CS 360
Programming Languages
Day 11 – Lexical Scope

What is scope?

- The ***scope*** of a variable is the region of a computer program where that variable can be used. (You know this.)

- Why do we care? (You may not know this.)

- Scoping rules of a programming language tell us:
  - How to find the value of a variable (aka ***name resolution***).
  - What to do when there are multiple variables with the same name in a program.

- Many scoping rules may seem "obvious" (because you've been programming for a while) but some are not.
  - And we'll also see how these rules are implemented under the hood of Racket (and other PLs).

Motivation for why you should care

```
(define x 5)
(define (add1 x) (+ x 1))
(define y (add1 7))
```

- What is the scope of each **x**?
  How does Racket keep the two versions of **x** separate?

```
(define (make-adder y)
  (lambda (x) (+ x y)))

(define add3 (make-adder 3))
(define add4 (make-adder 4))

(define z (add3 10))
(define w (add4 20))
```

- How does Racket keep the two versions of **y** separate?
  - And how are they available after they "go out of scope?"

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 (will explain later).

- Critically important to understand for HW, exams, and competent programming now and in the future.

- This concept is called ***lexical scope*** *(sometimes also called* ***static scope****).*

Another 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 passed in.
- 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 (first-class) 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 that was stored alongside that code when the closure was created.
  - (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 the variable **`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`** itself in case of recursion)

Behind the scenes: environments and frames

- You have probably drawn diagrams showing variables and their values.
  - Memory diagrams, recursion diagrams, environment diagrams, etc.
  - Most PLs implement these in similar ways during program execution.

- Today we're going to focus on how Racket does environment diagrams.

Behind the scenes: environments and frames

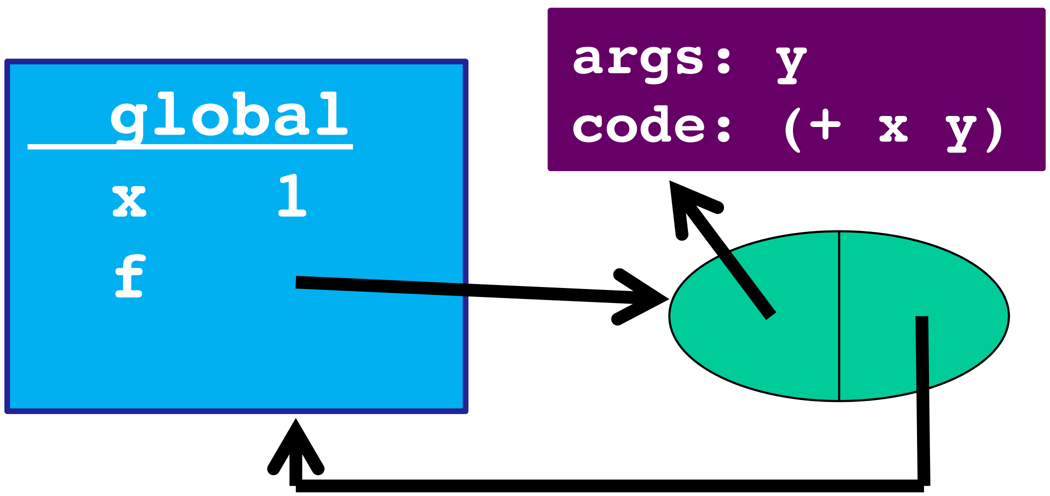
- An environment is represented using ***frames***.
- A ***frame*** is a table that maps variables to values.
  - Each frame (except the "global" or "top-level" frame) also has a pointer that always points another frame.

