Programming Languages

Course Motivation
(or, why we are spending so much time on a language that few people have heard of)

Course Motivation

(Did you think I forgot? ☺)

- Why learn languages that are quite different from Python or C++?
- Why learn the fundamental concepts that appear in all (most?) languages?
- Why focus on functional programming?

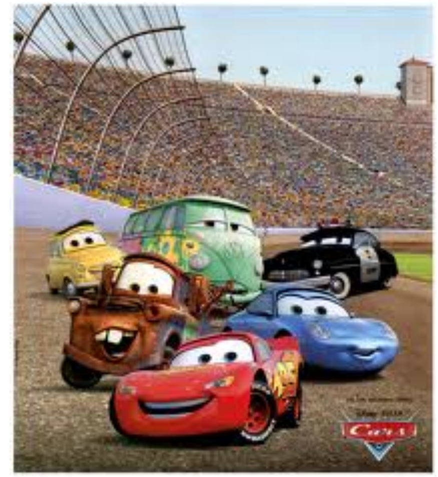
What is the best kind of car?

What is the best kind of shoes?

Cars / Shoes

Cars are used for rather different things:

- Winning the Indy 500
- Taking kids to soccer practice
- Off-roading
- Hauling a mattress
- Getting the wind in your hair
- Staying dry in the rain



Shoes:

- Playing basketball
- Going to a dance
- Going to the beach

More on cars

- A good mechanic might have a specialty, but also understands how "cars" (not 2014 Honda Civics) work
  - And that the syntax, I mean upholstery color, isn't essential
- A good mechanical engineer really knows how cars work, how to get the most out of them, and how to design better ones
- To learn how cars work, it may make sense to start with a classic design rather than the latest model
  - A popular car may not be a good car for learning how cars work

All cars are the same

- To make it easier to rent cars, it’s great that they all have steering wheels, brakes, windows, headlights, etc.
  - Yet it’s still uncomfortable to learn a new one

- And maybe programming languages are more like cars, trucks, boats, and bikes

- So are all programming languages really the same?

Are all languages the same?

Yes:

- Any input-output behavior implementable in language X is implementable in language Y [Church-Turing thesis]
- Python, C++, Racket, and a language with one loop and three infinitely-large integers are "the same"
- Beware "the Turing tarpit"

Yes:

- Same fundamentals reappear: variables, abstraction, recursive definitions, …

No:

- The primitive/default in one language is awkward in another

A note on reality

Reasonable questions when deciding to use/learn a language:

- What libraries are available for reuse?
- What can get me a summer internship?
- What does my boss tell me to do?
- What is the de facto industry standard?
- What do I already know?

CS 360 by design does not deal with these questions

- You have the rest of your life for that
- And the answers will change in 5, 10, 15, 20 years anyway

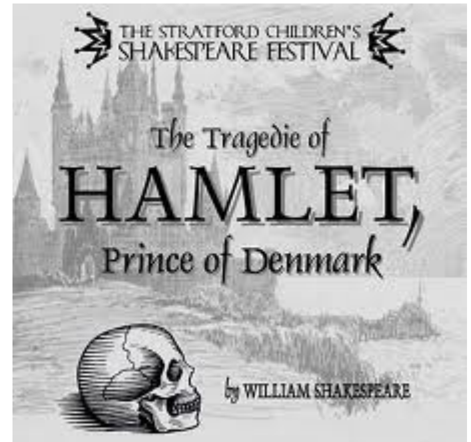
Why semantics and idioms

This course focuses as much as it can on semantics and idioms

- Correct reasoning about programs, interfaces, and interpreters or compilers *requires* a precise knowledge of semantics
  - Not “I feel that conditional expressions might work like this”
  - Not “I like curly braces more than parentheses”
  - Much of software development is designing precise interfaces; what a PL means is a *really* good example

- Idioms make you a better programmer
  - Best to see in multiple settings, including where they shine
  - Even if I never show you language X, when you see that idiom in the real world in language X, you'll understand it.

Hamlet



The play *Hamlet*:

- Is a beautiful work of art
- Teaches deep, eternal truths
- Is the source of some well-known sayings
- Makes you a better person

Continues to be studied (even in college) centuries later even though:

- The syntax is really annoying to many (yet rhythmic)
- There are more popular movies with some of the same lessons (just not done as well)
- Reading *Hamlet* will not get you a summer internship

Functional Programming

Okay, so why do we spend so much time with *functional languages*, i.e., languages where:

- Mutation is unavailable or discouraged
- Recursion expresses all forms of looping and iteration
- Higher-order functions are very convenient

Because:

1. These features are invaluable for correct, elegant, efficient software (great way to think about computation)
2. Functional languages have always been ahead of their time
3. Functional languages well-suited to where computing is going

Most of course is on (1), so a few minutes on (2) and (3) …

Ahead of their time

All of these were dismissed as "beautiful, worthless, slow things PL professors make you learn in school"

- Garbage collection (now used in Python, Java, …)
- Collections (i.e., lists) that can hold multiple data types at once (Python, Java through generics, C++ through templates)
- XML for universal data representation (like Racket/Scheme/LISP)
- Higher-order functions (Python, Ruby, JavaScript, more recent versions of C++, …)
- Recursion (a big fight in 1960 about this – I'm told ☺)

Somehow nobody notices the PL people were right all along.

Recent Surge

- Microsoft: F#, C# 3.0
- Scala (Twitter, LinkedIn, FourSquare)
- Java 8 (2014), C++ (2014)
- MapReduce / Hadoop (everybody)
  - Avoiding side-effects essential for fault-tolerance here
- Haskell (dozens of small companies/teams)
- Erlang (distributed systems, Facebook chat)

Why a surge?

My best guesses:

- Concise, elegant, productive programming
- JavaScript, Python, Ruby helped break the Java/C/C++ hegemony
  - And these functional languages do some things better
- Avoiding mutation is *the* easiest way to make concurrent and parallel programming easier
- Sure, functional programming is still a small niche, but there is so much software in the world today even niches have room

Is this real programming?

