

# Discussion 02

HOF, Environment Diagrams (for days)

6/28



# 1.2 (last part)

*Walking through how to execute function calls*

What will the following output?

```
def negate(f, x):  
    return -f(x)  
def square(n):  
    return n * n  
def double(n):  
    return 2 * n  
>>> negate(double, negate(square, -4))
```



# Evaluate Operators and Operands

```
>>> negate(double, negate(square, -4))
```



# Evaluate Operators and Operands

Evaluate the  
operator


```
>>> negate(double, negate(square, -4))
```



# Evaluate Operators and Operands

We have  
defined the  
function  
with the  
name  
**negate** in  
global

Evaluate the  
operator

>>>  **negate**(double, negate(square, -4))



# Evaluate Operators and Operands

We have  
defined the  
function  
with the  
name  
`negate` in  
global

Evaluate the  
operator



Evaluate the  
operands

```
>>> negate(double, negate(square, -4))
```



# Evaluate Operators and Operands

We have defined the function with the name **negate** in global

Evaluate the operator



We have defined the function with the name **double** in global



Evaluate the operands

```
>>> negate(double, negate(square, -4))
```



# Evaluate Operators and Operands

We have defined the function with the name **negate** in global

Evaluate the operator



We have defined the function with the name **double** in global



Evaluate the operands

```
>>> negate(double, negate(square, -4))
```

Evaluate the operator



# Evaluate Operators and Operands

We have defined the function with the name **negate** in global

Evaluate the operator

We have defined the function with the name **double** in global

Evaluate the operands

>>> **negate**(**double**, **negate**(square, -4))

Evaluate the operator



# Evaluate Operators and Operands

We have defined the function with the name **negate** in global

Evaluate the operator

We have defined the function with the name **double** in global

Evaluate the operands

>>>

negate(double,

negate(square, -4))

Evaluate the operator

Evaluate the operands



# Evaluate Operators and Operands

We have defined the function with the name **negate** in global

Evaluate the operator

We have defined the function with the name **double** in global

>>>  negate(double, negate(square, -4))

Evaluate the operator

Evaluate the operands

square and -4 are both primitive



# Evaluate Operators and Operands

We have defined the function with the name **negate** in global

Evaluate the operator

We have defined the function with the name **double** in global

Evaluate the operands

square and -4 are both primitive

>>>  negate(double, square, -4)

We evaluated the operator, evaluated the operands. We are now ready for our first function call to **negate**. Note that this function call comes before the call to the **negate** on the outside of all the parenthesis (gray)

Evaluate the operator

Evaluate the operands





# Evaluate Operators and Operands

We have defined the function with the name **negate** in global

Evaluate the operator

We have defined the function with the name **double** in global

Evaluate the operands

square and -4 are both primitive

>>> **negate**(**double**, **negate**(**square**, **-4**))

We evaluated the operator, evaluated the operands. We are now ready for our first function call to **negate**. Note that this function call comes before the call to the **negate** on the outside of all the parenthesis (gray)

Evaluate the operator

Evaluate the operands



Execute the function call:

f1: **negate** [P = G]  
f: **square**  
x: -4  
RV: -16

f2: **square** [P = G]  
x: -4  
RV: 16



# Evaluate Operators and Operands

We have defined the function with the name **negate** in global

Evaluate the operator

We have defined the function with the name **double** in global

Evaluate the operands

square and -4 are both primitive

>>>

negate(double,

-16

The function call returned 16, so we can replace the complicated looking operand with the value 16

We evaluated the operator, evaluated the operands. We are now ready for our first function call to negate. Note that this function call comes before the call to the negate on the outside of all the parenthesis (gray)

Evaluate the operator

Evaluate the operands

Execute the function call:

```
f1: negate [P = G]
  f: square
  x: -4
  RV: -16
```

```
f2: square [P = G]
  x: -4
  RV: 16
```



# Evaluate Operators and Operands

We have defined the function with the name **negate** in global

Evaluate the operator

We have defined the function with the name **double** in global

Evaluate the operands

square and -4 are both primitive

>>>

negate(double,

-16

)

The function call returned 16, so we can replace the complicated looking operand with the value 16

We evaluated the operator, evaluated the operands. We are now ready for our first function call to negate. Note that this function call comes before the call to the negate on the outside of all the parenthesis (gray)

Evaluate the operator

Evaluate the operands



Execute the function call:

f1: negate [P = G]  
f: square  
x: -4  
RV: -16

f2: square [P = G]  
x: -4  
RV: 16

Note: this is not a fully complete environment diagram (there are missing components)

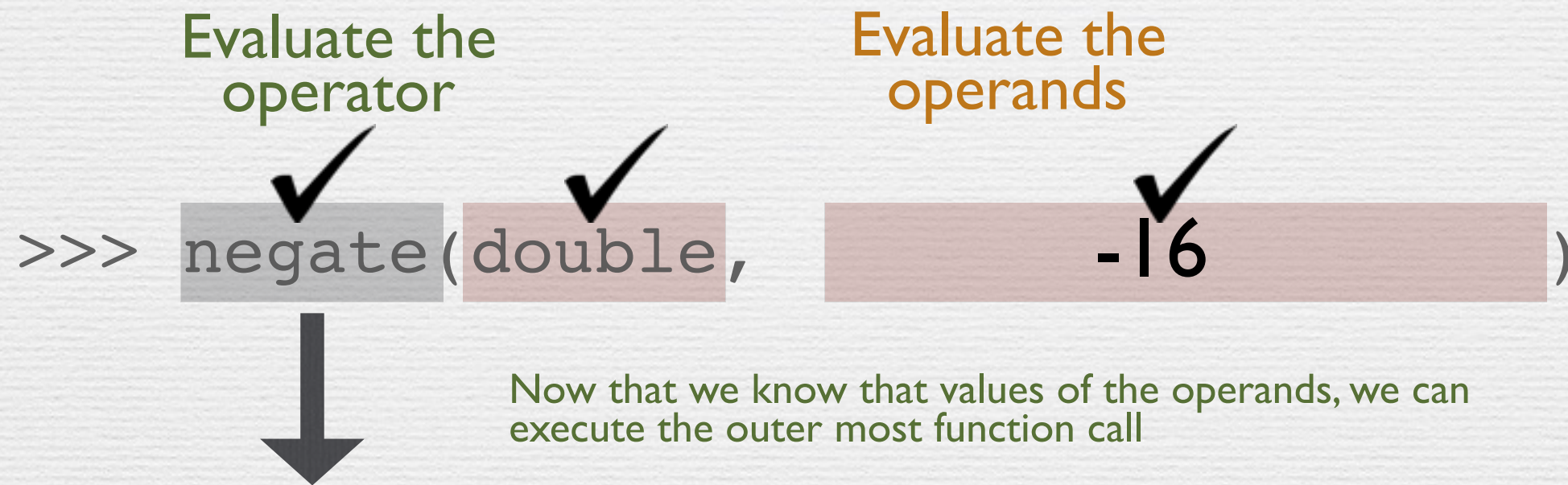
The rules for executing a function call are:

1. Evaluate the operator
2. Evaluate the operands
3. Execute the body of the function

Note that these rules can be interrupted. In this example we were preparing to execute the first negate, but were interrupted in the process of evaluating it's operands. Sometimes it is necessary to complete another function call before completing the one we initially started

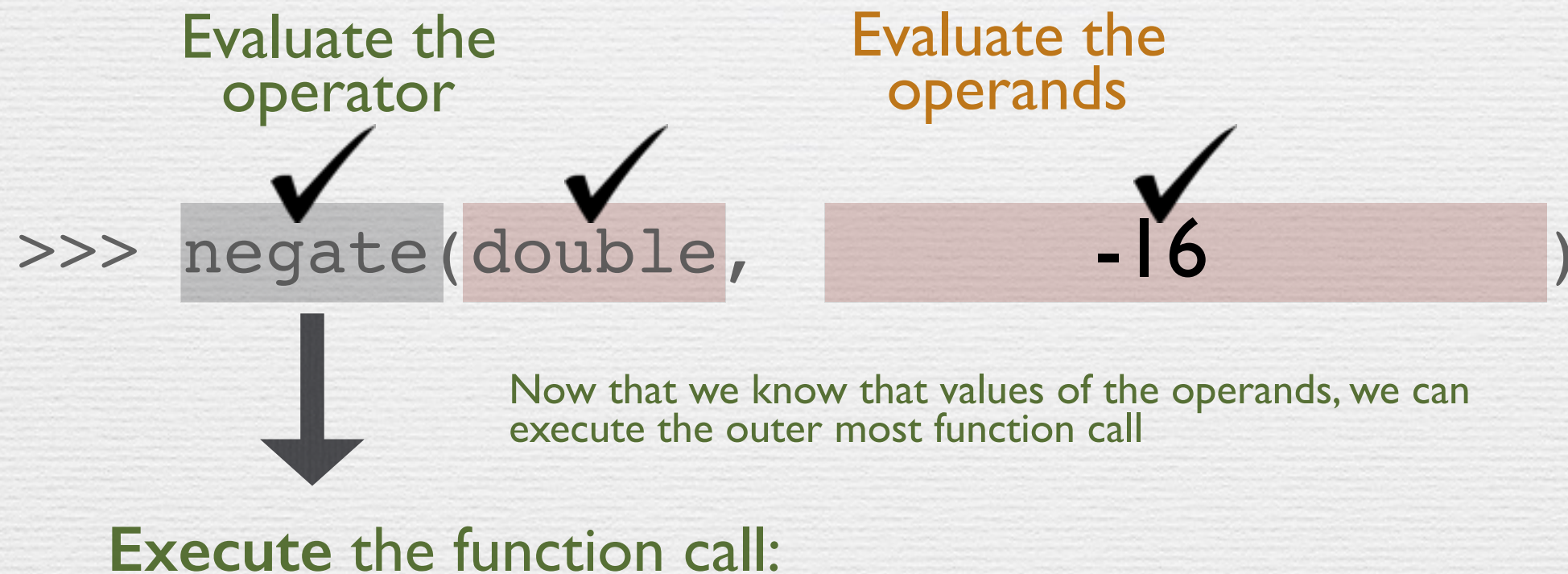


# Execute the Function Call



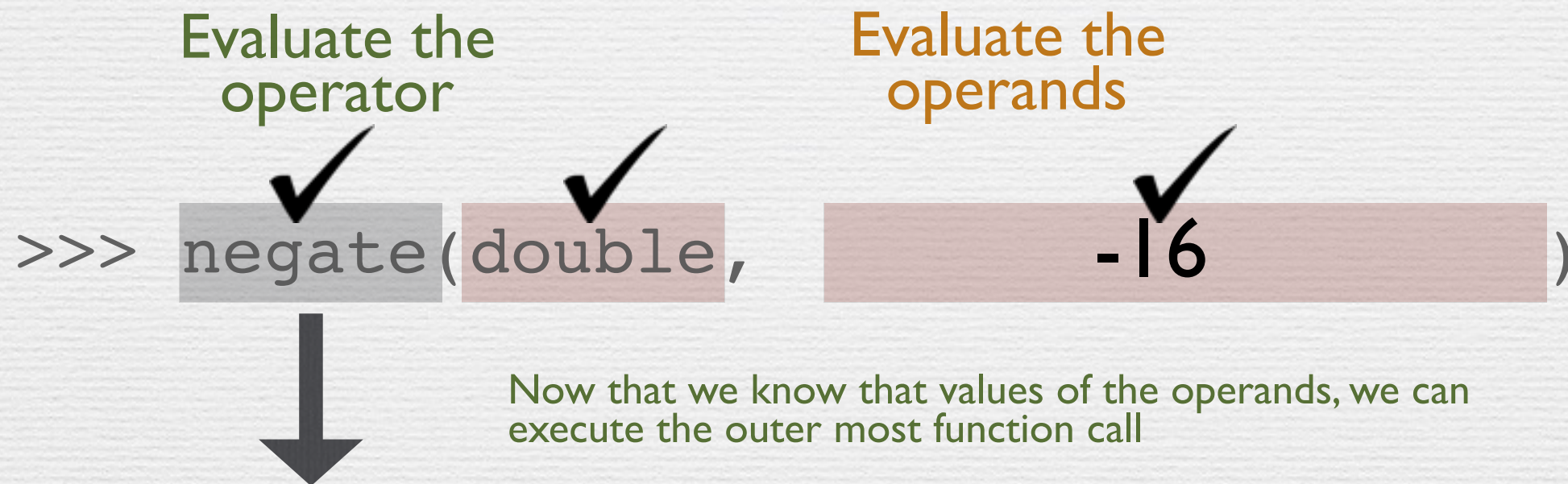


# Execute the Function Call





# Execute the Function Call



Execute the function call:

f1: negate [P = G]  
f: double  
x: -16  
RV: 32

f2: double [P = G]  
x: -16  
RV: -32



# Execute the Function Call



Now that we know the values of the operands, we can execute the outer most function call

Execute the function call:

f1: negate [P = G]  
f: double  
x: -16  
RV: 32

f2: double [P = G]  
x: -16  
RV: -32

**Solution: 32**



# What's different with HOF?

What's different between the code on the left and the code on the right? What will be printed when the code on the left is executed? What about the code on the right?

```
t = "surprise!"
def outer(t):
    def inner():
        print(t)
    return inner
outer("boo!")()
```

```
t = "surprise!"
def inner():
    print(t)
def outer(t):
    return inner
outer("boo!")()
```



```
t = "surprise!"  
def outer(t):  
    def inner():  
        print(t)  
    return inner  
outer("boo!")()
```

```
t = "surprise!"  
def inner():  
    print(t)  
def outer(t):  
    return inner  
outer("boo!")()
```

Draw environment diagrams to see what's different



```
t = "surprise!"  
def outer(t):  
    def inner():  
        print(t)  
    return inner  
outer("boo!")()
```

```
t = "surprise!"  
def inner():  
    print(t)  
def outer(t):  
    return inner  
outer("boo!")()
```

Draw environment diagrams to see what's different

Global Frame

t: "surprise!"