- 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 (the global frame) and the variable still isn't found, 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))))
```

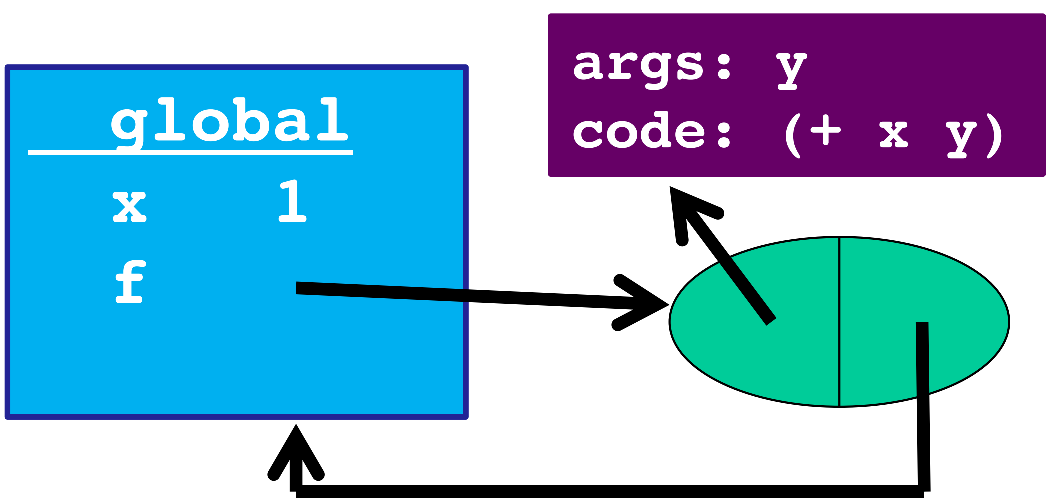
```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define q (f 5))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```

```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define q (f 5))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```



