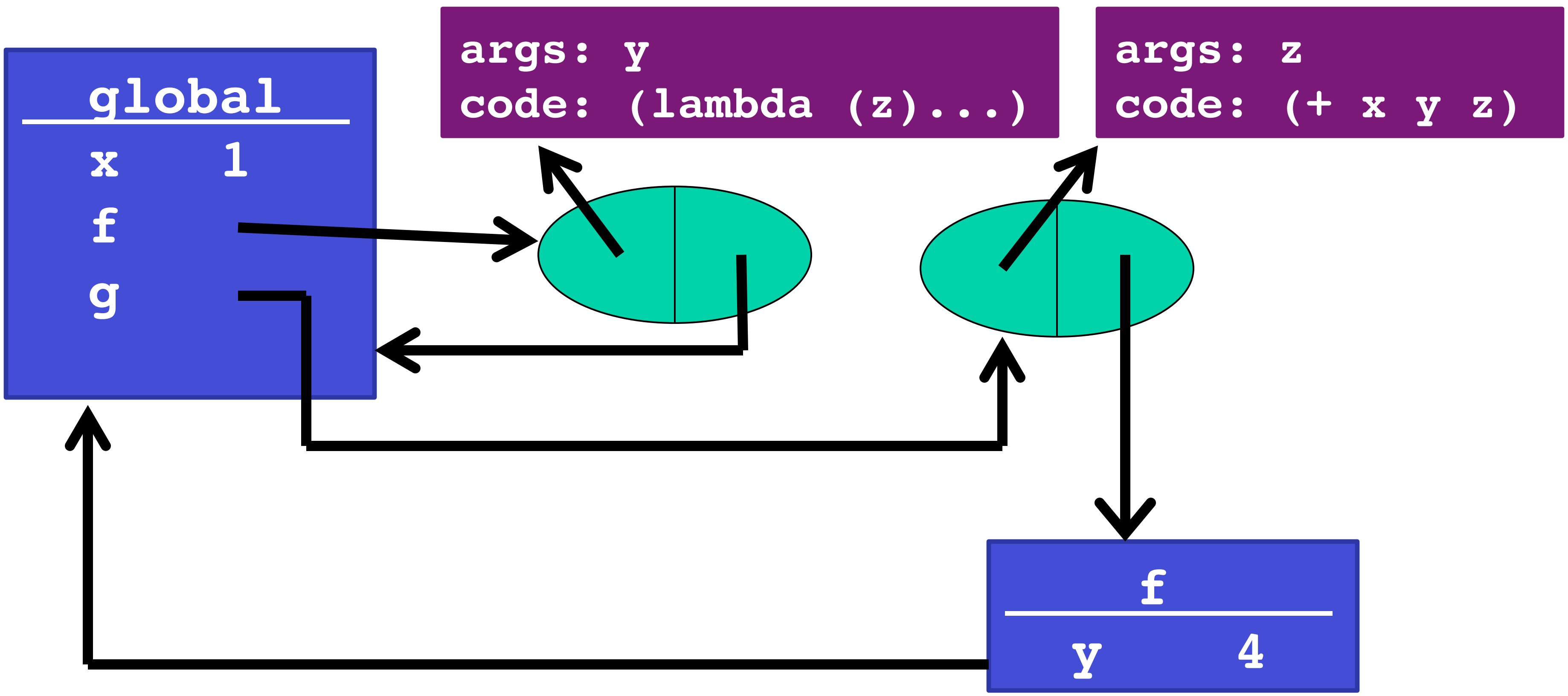
```scheme
(define x 1)
(define (f y) (lambda (z) (+ x y z)))
(define g (f 4))
(define z (g 6))
```