- The way we're using Racket in this class can make the language seem almost "silly" precisely because lecture and homework focus on interesting language constructs
- "Real" programming needs file I/O, string operations, graphics, project managers, testing frameworks, threads, build systems, …
  - Functional languages have all that and more
  - If we used C++ or Python the same way, those languages would seem "silly" too

Summary

- No such thing as a “best” PL
- There are good general design principles for PLs
- A good language is a relevant, crisp interface for writing software
- Software leaders should know PL semantics and idioms
- Learning PLs is not about syntactic tricks for small programs
- Functional languages have been on the leading edge for decades
  - Ideas get absorbed by the mainstream, but very slowly
  - Meanwhile, use the ideas to be a better programmer in C++ and Python.

Programming Languages

Lexical Scope and Closures

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))
```

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)

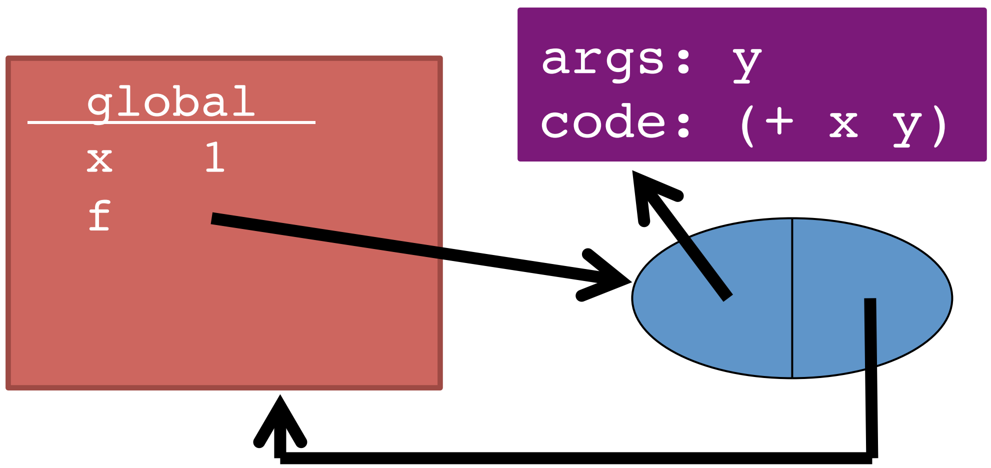
What's happening behind the scenes

- 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))))
```