outer: func outer(t) [P = G]



```
t = "surprise!"
def outer(t):
    def inner():
        print(t)
    return inner
outer("boo!")()
```

```
t = "surprise!"
def inner():
    print(t)
def outer(t):
    return inner
outer("boo!")()
```

Draw environment diagrams to see what's different

Global Frame

t: "surprise!"  
outer: func outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
inner: func inner() [P = f1]  
rv: inner



```
t = "surprise!"  
def outer(t):  
    def inner():  
        print(t)  
    return inner  
outer("boo!")()
```

```
t = "surprise!"  
def inner():  
    print(t)  
def outer(t):  
    return inner  
outer("boo!")()
```

Draw environment diagrams to see what's different

Global Frame

t: "surprise!"  
outer: func outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
inner: func inner() [P = f1]  
rv: inner

f2: inner [P = f1]



```

t = "surprise!"
def outer(t):
    def inner():
        print(t)
    return inner
outer("boo!")()

```

```

t = "surprise!"
def inner():
    print(t)
def outer(t):
    return inner
outer("boo!")()

```

Draw environment diagrams to see what's different

Global Frame

t: "surprise!"  
 outer: func outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
 inner: func inner() [P = f1]  
 rv: inner

f2: inner [P = f1]

Global Frame

t: "surprise!"  
 inner: func inner() [P = G]  
 outer: fun outer(t) [P = G]



```

t = "surprise!"
def outer(t):
    def inner():
        print(t)
    return inner
outer("boo!")()

```

```

t = "surprise!"
def inner():
    print(t)
def outer(t):
    return inner
outer("boo!")()

```

Draw environment diagrams to see what's different

Global Frame

t: "surprise!"  
 outer: func outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
 inner: func inner() [P = f1]  
 rv: inner

f2: inner [P = f1]

Global Frame

t: "surprise!"  
 inner: func inner() [P = G]  
 outer: fun outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
 rv: inner



```

t = "surprise!"
def outer(t):
    def inner():
        print(t)
    return inner
outer("boo!")()

```

```

t = "surprise!"
def inner():
    print(t)
def outer(t):
    return inner
outer("boo!")()

```

Draw environment diagrams to see what's different

Global Frame

t: "surprise!"  
outer: func outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
inner: func inner() [P = f1]  
rv: inner

f2: inner [P = f1]

Global Frame

t: "surprise!"  
inner: func inner() [P = G]  
outer: fun outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
rv: inner

f2: inner [P = G]



```
t = "surprise!"
def outer(t):
    def inner():
        print(t)
    return inner
outer("boo!")()
```

```
t = "surprise!"
def inner():
    print(t)
def outer(t):
    return inner
outer("boo!")()
```

Draw environment diagrams to see what's different

Global Frame

t: "surprise!"  
outer: func outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
inner: func inner() [P = f1]  
rv: inner

f2: inner [P = f1]

Global Frame

t: "surprise!"  
inner: func inner() [P = G]  
outer: fun outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
rv: inner

f2: inner [P = G]

All inner does is print(t).  
Since t is not defined in  
the **local** frame, where  
does inner find t?



```
t = "surprise!"
def outer(t):
    def inner():
        print(t)
    return inner
outer("boo!")()
```

```
t = "surprise!"
def inner():
    print(t)
def outer(t):
    return inner
outer("boo!")()
```

Draw environment diagrams to see what's different

Global Frame

t: "surprise!"  
outer: func outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
inner: func inner() [P = f1]  
rv: inner

f2: inner [P = f1]

Global Frame

t: "surprise!"  
inner: func inner() [P = G]  
outer: fun outer(t) [P = G]

f1: outer [P = G]

t: "boo!"  
rv: inner

f2: inner [P = G]

Python prints:

boo!

All inner does is print(t).  
Since t is not defined in  
the **local** frame, where  
does inner find t?

Python prints:

surprise!



# Environment Diagrams

Know the rules!

## 1. Def statements:

1. create a new function whose parent is the current frame
2. skip the body of the function
3. bind the function to its name in the current frame

## 2. Assignment statements:

1. evaluate the RHS
2. bind the value of the RHS to the name on the LHS
3. NOTE: names can only have one value per frame

## 3. Function calls:

1. evaluate the operator
2. evaluate the operands
3. execute the body of the function



# 1.5 #1

```
def curry2(h):  
    def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    return f  
  
make_adder = curry2(lambda x, y: x + y)  
  
add_three = make_adder(3)  
  
five = add_three(2)
```



# 1.5 #1

```
→ def curry2(h):  
    def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    return f  
  
make_adder = curry2(lambda x, y: x + y)  
  
add_three = make_adder(3)  
  
five = add_three(2)
```

Global Frame:


curry2 → func curry2(h) [P = G]



# 1.5 #1

Global Frame:

curry2 → func curry2(h) [P = G]



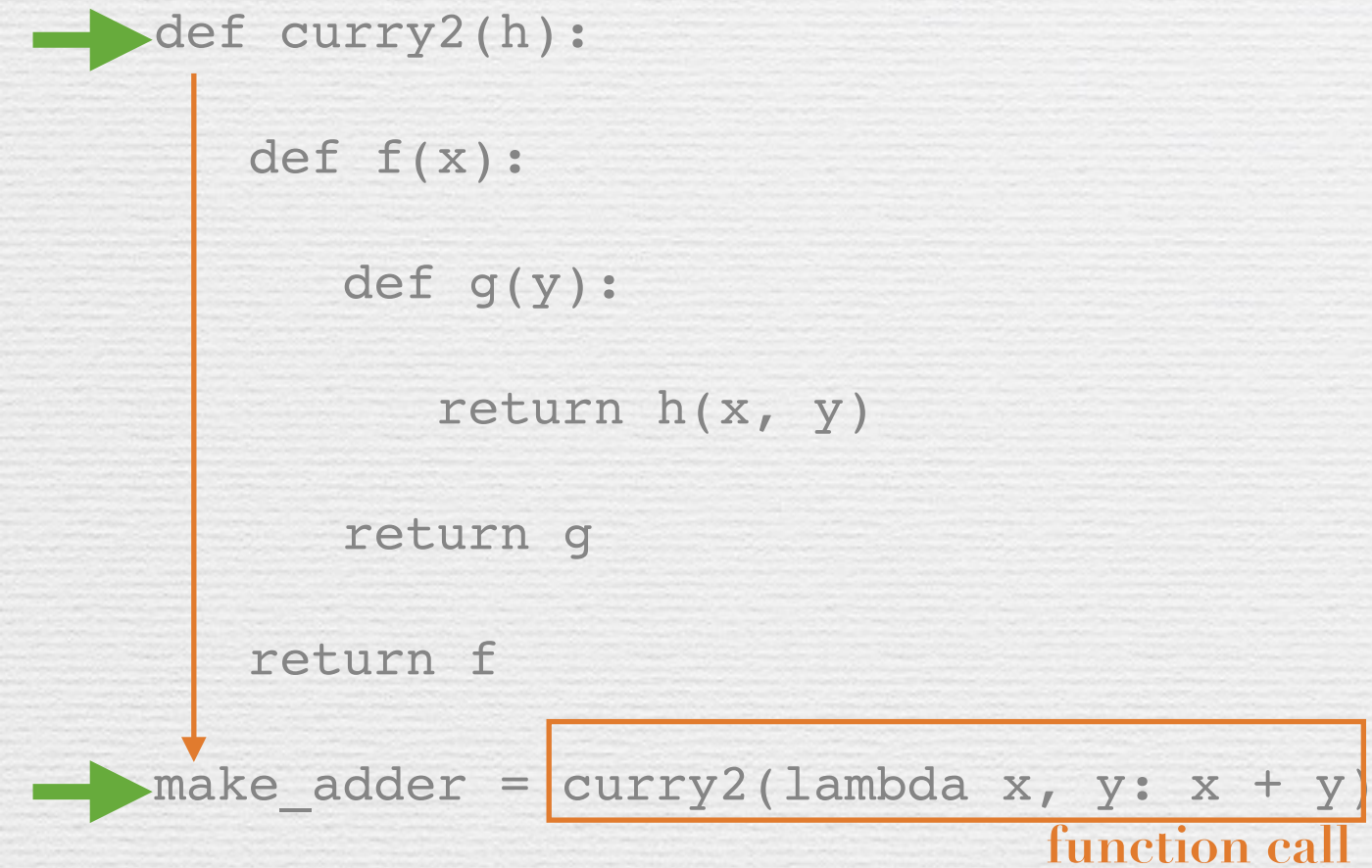
```
def curry2(h):  
    def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    return f  
make_adder = curry2(lambda x, y: x + y)
```

```
add_three = make_adder(3)
```

```
five = add_three(2)
```