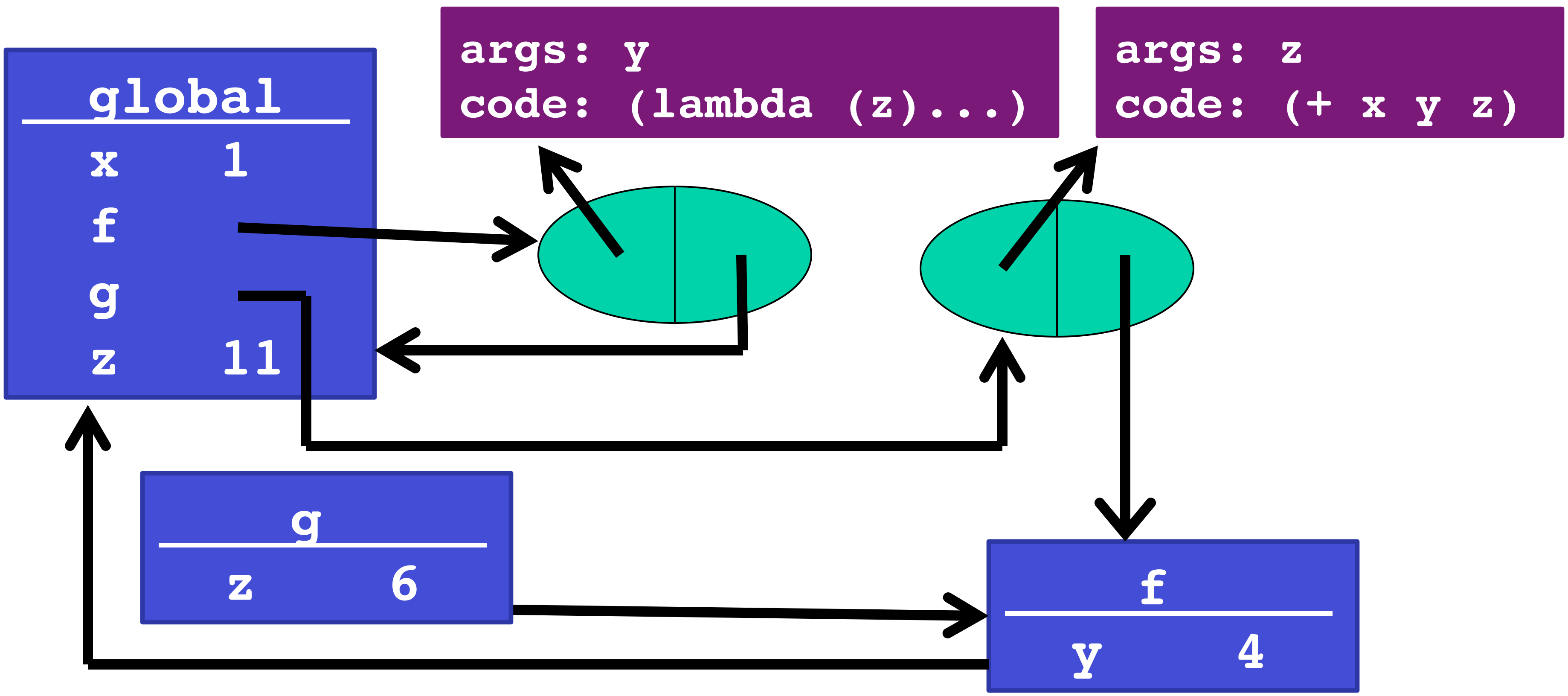
```
(define x 1)
(define (f y) (lambda (z) (+ x y z)))
(define g (f 4))
(define z (g 6))
```



```
(define x 1)
(define (f y) (lambda (z) (+ x y z)))
(define g (f 4))
(define z (g 6))
```



```
1   (define x 1)
2   (define (f y) (lambda (z) (+ x y z)))
3   (define g (f 4))
4   (define z (g 6))
```



Rules for frames and environments

- Rule 2a:
  - Every evaluation of a "let" expression creates a new frame as follows:
    - the new frame's table has bindings for all of the let expressions variables and their corresponding values
    - the new frame's pointer points to the frame where the let expression was defined