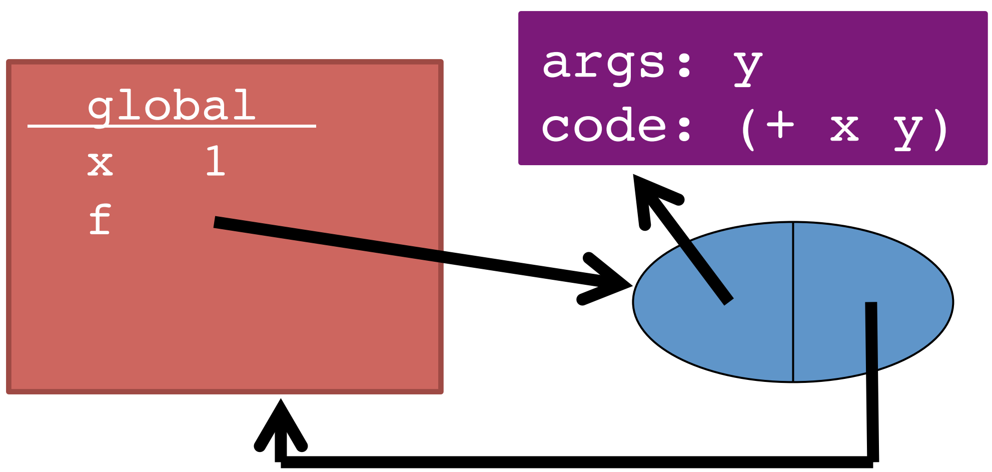
```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define y 4)
-4- (define z (let ((x 2)) (f (+ x y))))
```

```
-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

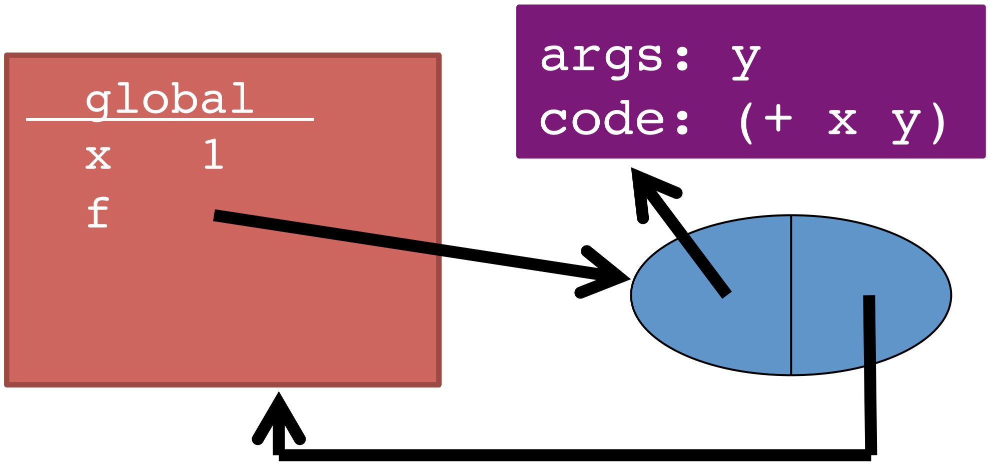
- Rule 1:
  - Every function **definition** (including anonymous function definitions) creates a closure where
    - the code part of the closure points to the function's code
    - the environment part of the closure points to the frame that was current when the function was defined (the frame we are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



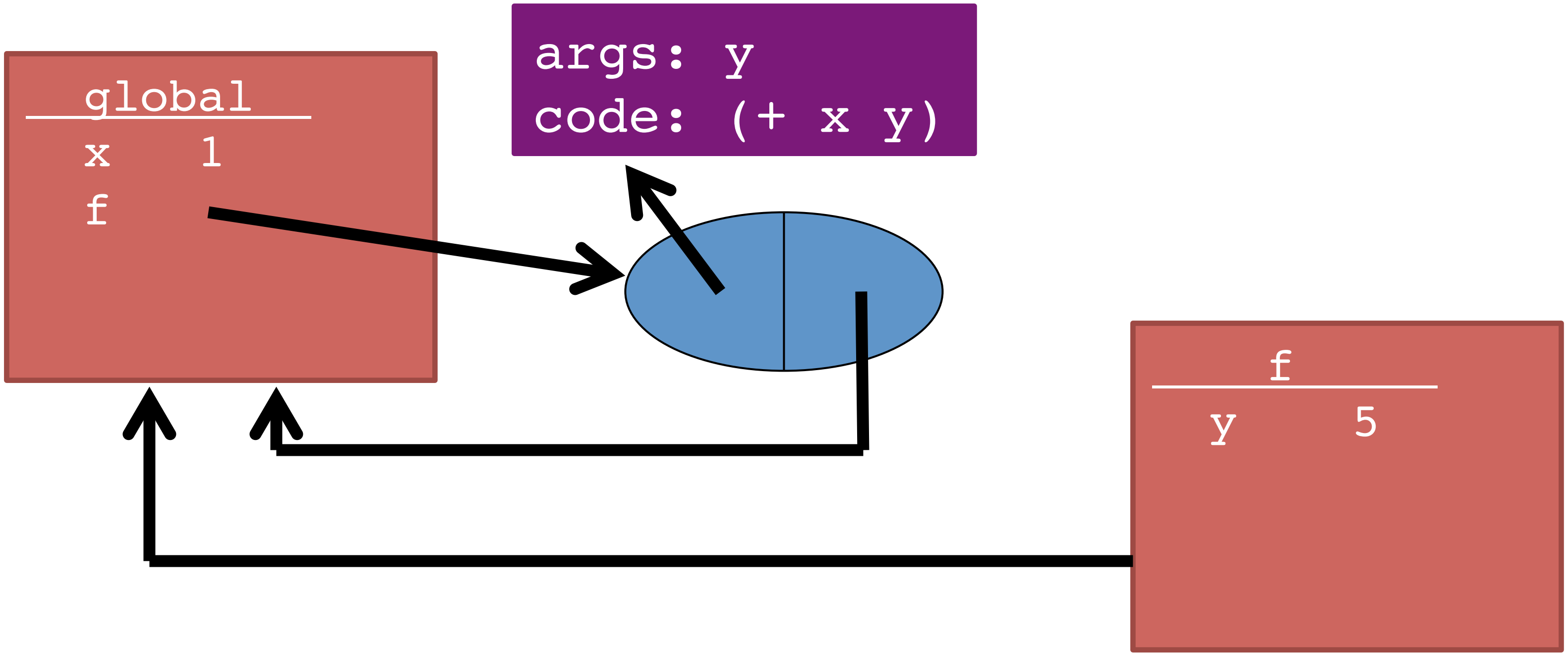
Rules for frames and environments

- 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.

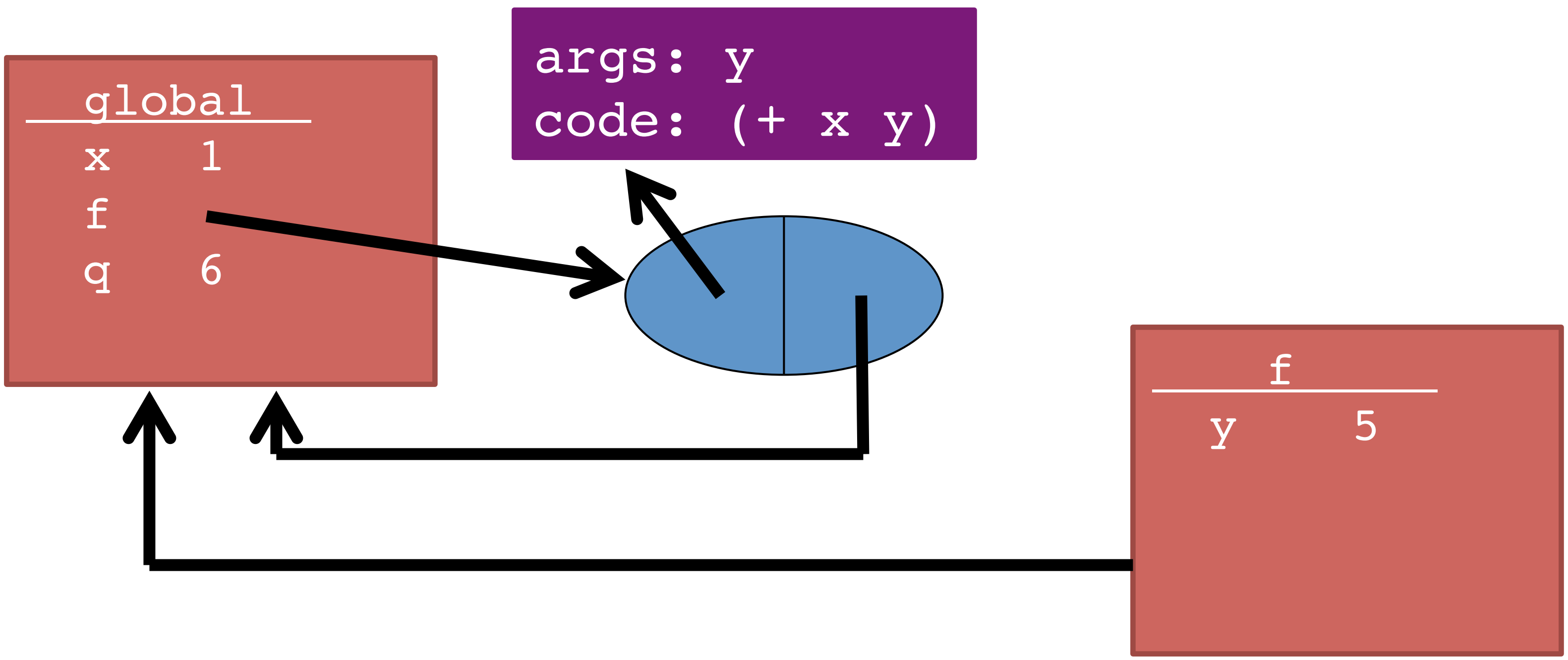
```
-1- (define x 1)
-2- (define (f y) (+ x y))
-3- (define q (f 5))   ; changed this line
```