# 1.5 #1



```
def curry2(h):  
    def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    return f  
  
make_adder = curry2(lambda x, y: x + y)  
  
add_three = make_adder(3)  
  
five = add_three(2)
```

function call

Global Frame:

curry2 → func curry2(h) [P = G]

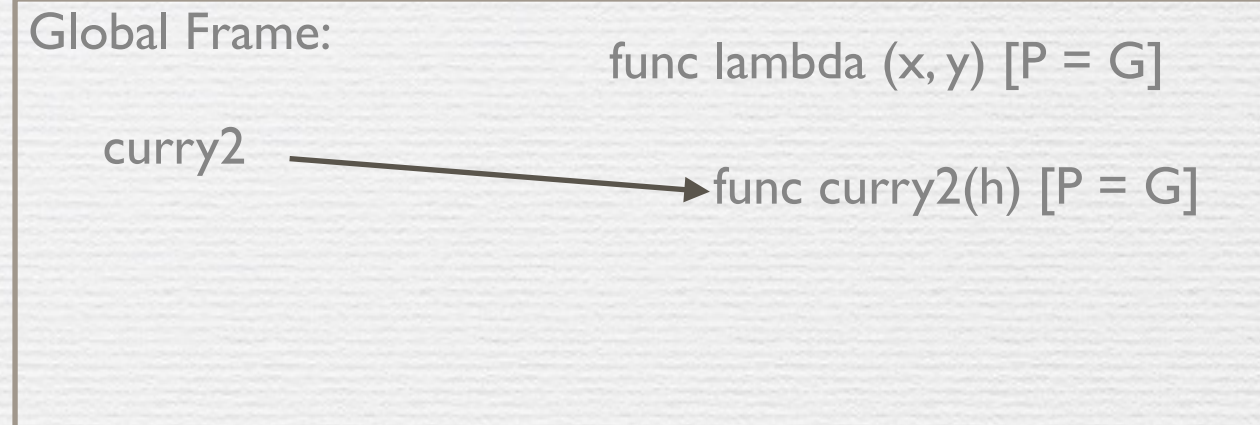


# 1.5 #1

```
→ def curry2(h):  
    def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    return f  
→ make_adder = curry2(lambda x, y: x + y)  
                                function call
```

```
add_three = make_adder(3)
```

```
five = add_three(2)
```



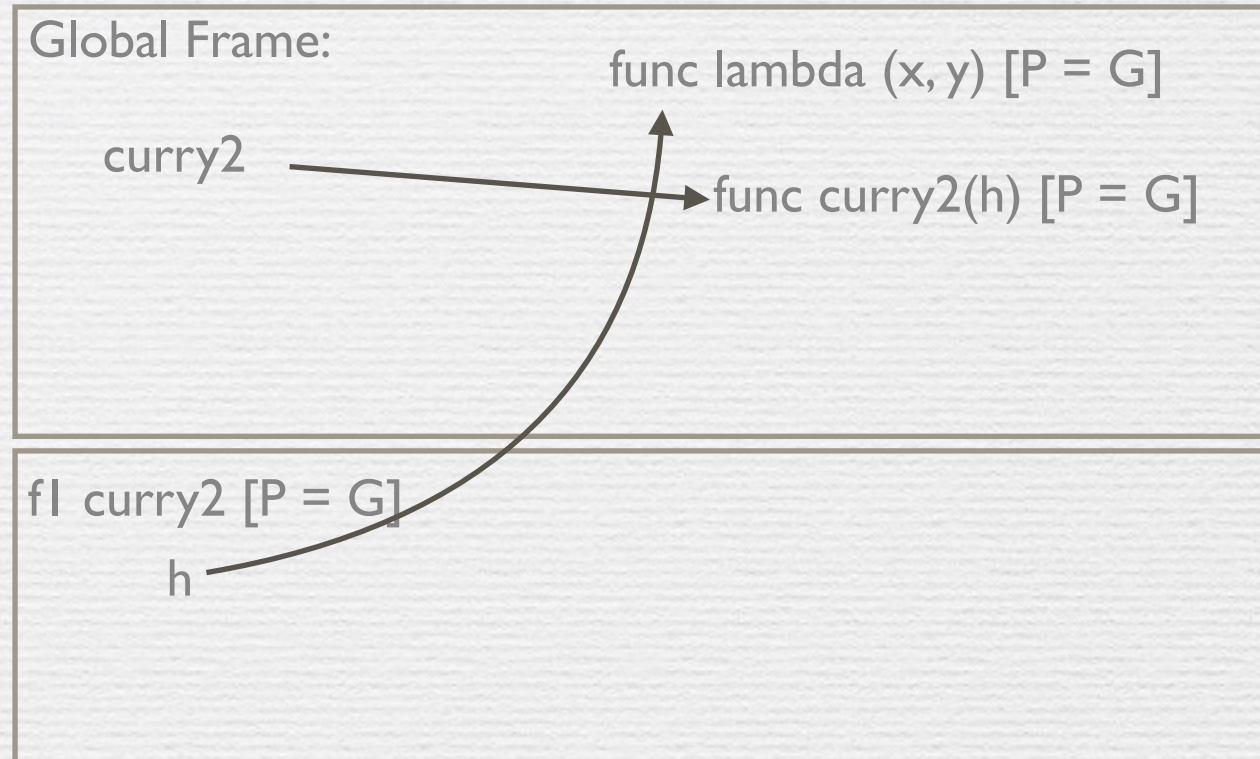


# 1.5 #1

```
→ def curry2(h):  
    def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    return f  
→ make_adder = curry2(lambda x, y: x + y)  
                    function call
```

```
add_three = make_adder(3)
```

```
five = add_three(2)
```