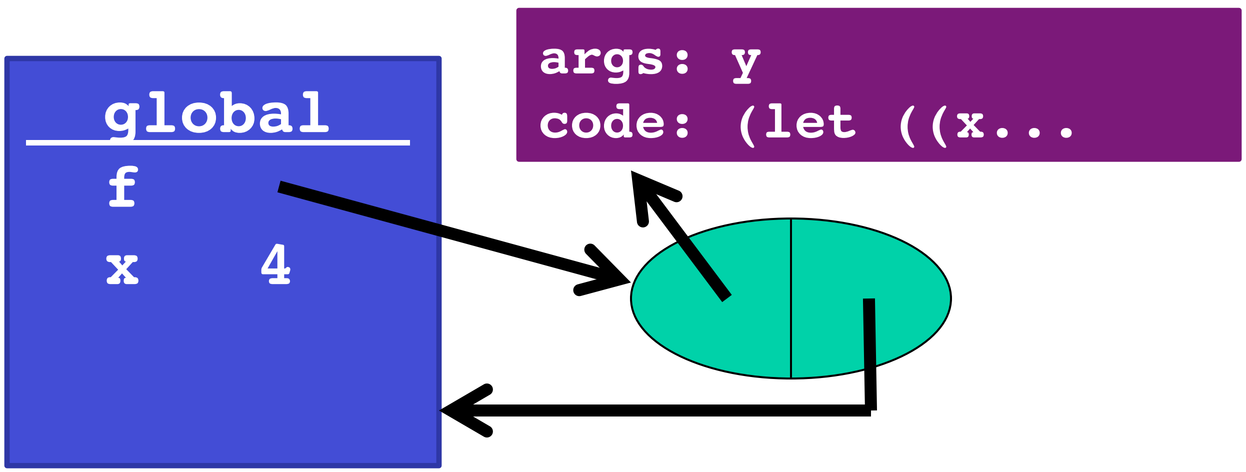
Example: Passing a function

```
1  (define (f g) (let ((x 3)) (g 2)))
2  (define x 4)
3  (define (h y) (+ x y z))
4  (define z (f h))
```

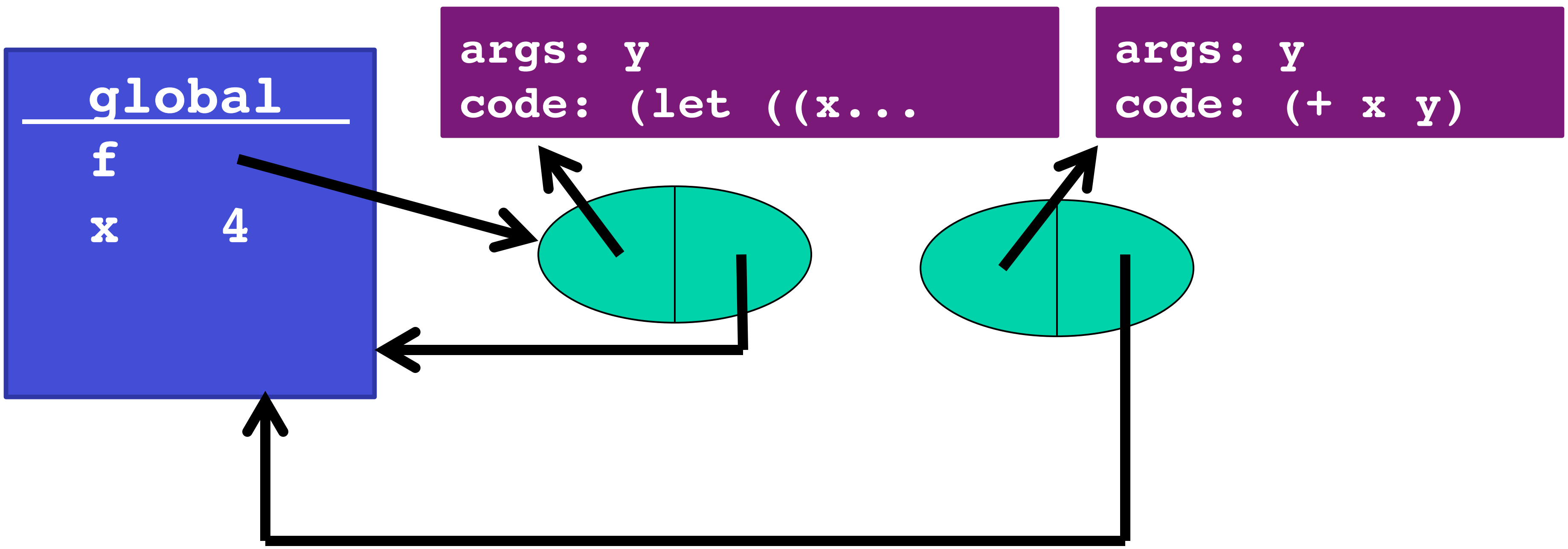
- Trust the rules:
  - Evaluating line 1 binds f to a closure.
  - Evaluating line 2 binds x to 4.
  - Evaluating line 3 binds h to a closure.