```
-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

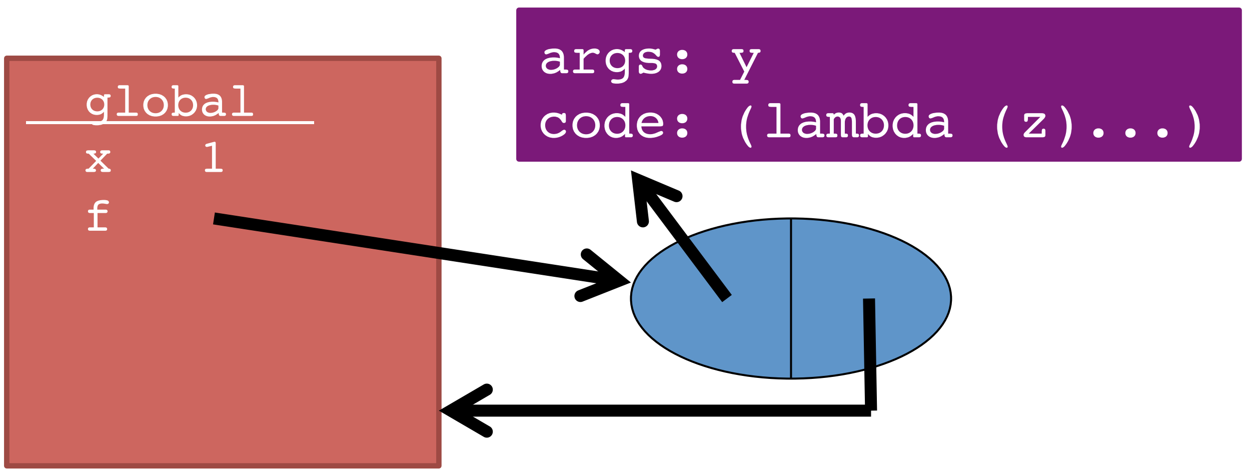
Example: Returning a function

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

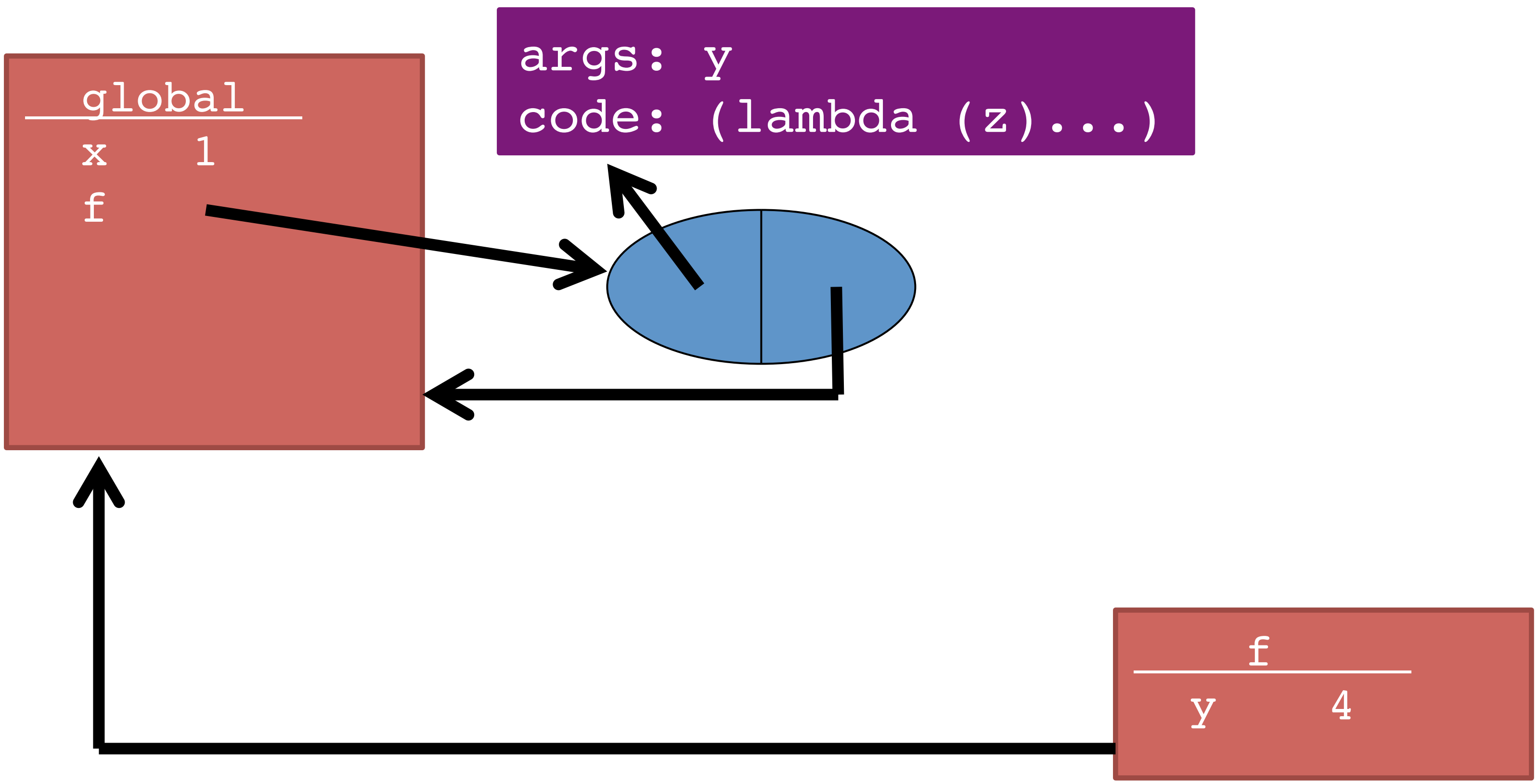
- Trust the rules:
  - Evaluating line 2 binds f to a closure.
  - Evaluating line 3 binds g to a closure as well.
    - New frame is created for the call to f.
  - Evaluating line 4 binds z to a number.
    - New frame is created for the call to g.