Rules for frames and environments

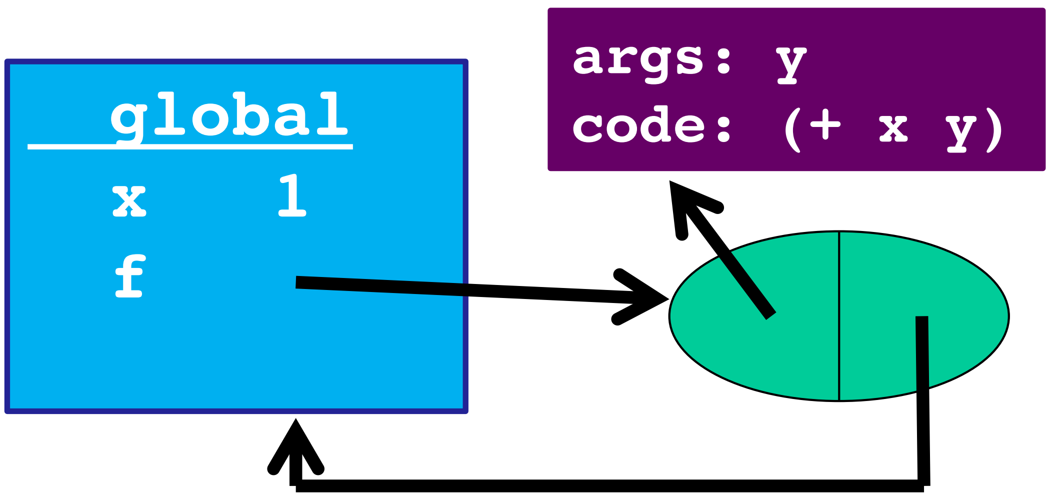
- Rule 1:
  - Every function **definition** (including anonymous function definitions) creates a closure where
    - the code part of the closure points to the function's code
    - the environment part of the closure points to the frame that was current when the function was defined (the frame we are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