  - Evaluating line 4 binds z to a number.
    - First, calls f (creates new frame), then evaluates "let" (creates a new frame), then calls g (creates a new frame).

```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```

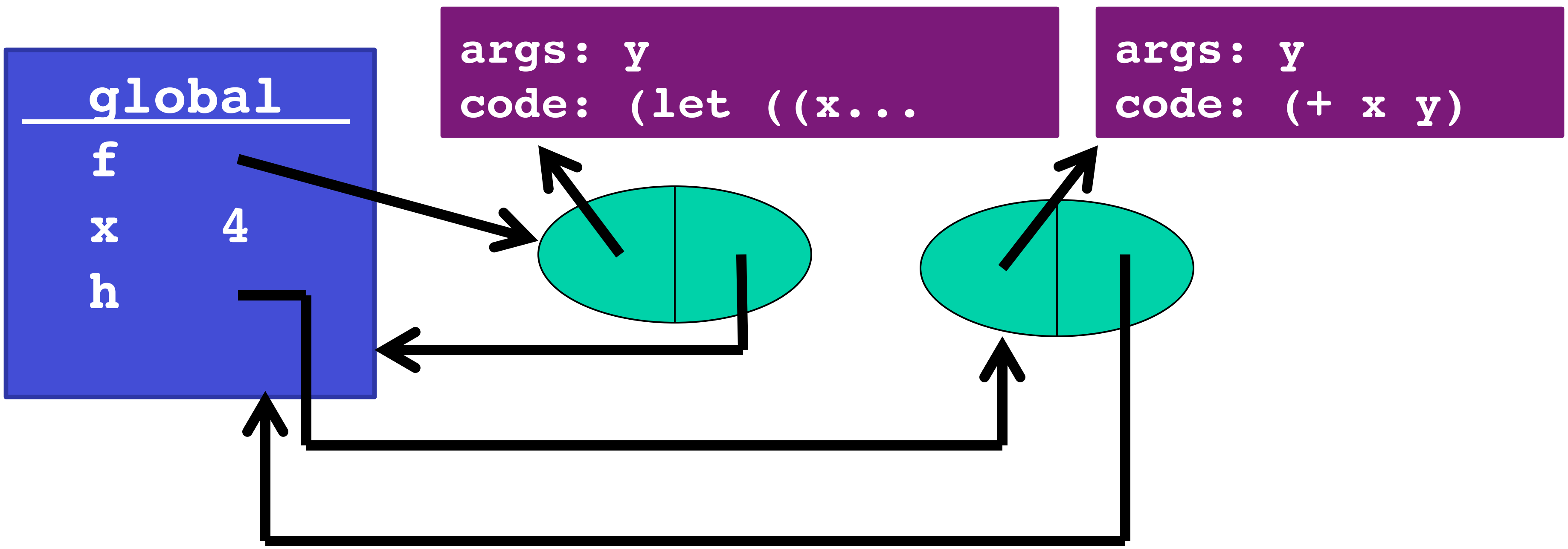
```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