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

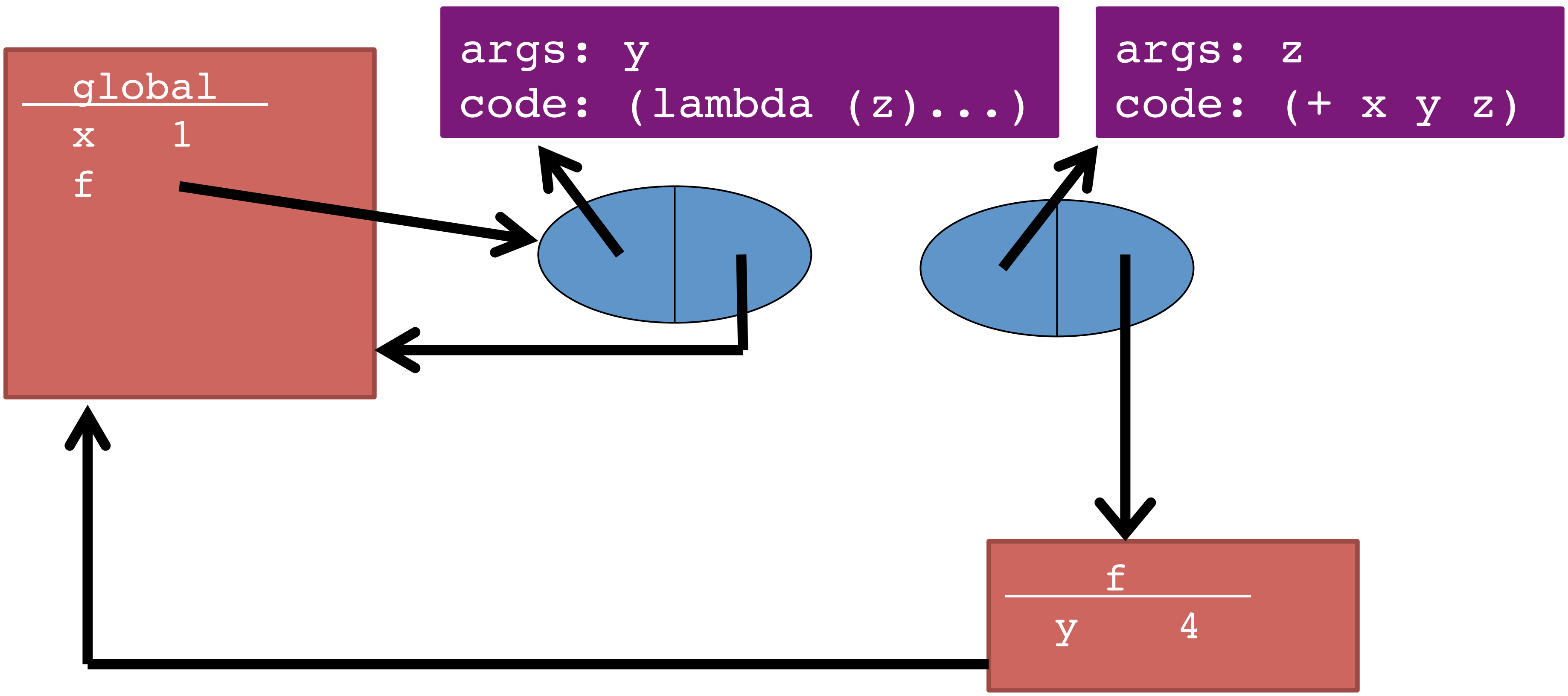
```
1   (define x 1)
2   (define (f y) (lambda (z) (+ x y z)))
3   (define g (f 4))
4   (define z (g 6))
```



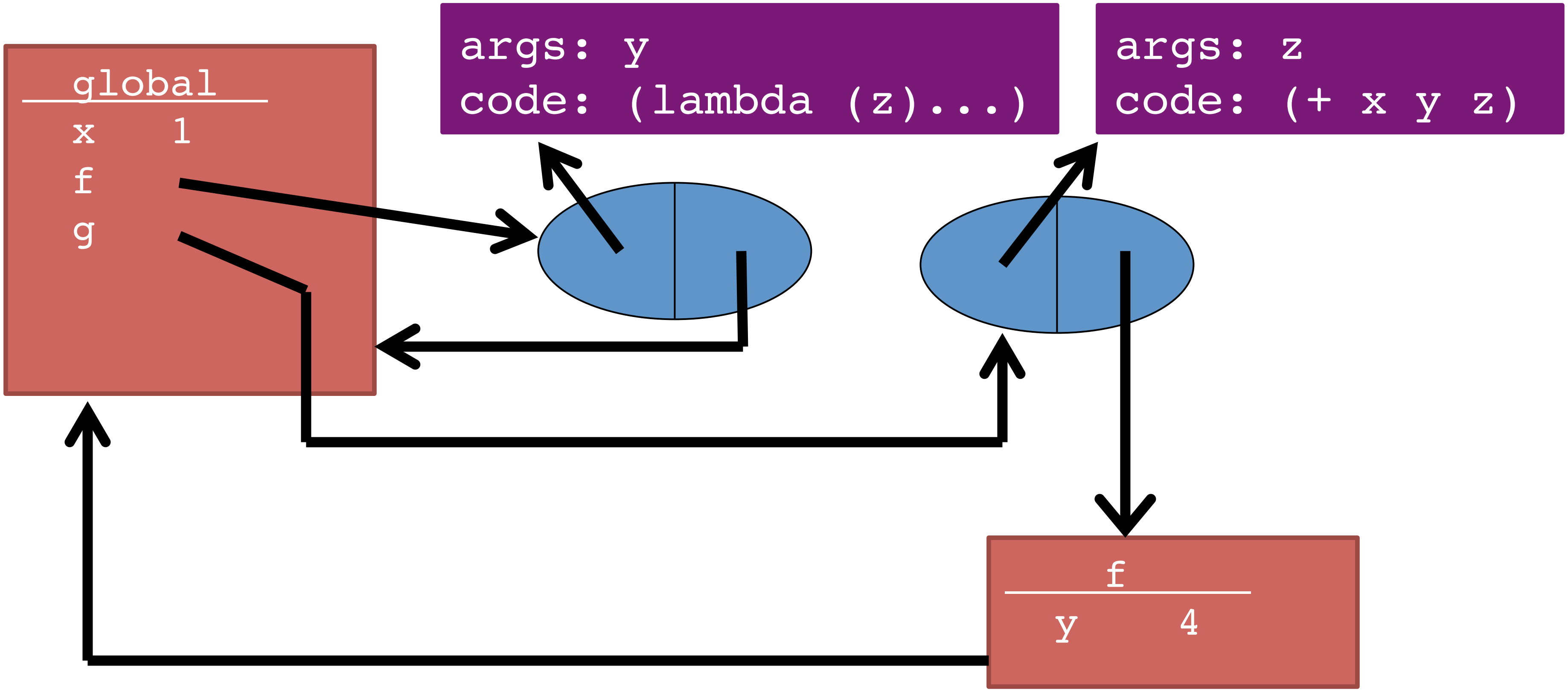
```
1   (define x 1)
2   (define (f y) (lambda (z) (+ x y z)))
3   (define g (f 4))
4   (define z (g 6))
```