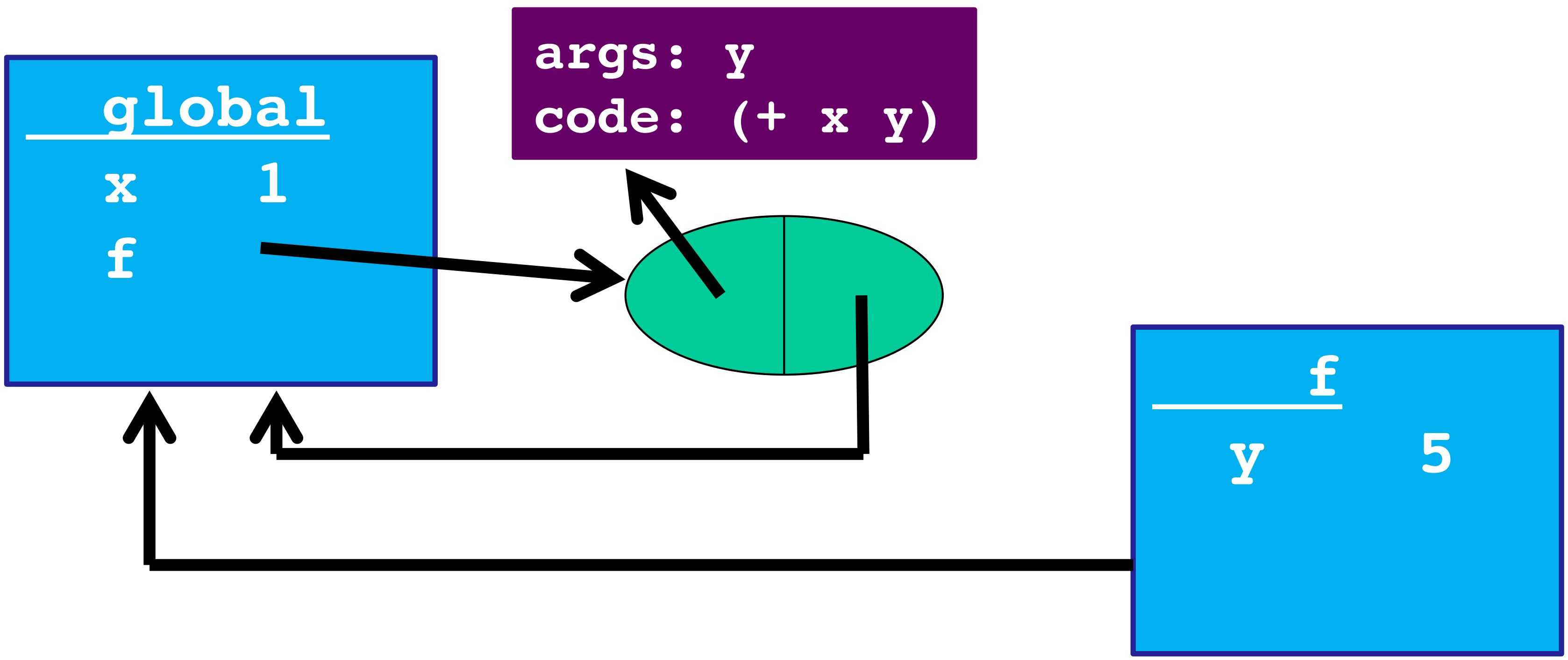
Rules for frames and environments

- 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.

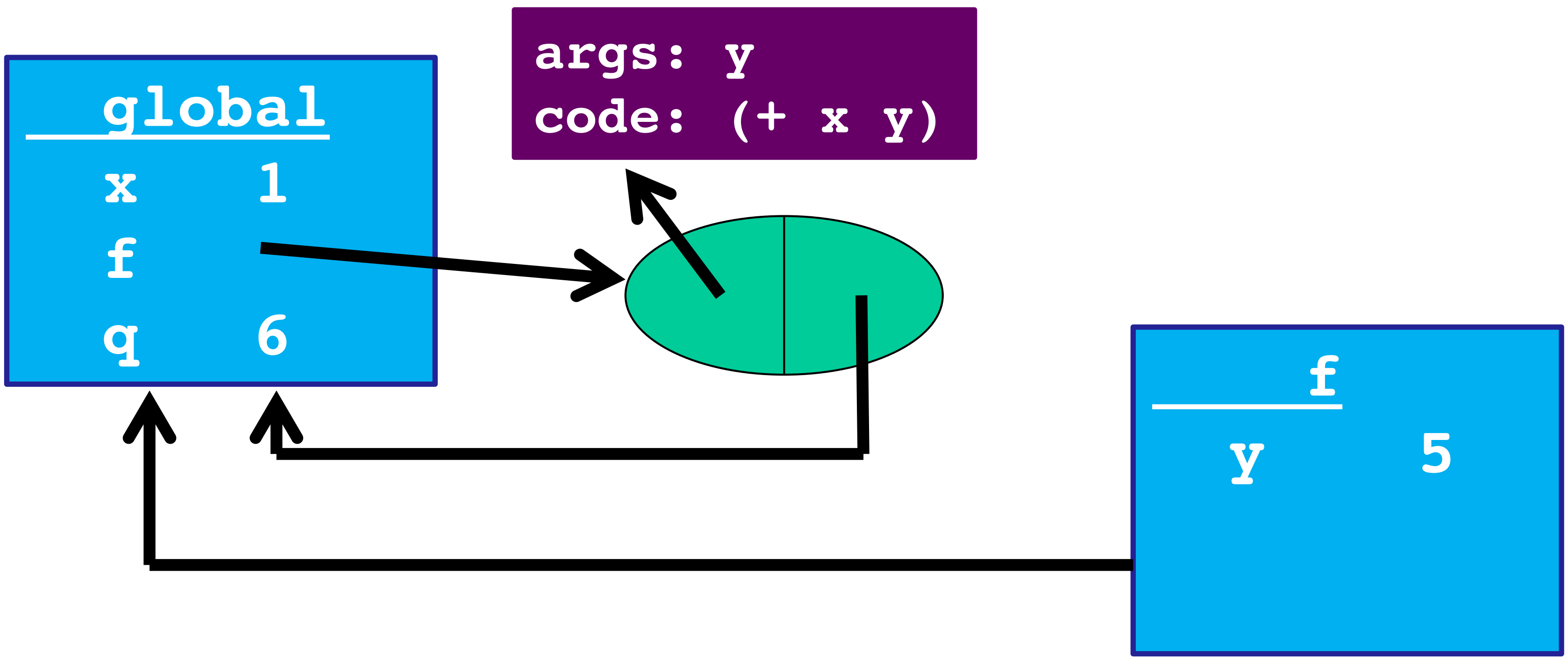
```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define q (f 5))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```



```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define q (f 5))