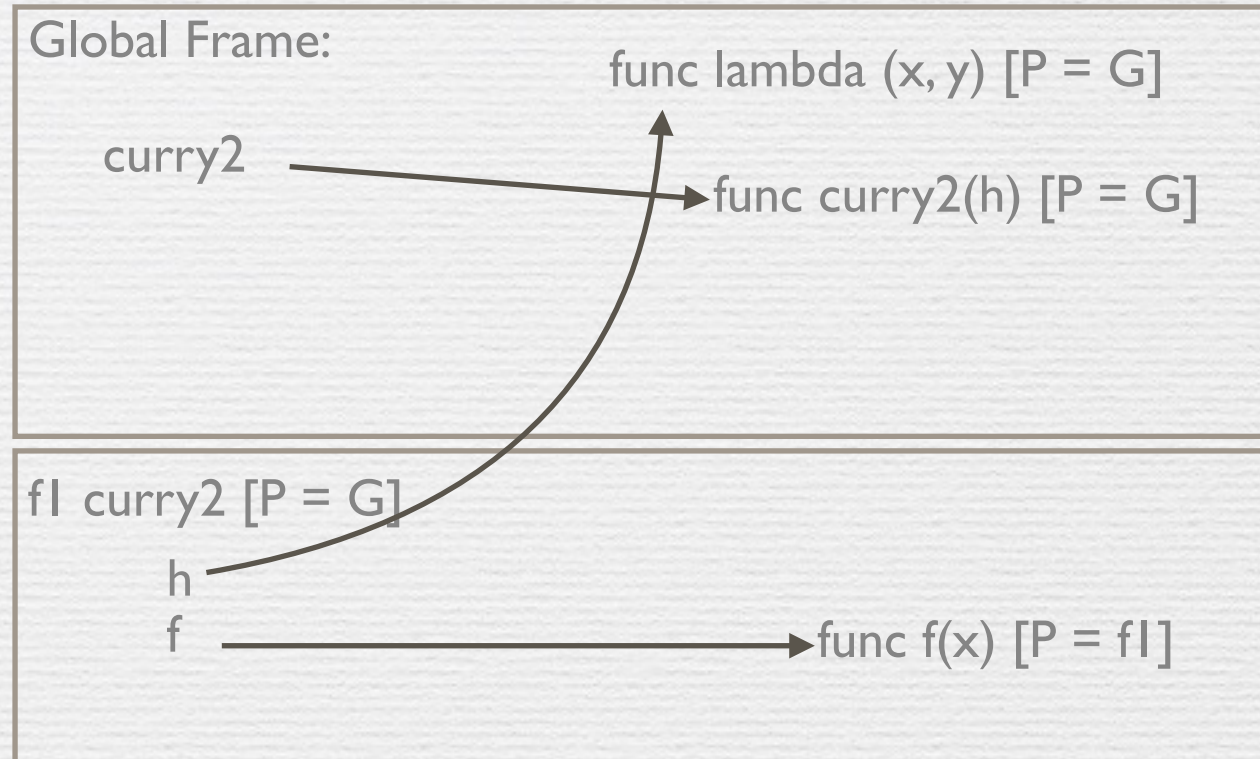


# 1.5 #1

```
→ def curry2(h):  
    → def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    return f  
→ make_adder = curry2(lambda x, y: x + y)  
                    function call
```

```
add_three = make_adder(3)
```

```
five = add_three(2)
```



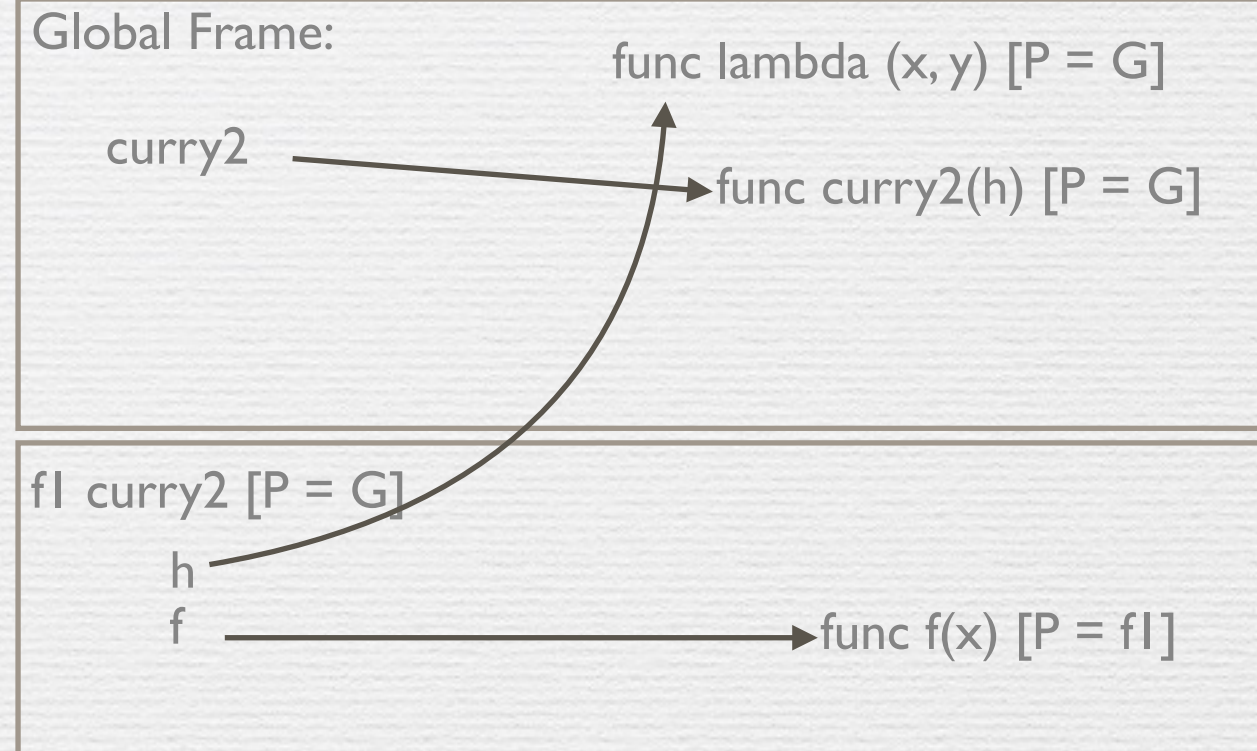


# 1.5 #1

```
→ def curry2(h):  
    → def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    return f  
→ make_adder = curry2(lambda x, y: x + y)  
                    function call
```

```
add_three = make_adder(3)
```

```
five = add_three(2)
```