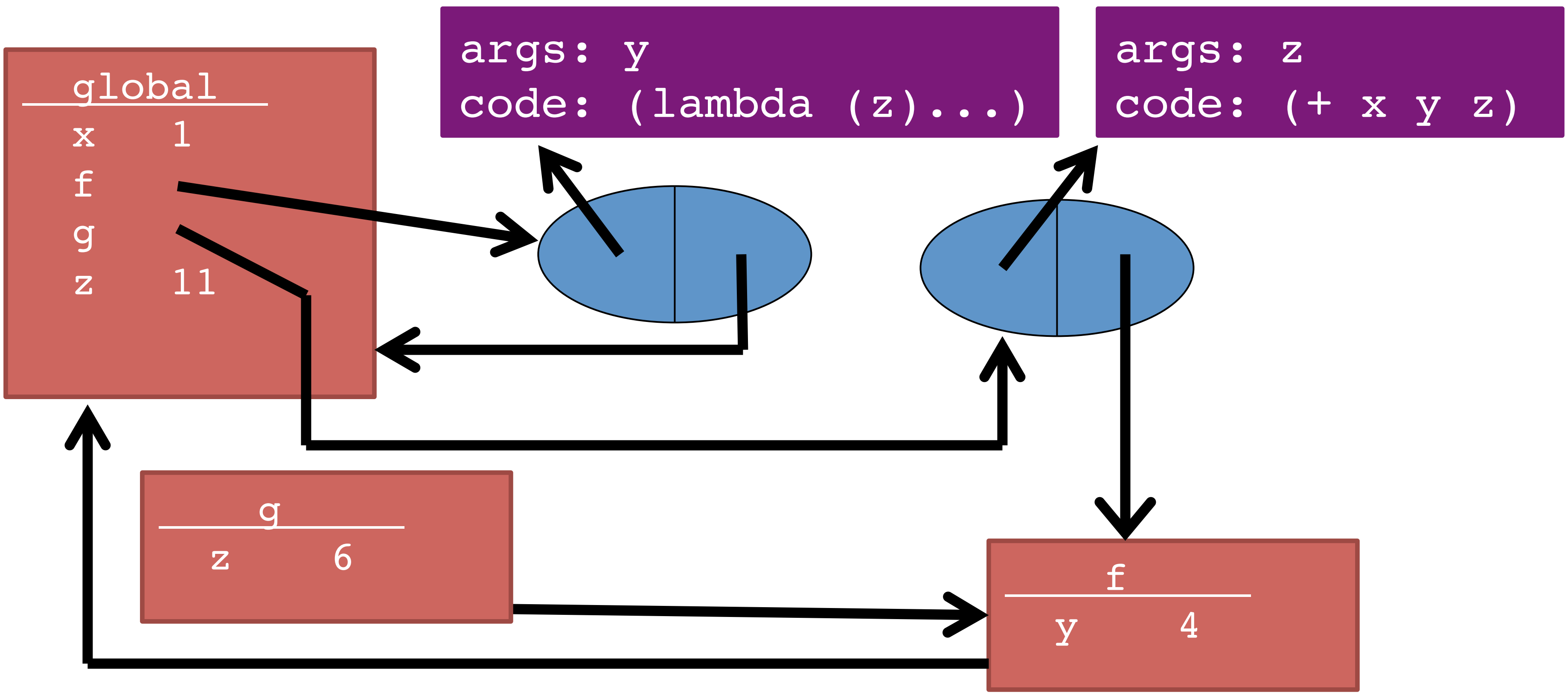
```
1   (define x 1)
2   (define (f y) (lambda (z) (+ x y z)))
3   (define g (f 4))
4   (define z (g 6))
```