-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 q (f 5))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```



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

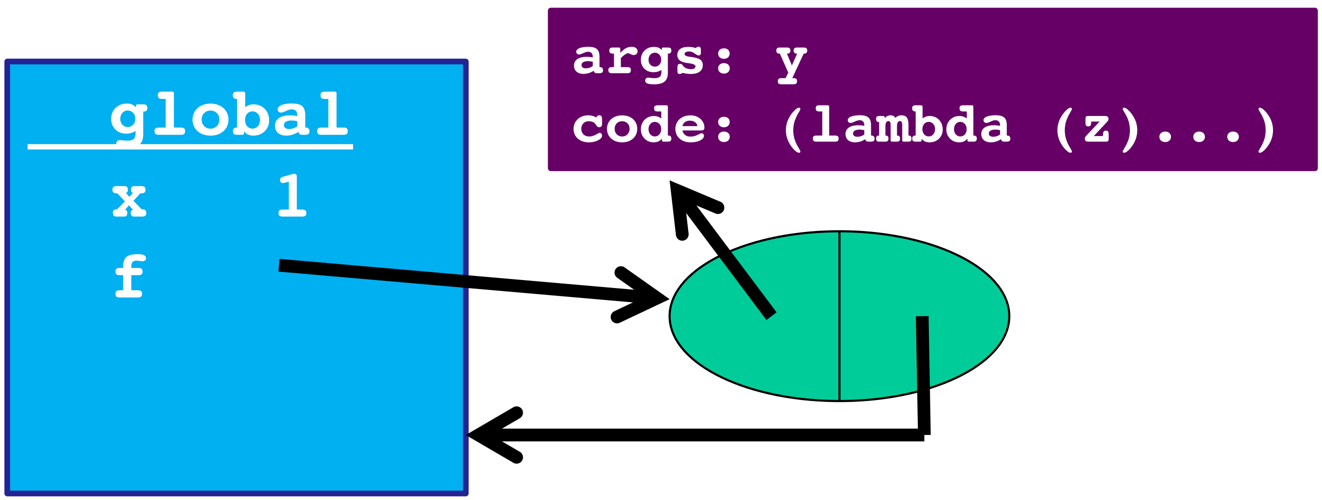
Example: Returning a function

- 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.

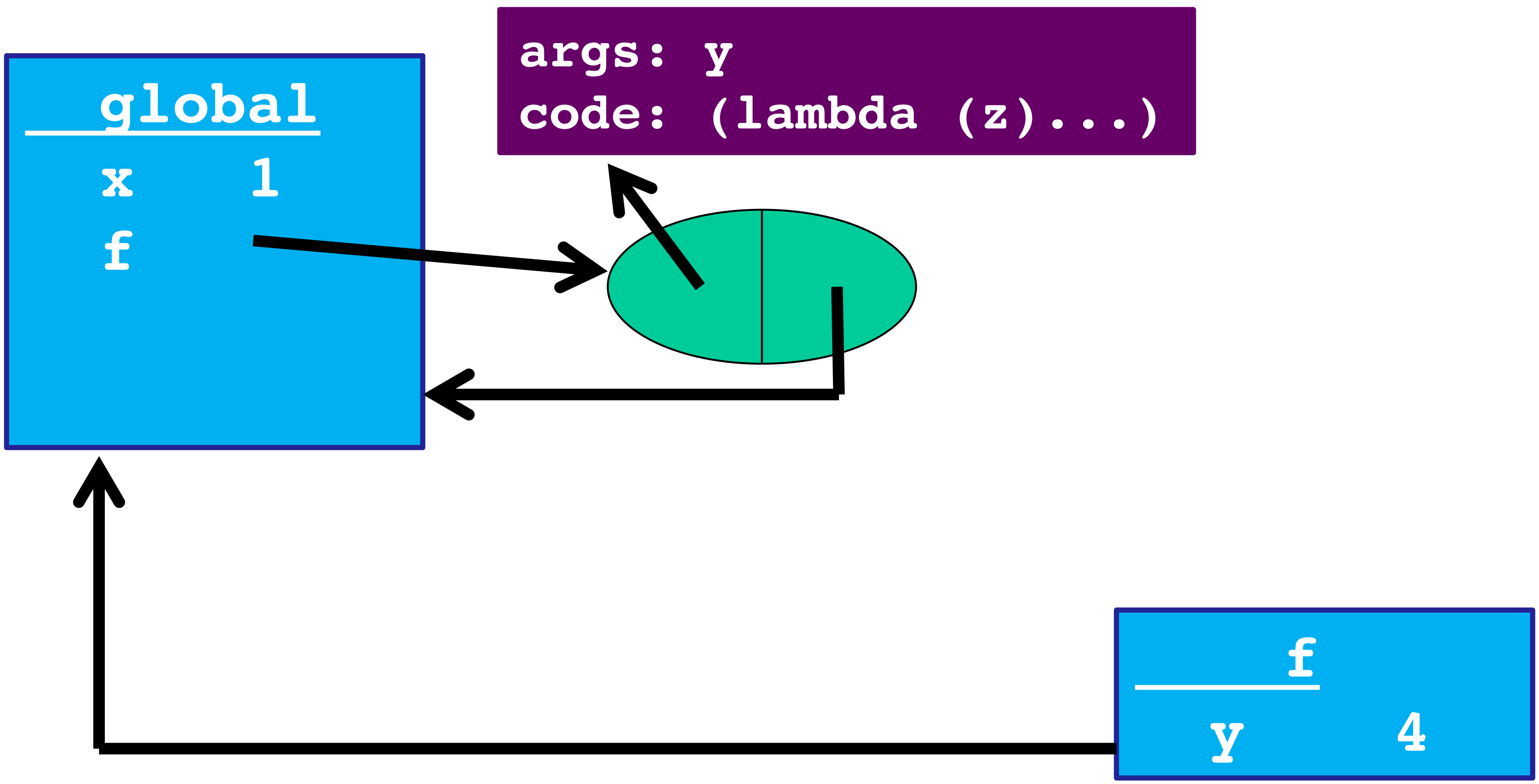
```
1  (define x 1)
2  (define (f y) (lambda (z) (+ x y z)))
3  (define g (f 4))
4  (define z (g 6))
```