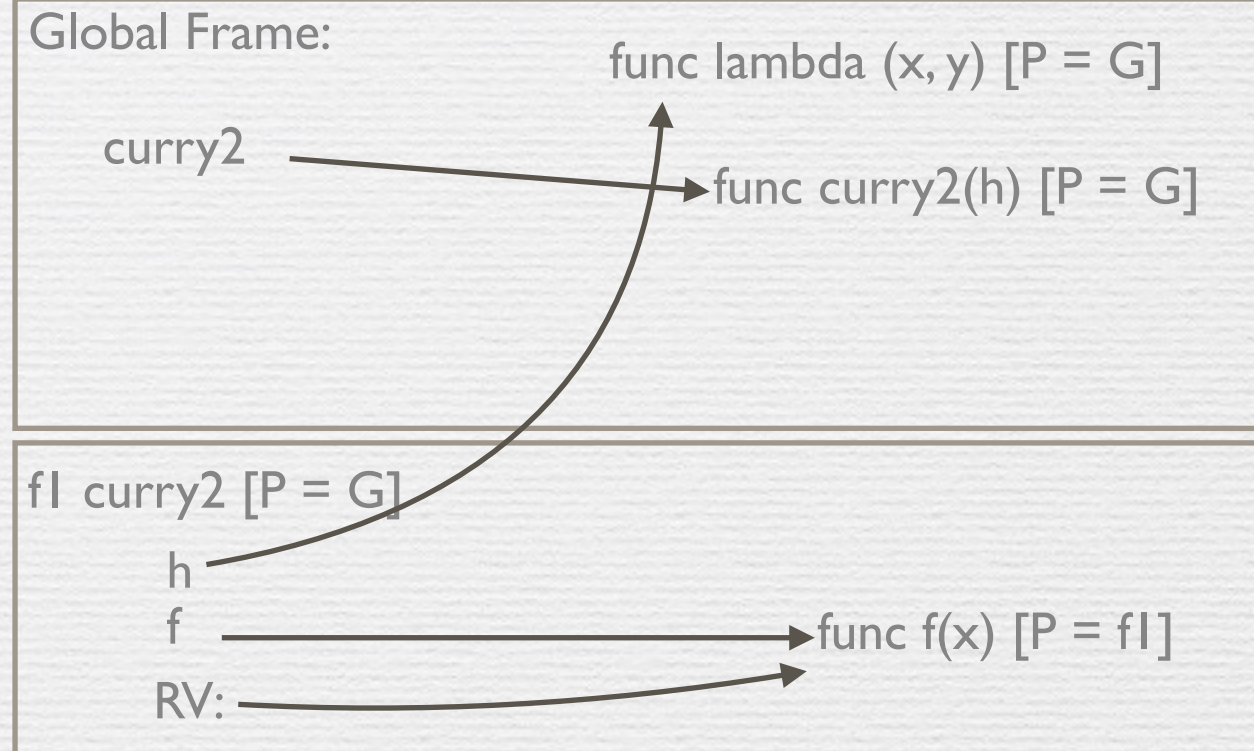


# 1.5 #1

```
→ def curry2(h):  
    → def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    → return f  
→ make_adder = curry2(lambda x, y: x + y)  
                    function call
```

```
add_three = make_adder(3)
```

```
five = add_three(2)
```





# 1.5 #1

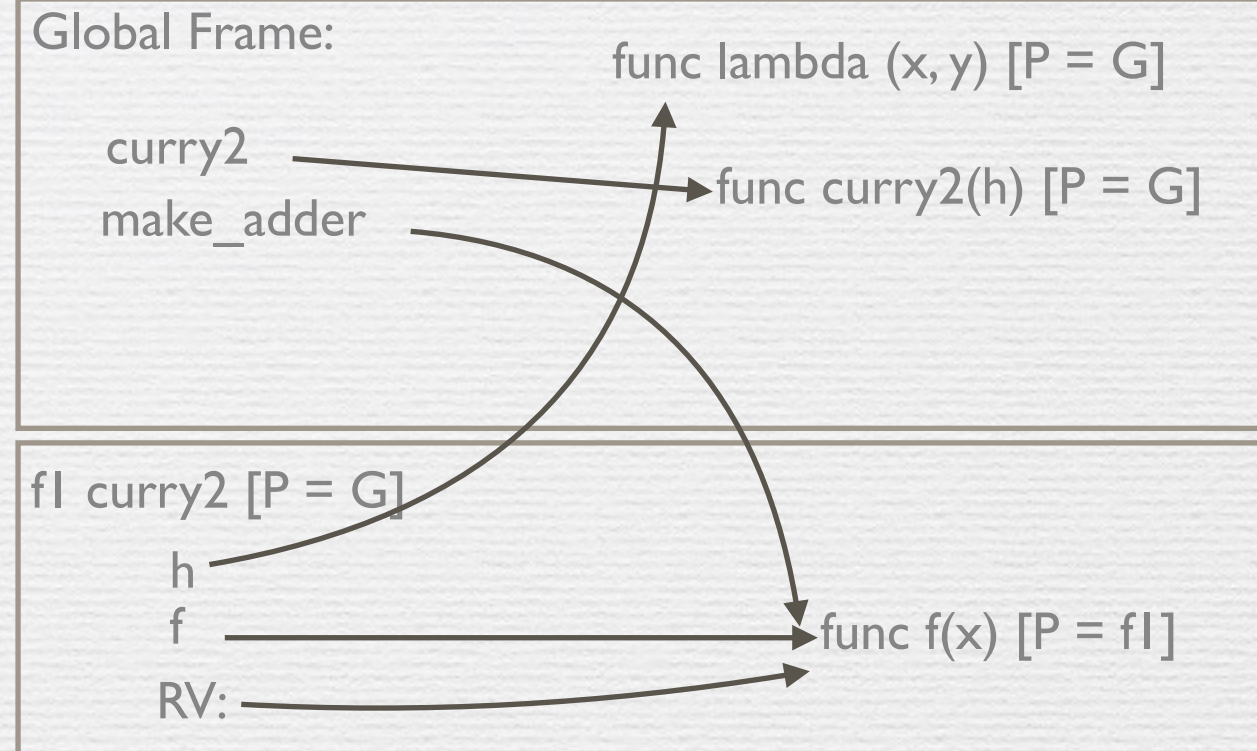
```
→ def curry2(h):  
    → def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    → return f  
→ make_adder = curry2(lambda x, y: x + y)
```