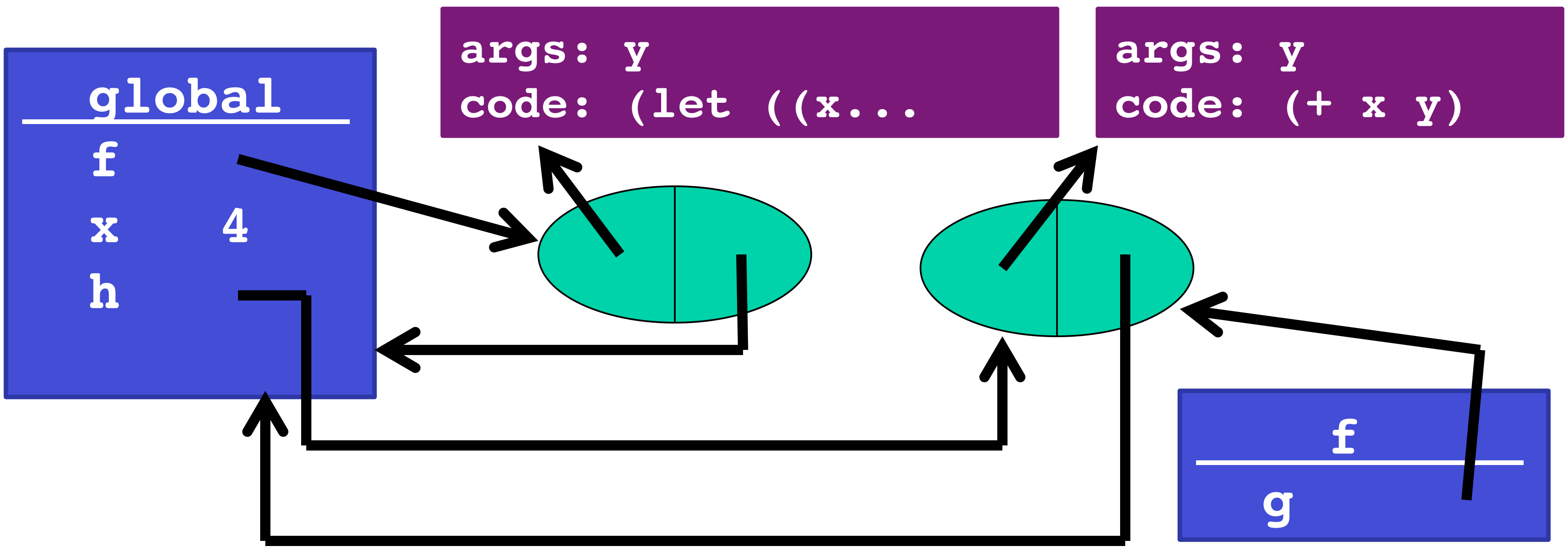
```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



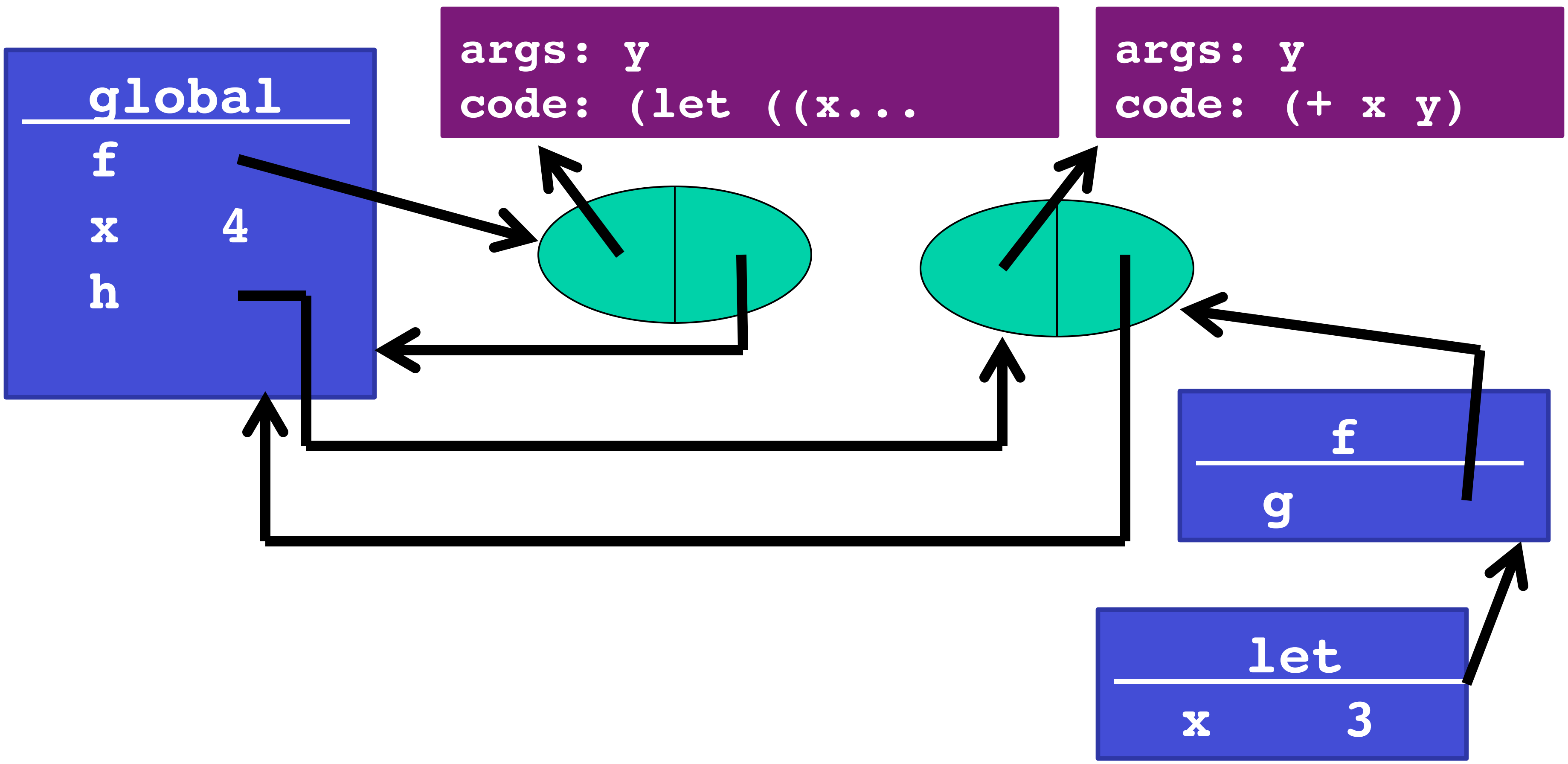
```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