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

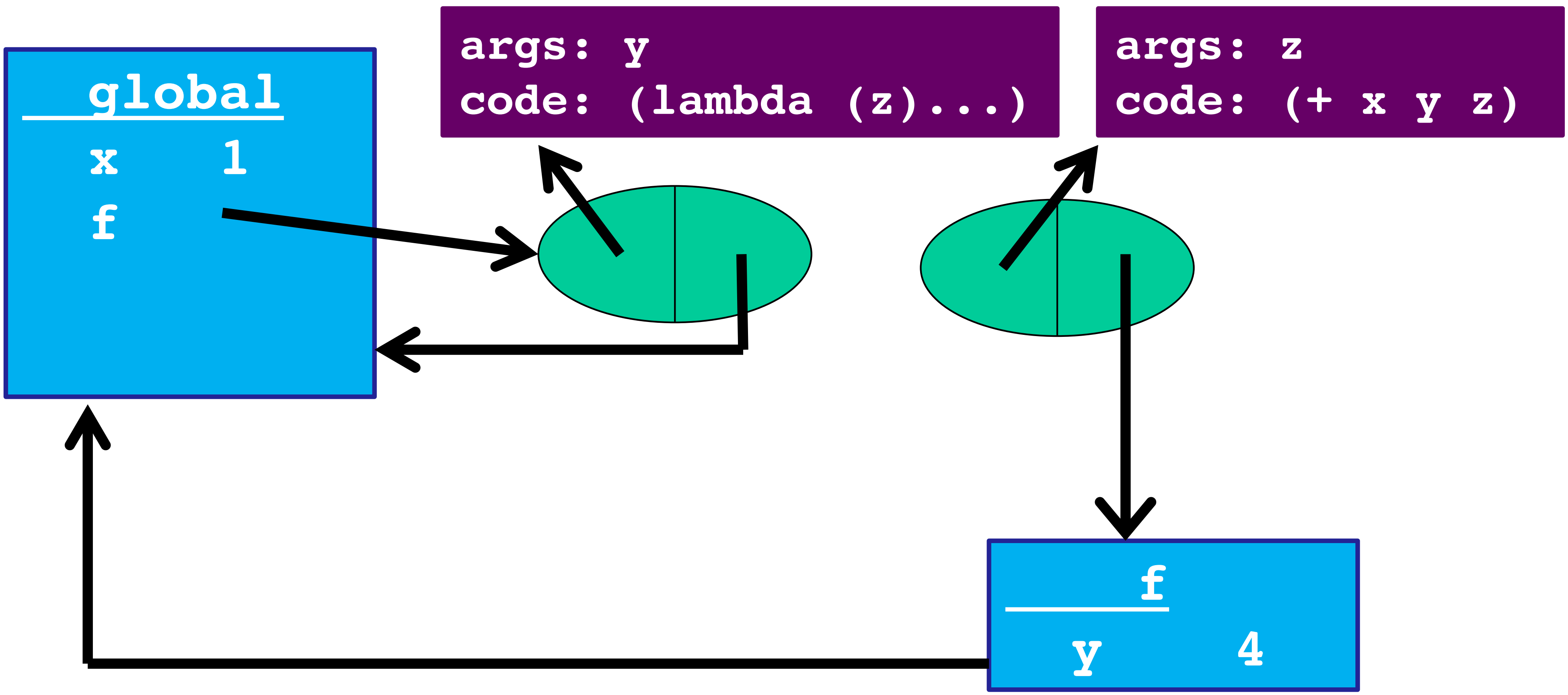
```
(define x 1)
(define (f y) (lambda (z) (+ x y z)))
(define g (f 4))
(define z (g 6))
```



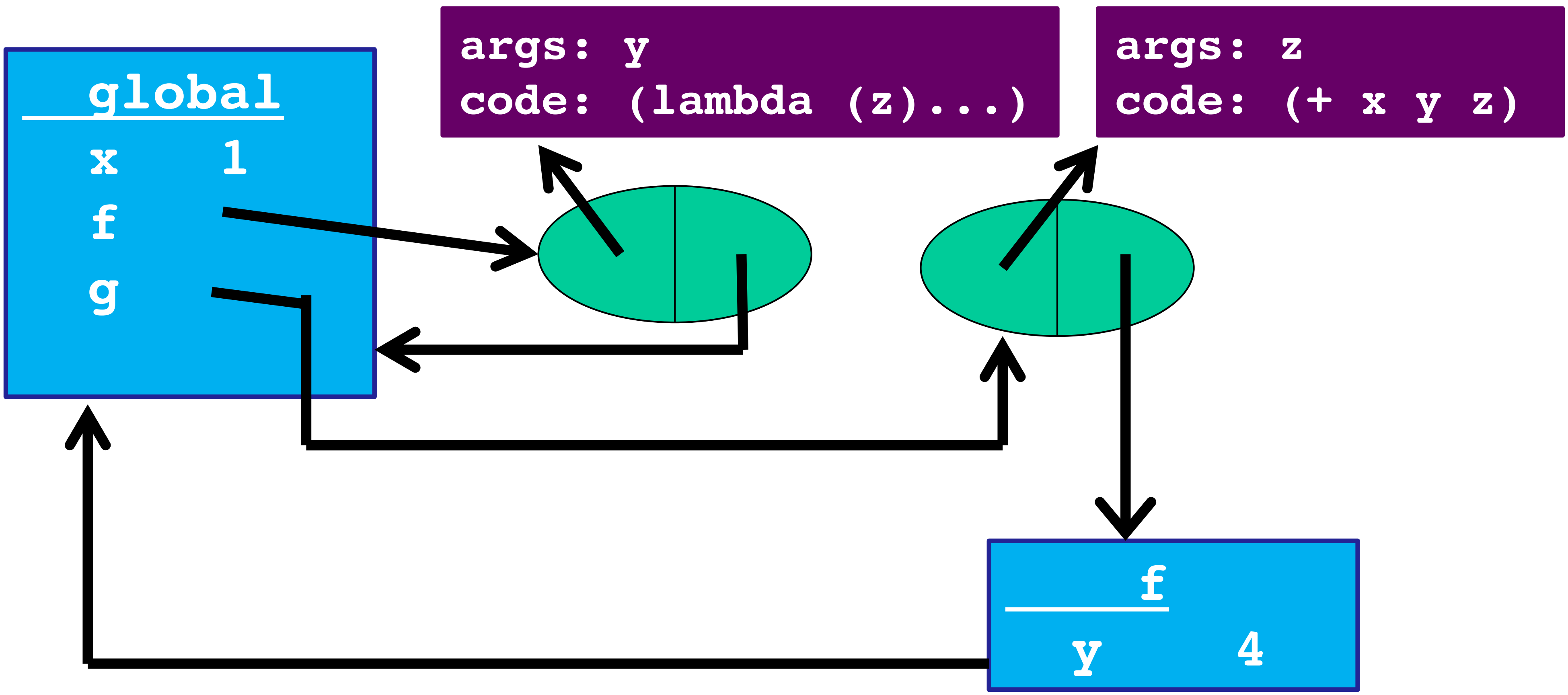
```
(define x 1)
(define (f y) (lambda (z) (+ x y z)))
(define g (f 4))
(define z (g 6))
```



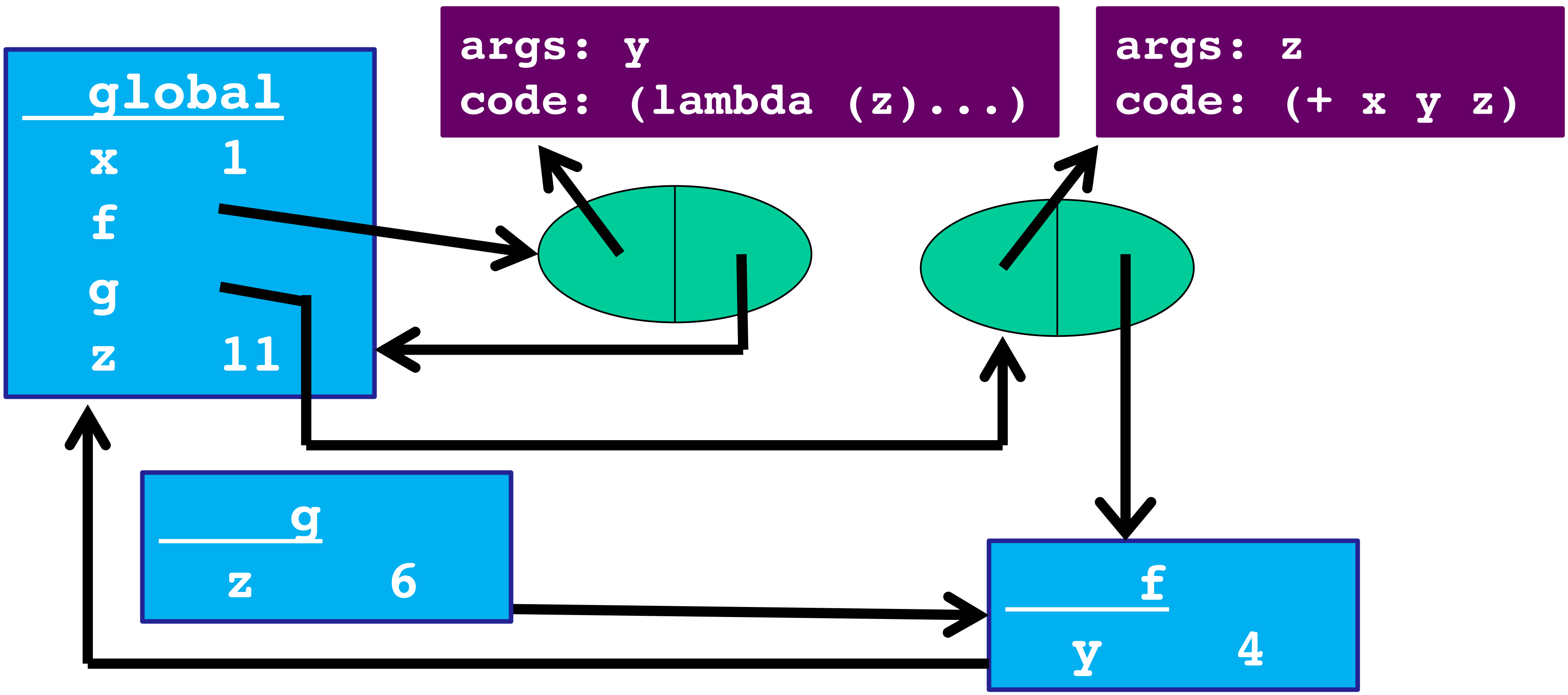
```
(define x 1)
(define (f y) (lambda (z) (+ x y z)))
(define g (f 4))
(define z (g 6))
```