assignment

function call

```
add_three = make_adder(3)
```

```
five = add_three(2)
```





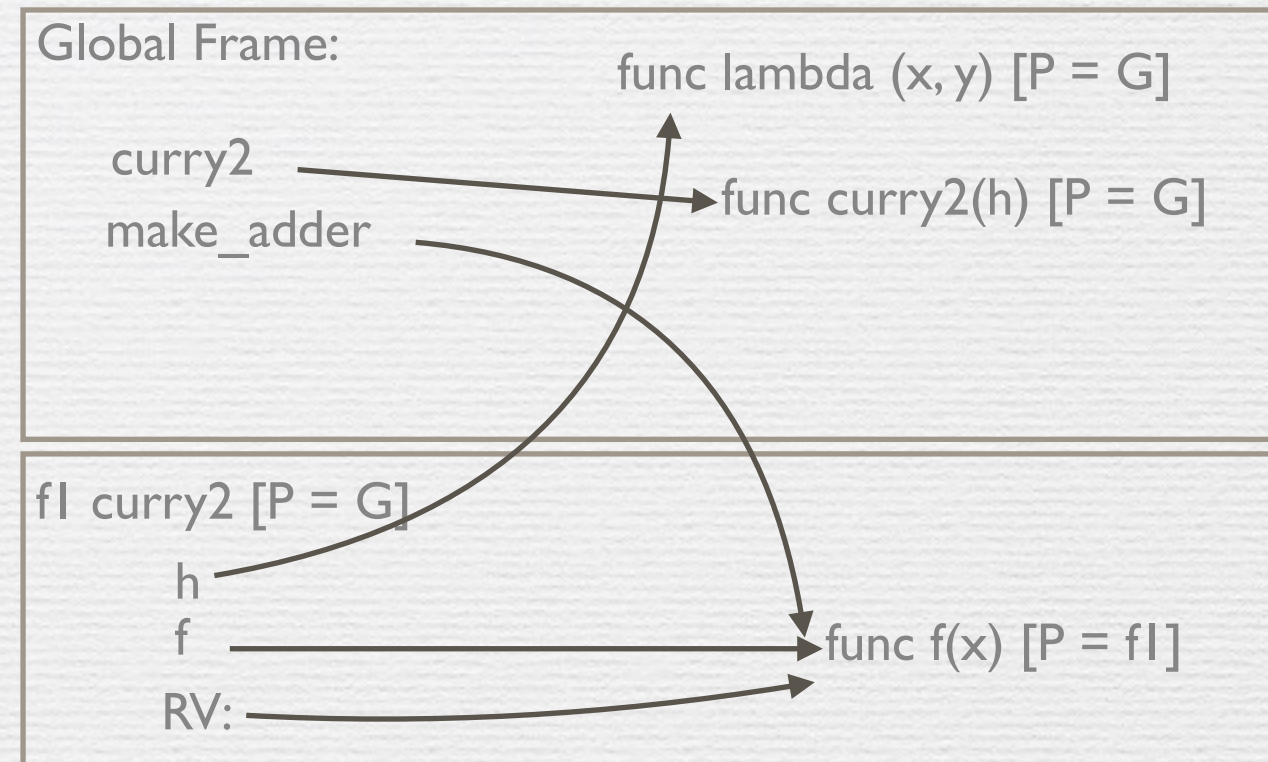
# 1.5 #1

```
→ def curry2(h):  
    → def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    → return f  
→ make_adder = curry2(lambda x, y: x + y)  
→ add_three = make_adder(3)
```

assignment

function call

function call



```
five = add_three(2)
```



# 1.5 #1

```
→ def curry2(h):  
    → def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    → return f  
→ make_adder = curry2(lambda x, y: x + y)  
→ add_three = make_adder(3)
```

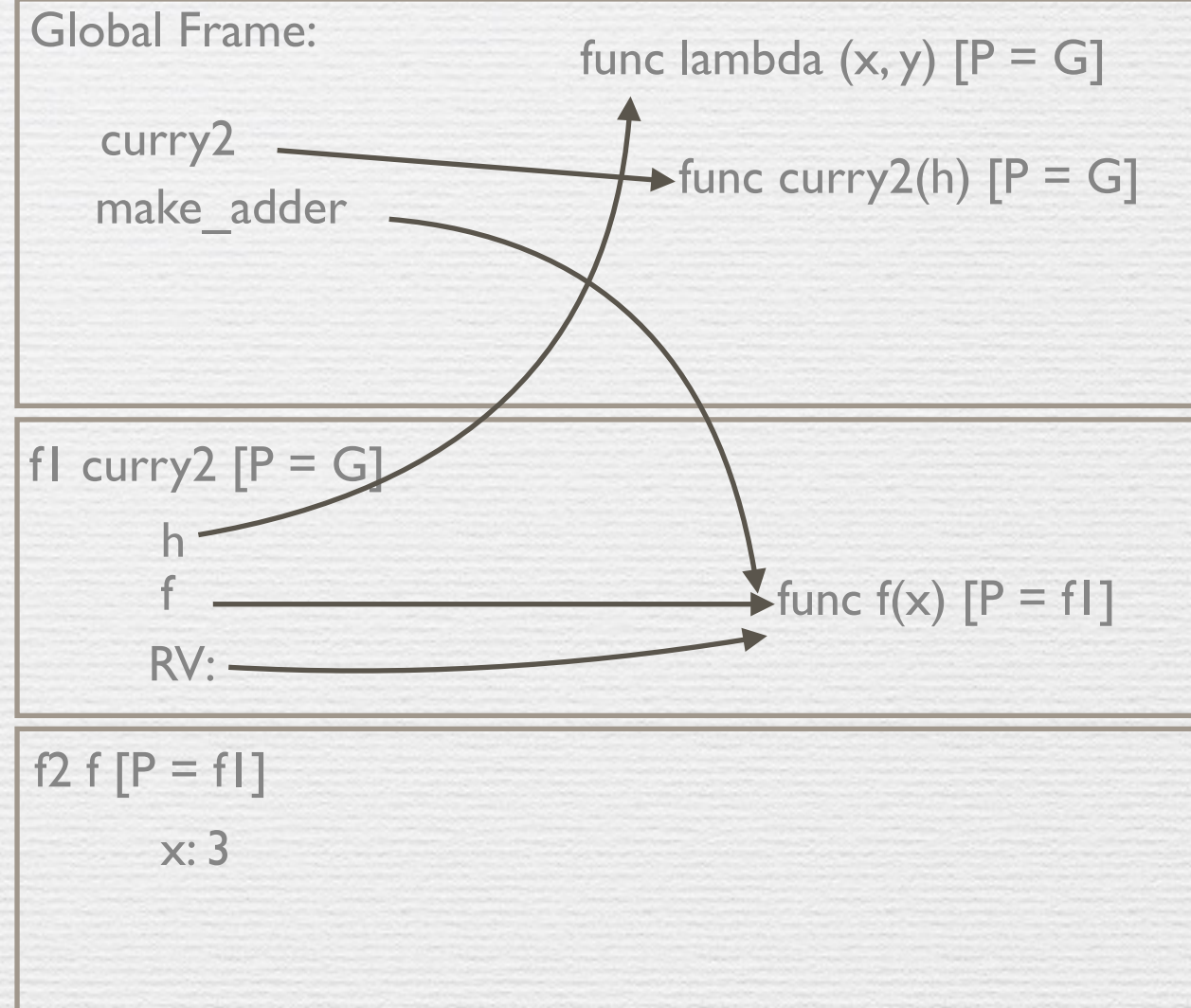
assignment

function call

function call

```
five = add_three(2)
```

tip: when you start doing a function call, mark where you were before so that you know which line to go back to