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



```
(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

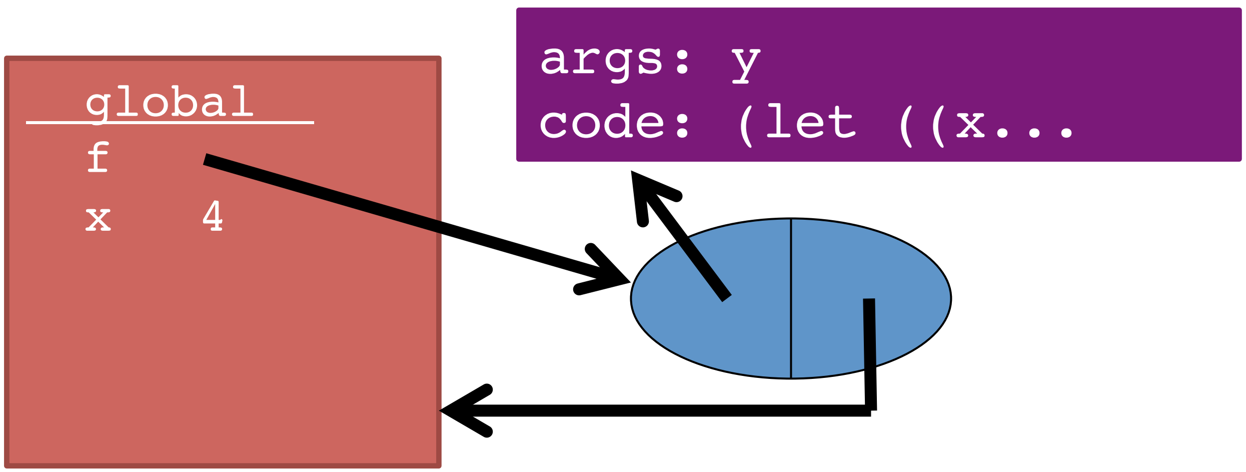
Example: Passing a function

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

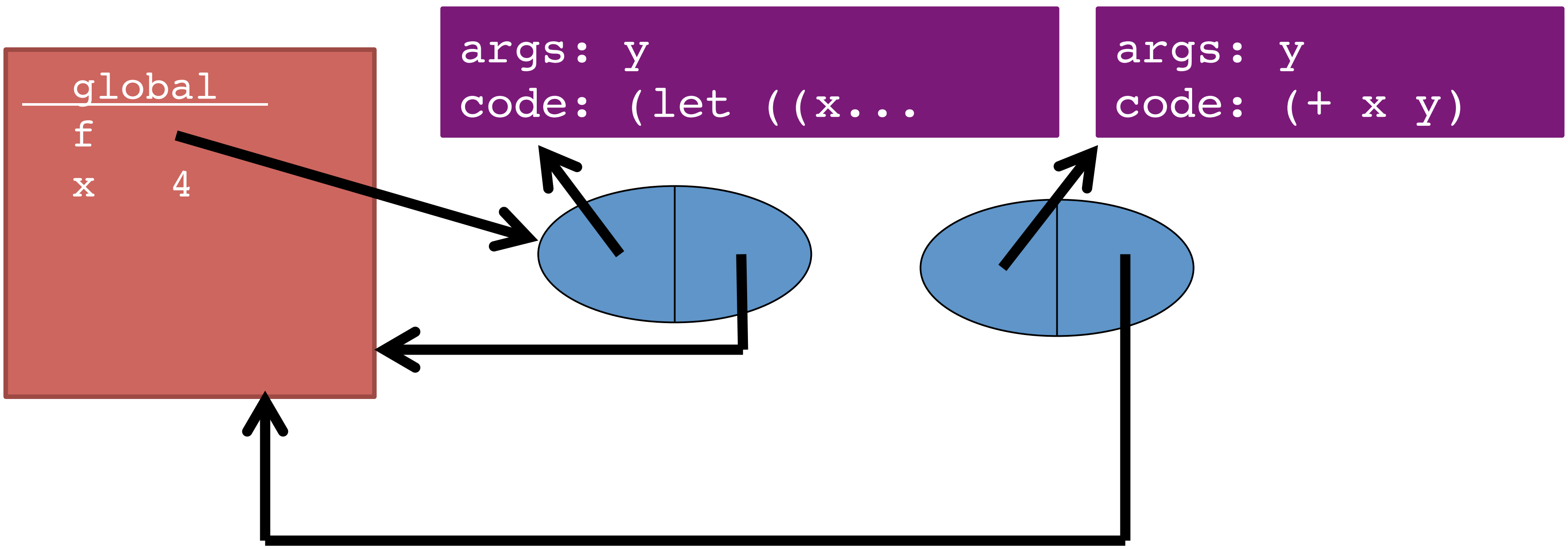
- Trust the rules:
  - Evaluating line 1 binds f to a closure.
  - Evaluating line 2 binds x to 4.
  - Evaluating line 3 binds h to a closure.
  - Evaluating line 4 binds z to a number.
    - First, calls f (creates new frame), then evaluates "let" (creates a new frame), then calls g (creates a new frame).

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