```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))
```



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

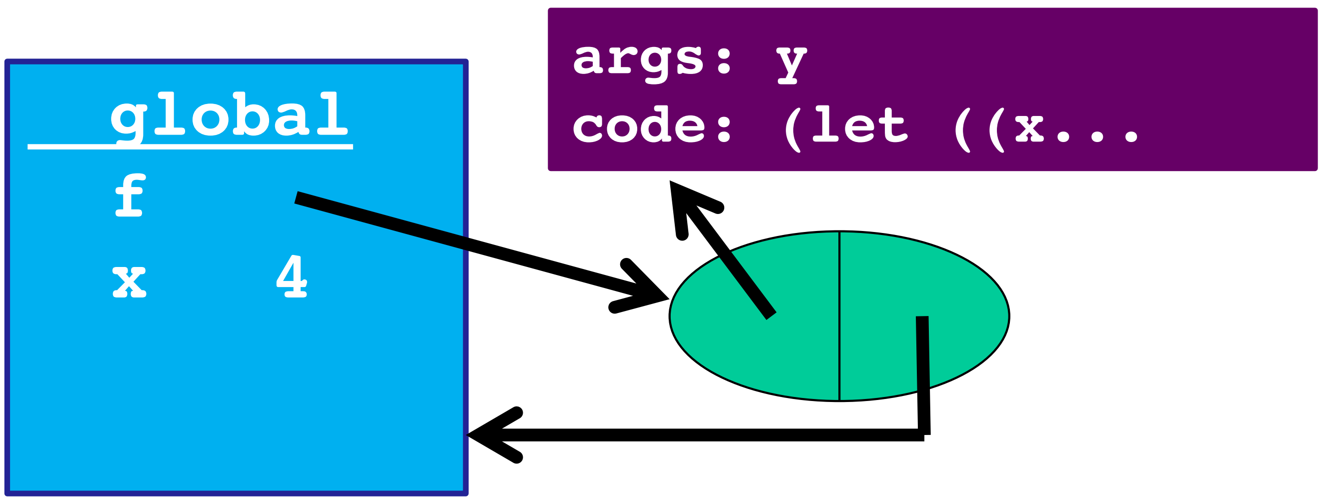
Example: Passing a function

- 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).

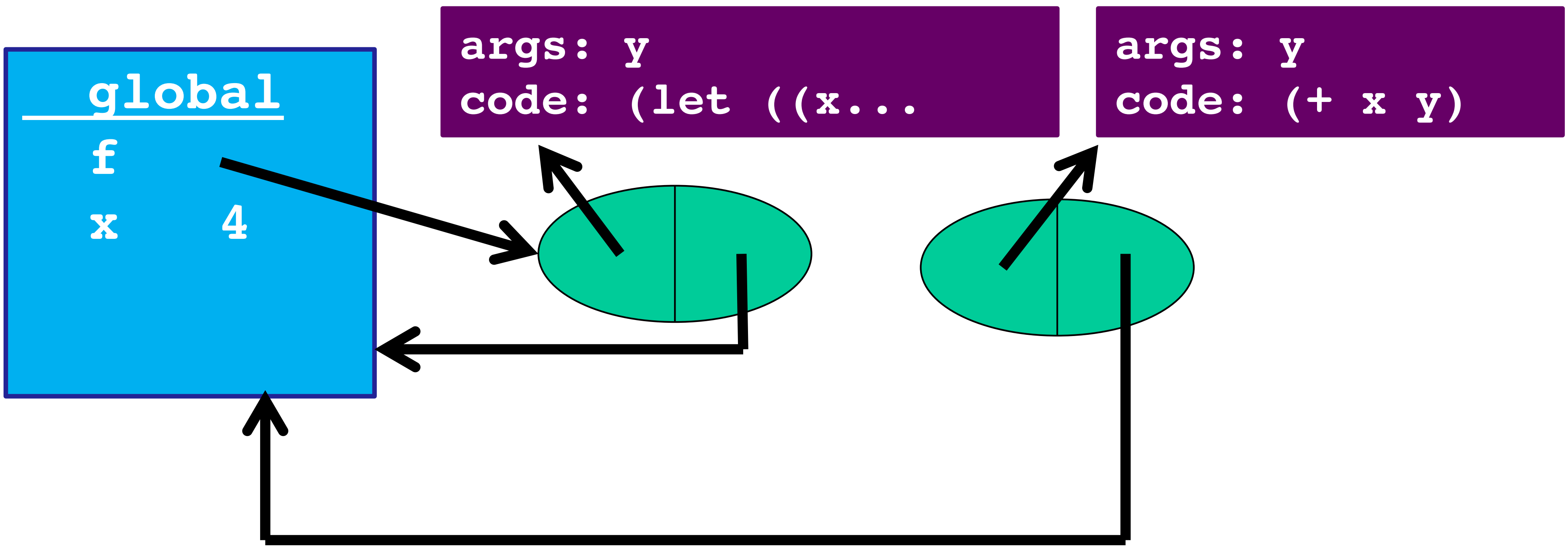
```
1  (define (f g) (let ((x 3)) (g 2)))
2  (define x 4)
3  (define (h y) (+ x y))
4  (define z (f h))
```

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

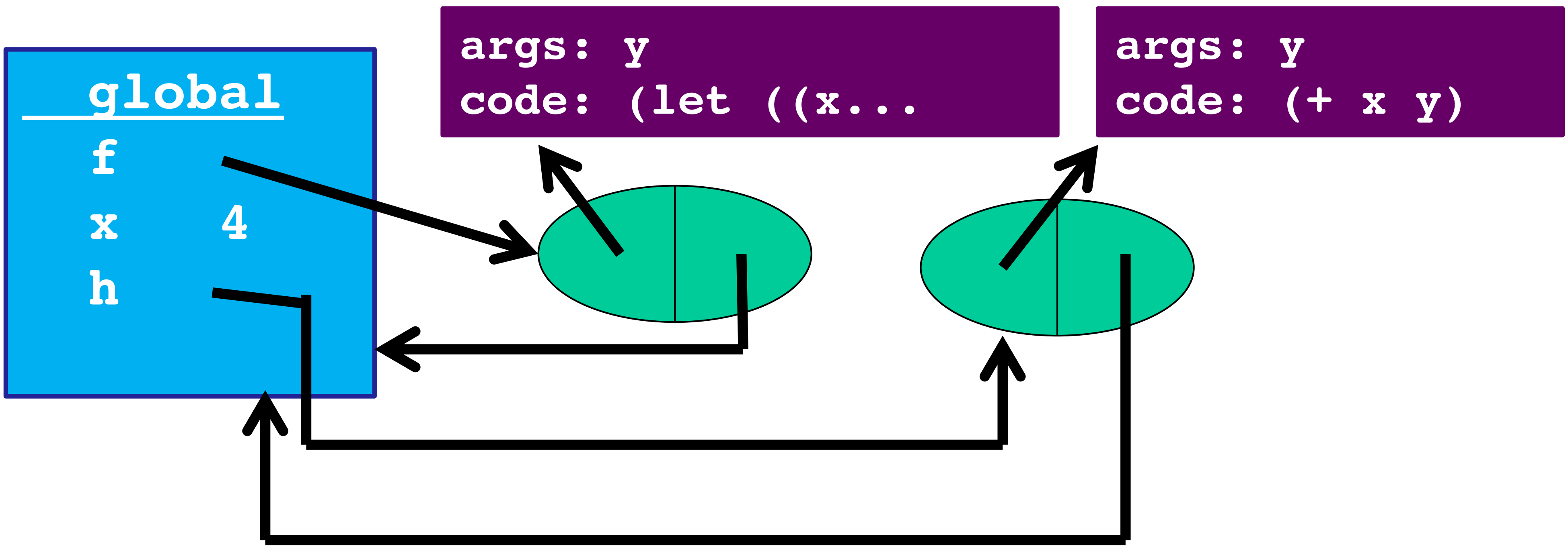
```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