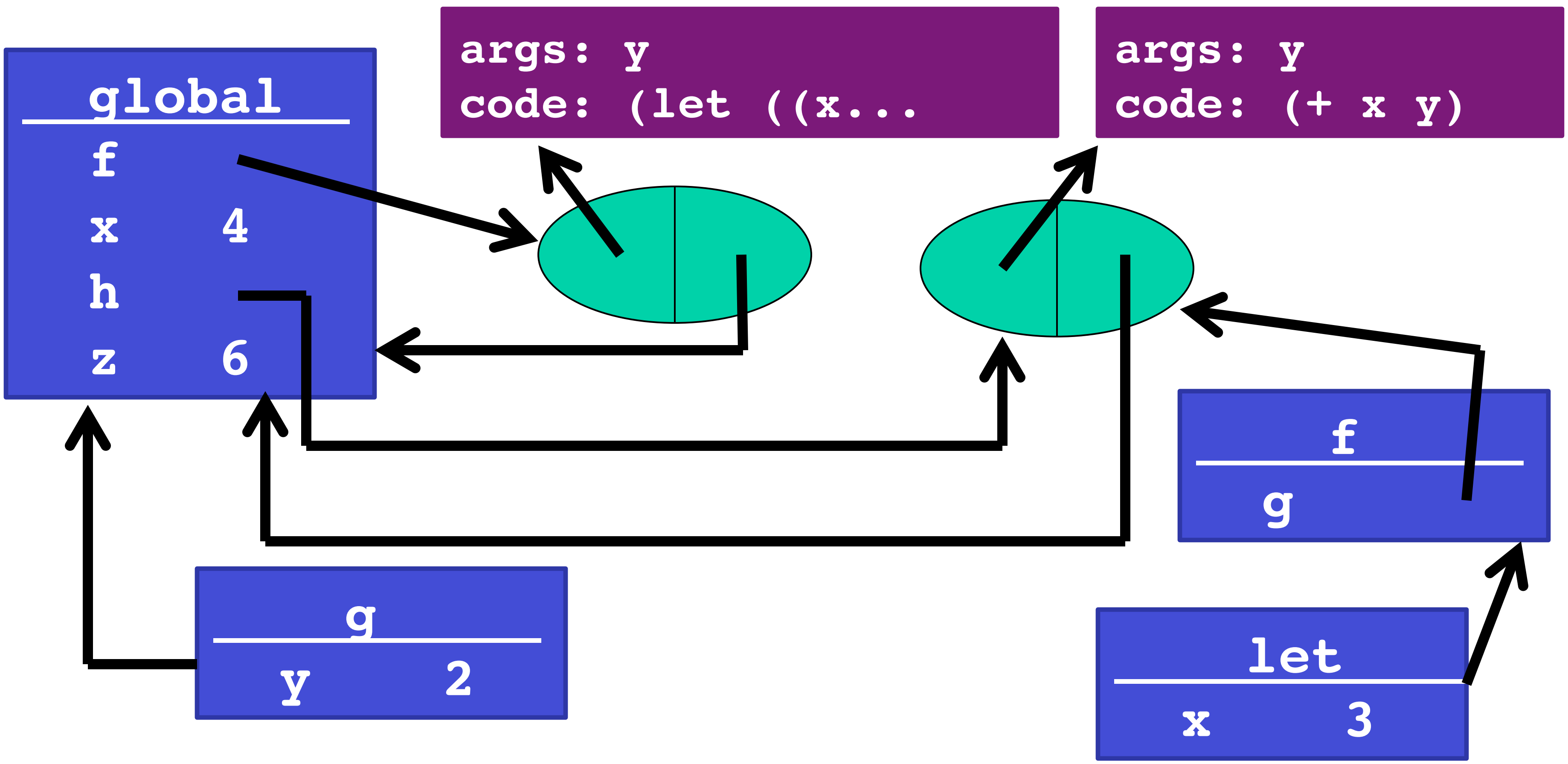
```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



Lexical scoping vs dynamic scoping

- The alternative to lexical scoping is called dynamic scoping.
- In dynamic scoping, if a function f references a non-local variable x, the language will look for x in the function that **called** f.
  - If it's not found, will look in the function that called the function that called f (and so on).
- Contrast with lexical scoping, where the language would look for x in the scope where f was **defined**.

Why lexical scope?

1. **Function meaning does not depend on variable names used**

Example: Can change body of a function to use **`q`** instead of **`x`**

- Lexical scope: it can't matter
- Dynamic scope: Depends how result is used

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

When the anonymous function that f returns is called, in lexical scoping, we always know where the values of x, y, and z will be (what frames they're in). With dynamic scoping, x and y will be searched for in the functions that called the anonymous function, so who knows where they'll be.

Why lexical scope?

1. **Function meaning does not depend on variable names used**

Example: Can remove unused variables

- Dynamic scope: But maybe some **g** uses it (weird)