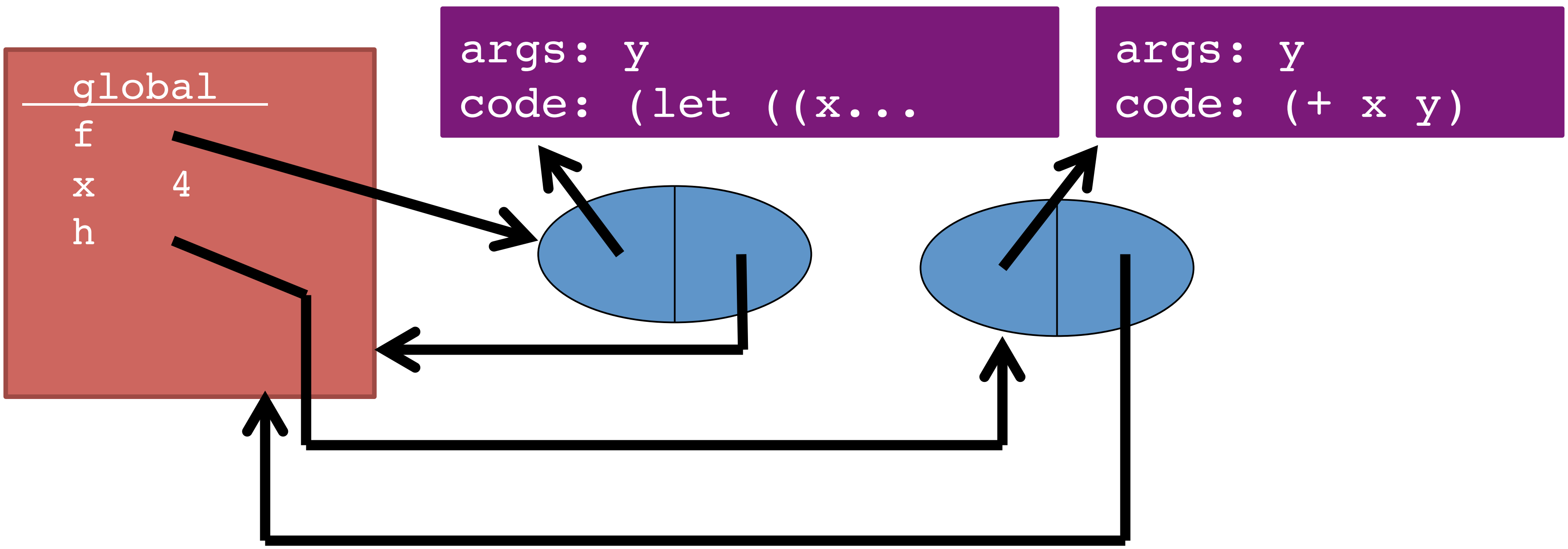
```
1   (define (f g) (let ((x 3)) (g 2)))
2   (define x 4)
3   (define (h y) (+ x y))
4   (define z (f h))
```



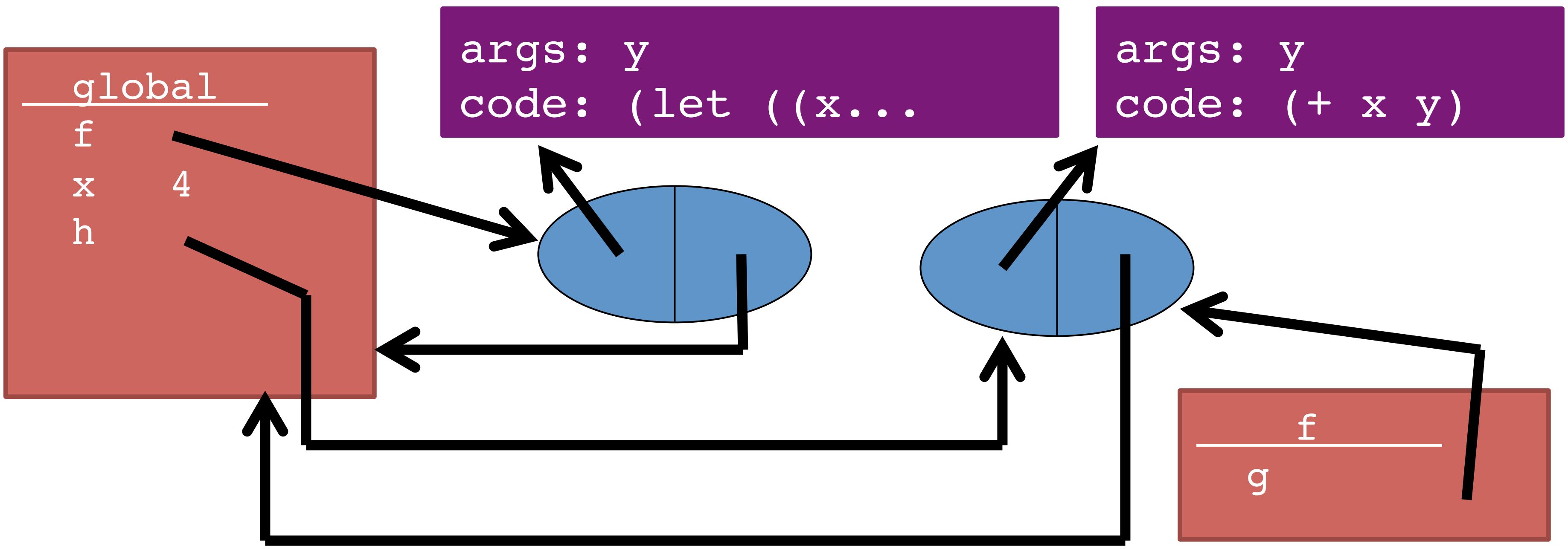
```
(define (f g) (let ((x 3)) (g 2)))
(define x 4)
(define (h y) (+ x y))
(define z (f h))
```



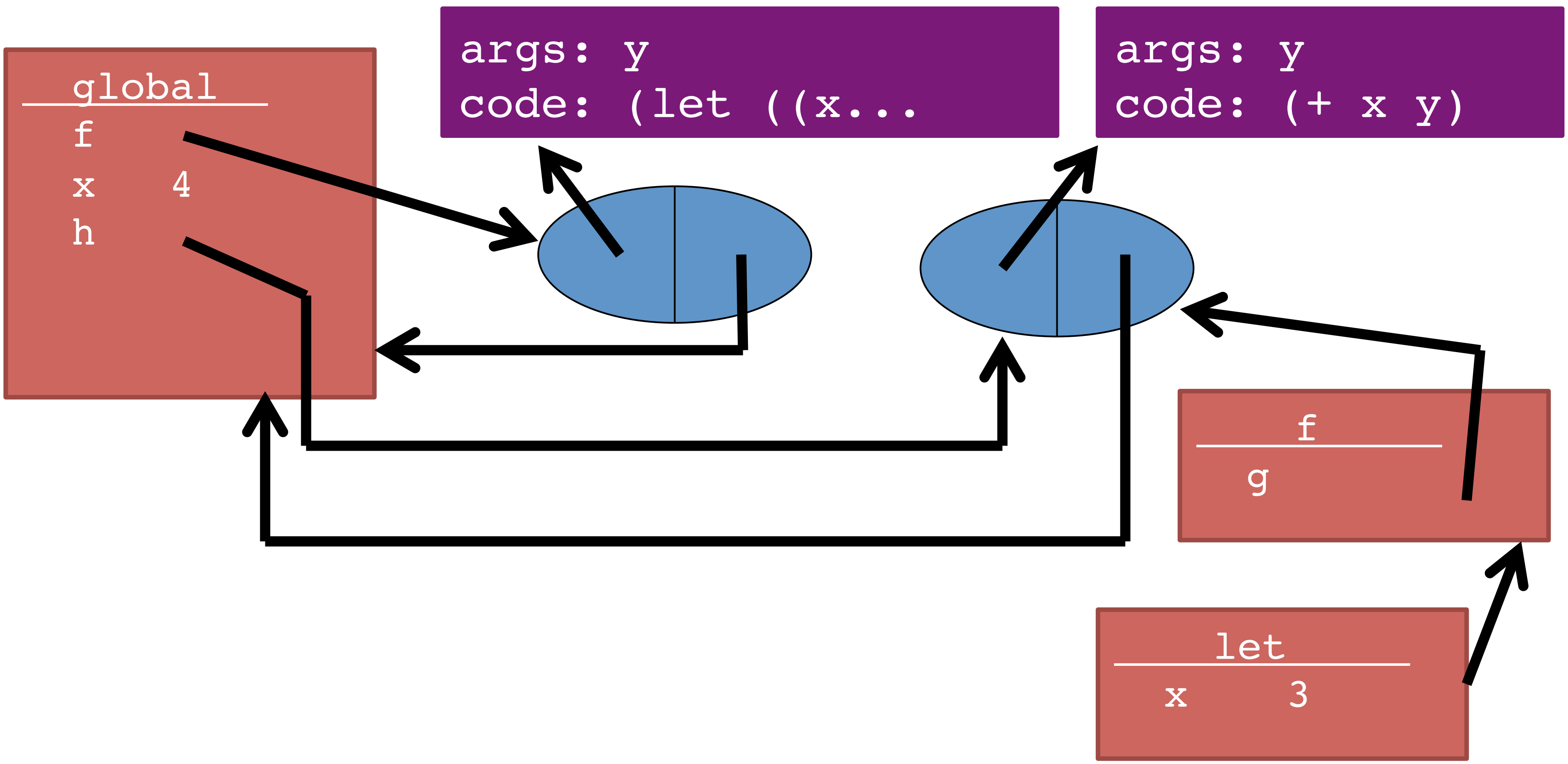
```
1   (define (f g) (let ((x 3)) (g 2)))
2   (define x 4)
3   (define (h y) (+ x y))
4   (define z (f h))
```



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



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



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