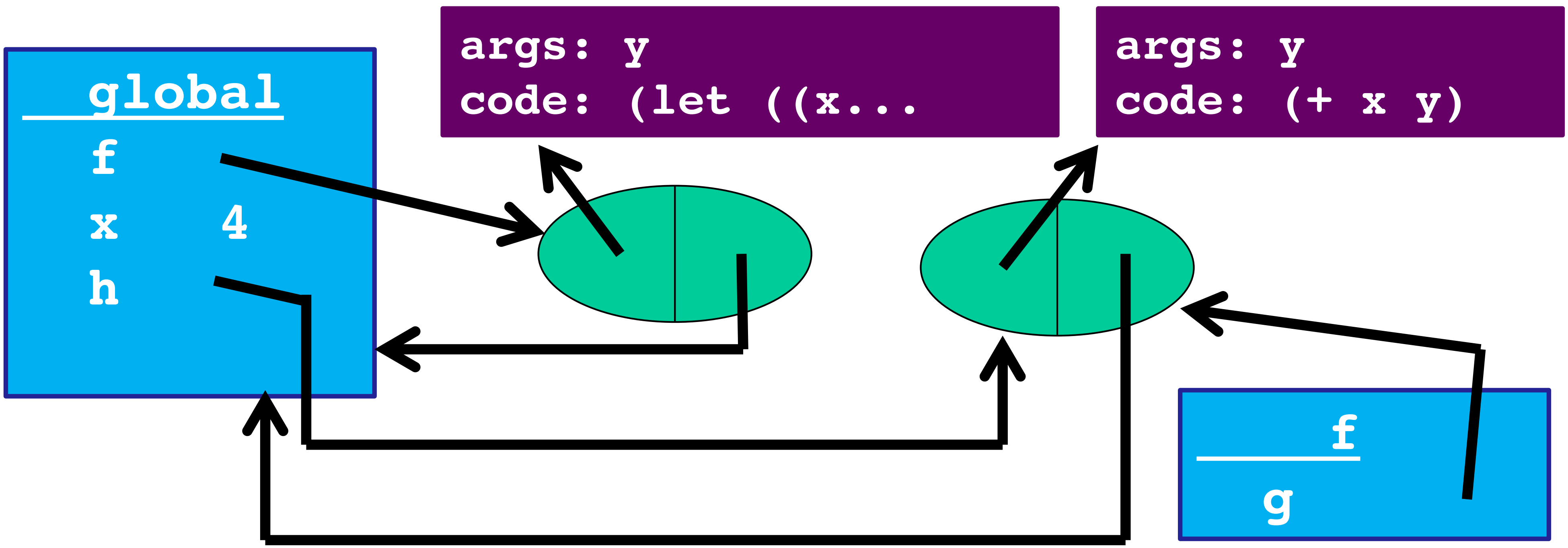
```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



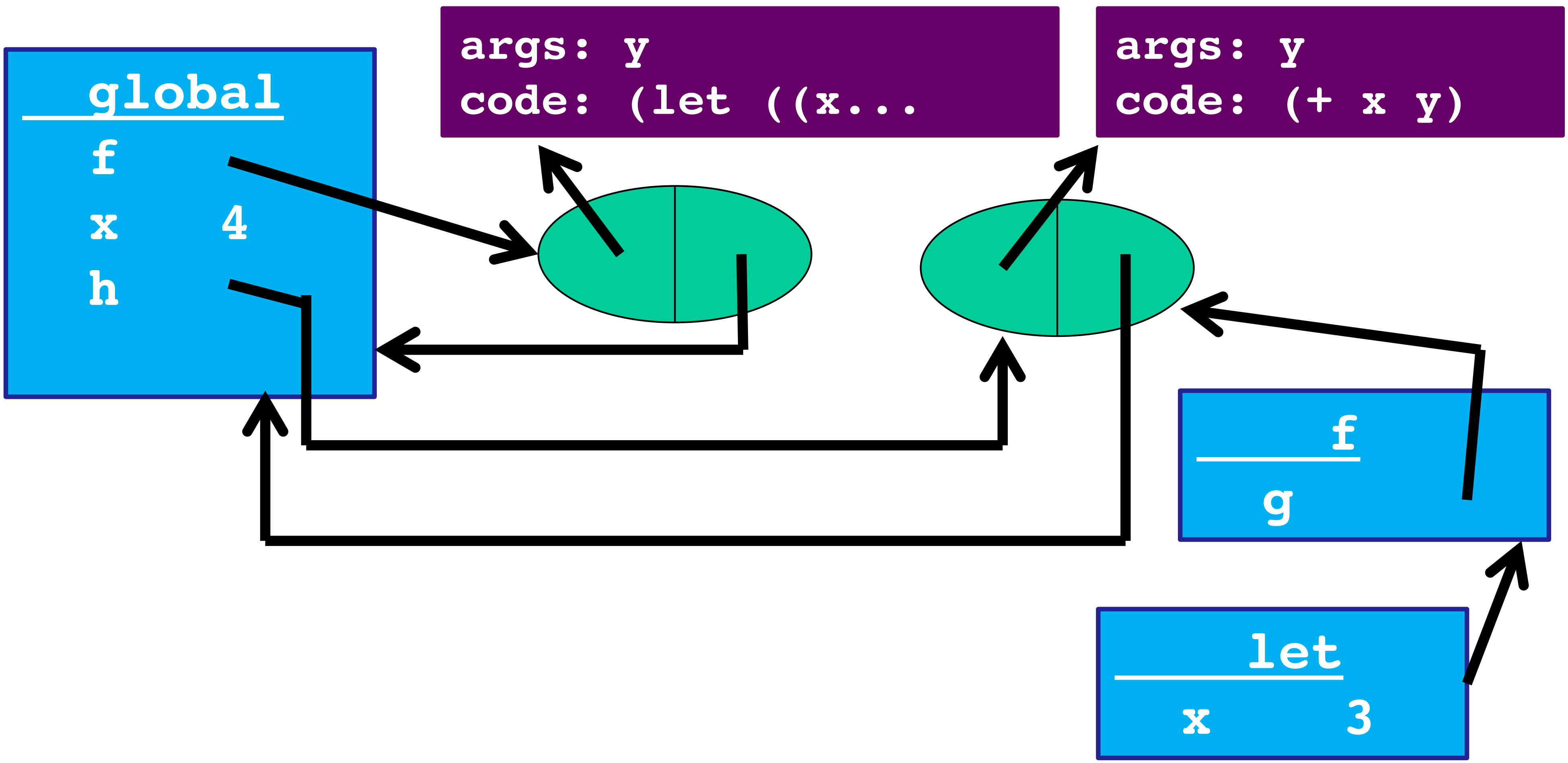
```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



```
(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))
```