```
(define (f g)
  (let ((x 3))
    (g 2)))
```

- You would never write this in a lexically-scoped language, because the binding of x to 3 is never used.
  - (No way for g to access this particular binding of x.)
- In a dynamically-scoped language, g might refer to a non-local variable x, and this binding might be necessary.

Why lexical scope?

2. Easy to reason about functions where they're defined.

Example: Dynamic scope tries to add a string to a number (b/c in the call to (+ x y), x will be "hello")

```
(define x 1)
(define (f y)
     (+ x y))
(define g
   (let ((x "hello"))
     (f 4))
```

Why lexical scope?

3. Closures can easily store the data they need
   – Many more examples and idioms to come

```
(define (gteq x) (lambda (y) (>= y x)))
(define (no-negs lst) (filter (gteq 0) lst))
```

- The anonymous function returned by gteq references a non-local variable x.
- In lexical scoping, the closure created for the anonymous function will point to gteq's frame so x can be found.
- In dynamic scoping, x would not be found at all.

Does dynamic scope exist?

- Lexical scope for variables is definitely the right default
  - Very common across languages
- Dynamic scope is occasionally convenient in some situations
  - So some languages (e.g., Racket) have special ways to do it
  - But most don't bother
- Historically, dynamic scoping was used more frequently in older languages because it's easier to implement than lexical scoping.
  - Strategy: Just search through the call stack until variable is found. No closures needed.
  - Call stack maintains list of functions that are currently being called, so might as well use it to find non-local variables.

Iterators made better

- Functions like **`map`** and **`filter`** are *much* more powerful thanks to closures and lexical scope

- Function passed in can use any "private" data in its environment

- Iterator (e.g., map or filter) "doesn't even know the data is there"
  - It just calls the function that it's passed, and that function will take care of everything.

Review of foldr

`foldr` (sometimes also called accumulate, reduce, or inject) is another very famous iterator over recursive structures

Accumulates an answer by repeatedly applying **`f`** to answer so far

- **`(foldr f base (x1 x2 x3 x4))`** computes **`(f x1 (f x2 (f x3 (f x4 base))))`**

```
(define (foldr f base lst)
  (if (null? lst) base
    (f (car lst)
       (foldr f base (cdr lst)))))
```

- This version “folds right”; another version “folds left”
- Whether the direction matters depends on **`f`** (often not)

Examples with foldr

These are useful and do not use "private data"

```
(define (f1 lst) (foldr + 0 lst))
(define (f2 lst)
  (foldr (lambda (x y) (and (>= x 0) y)) #t lst))
```

These are useful and do use "private data"

```
(define (f3 lo hi lst)
  (foldr (lambda (x y)
    (+ (if (and (>= x lo) (<= x hi)) 1 0) y))
  0 lst))

(define (f4 g lst)
  (foldr (lambda (x y) (and (g x) y)) #t lst))
```