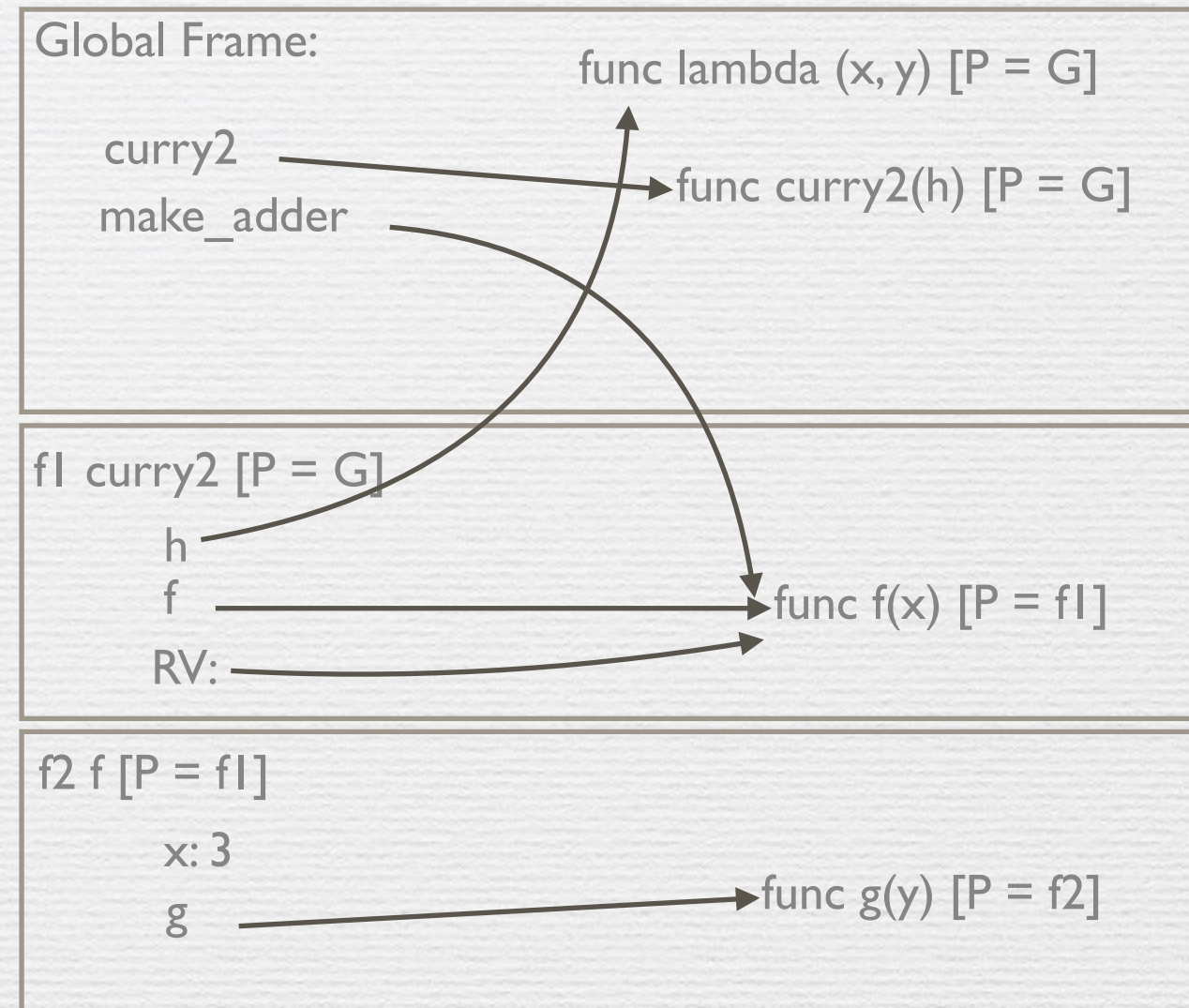
# 1.5 #1

```
→ def curry2(h):  
    → def f(x):  
        → def g(y):  
            return h(x, y)  
        return g  
    → return f  
→ make_adder = curry2(lambda x, y: x + y)  
→ add_three = make_adder(3)
```

assignment

function call

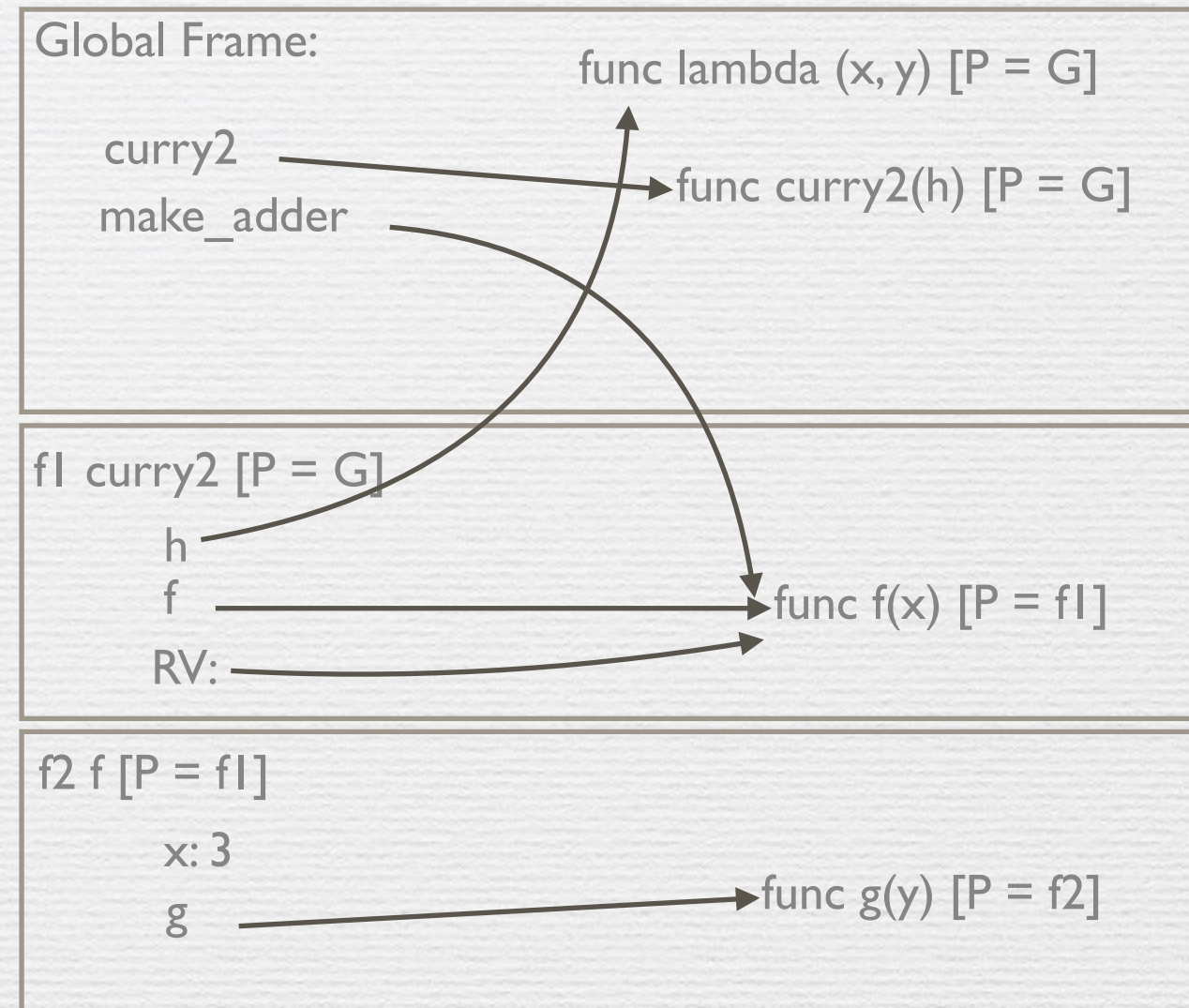
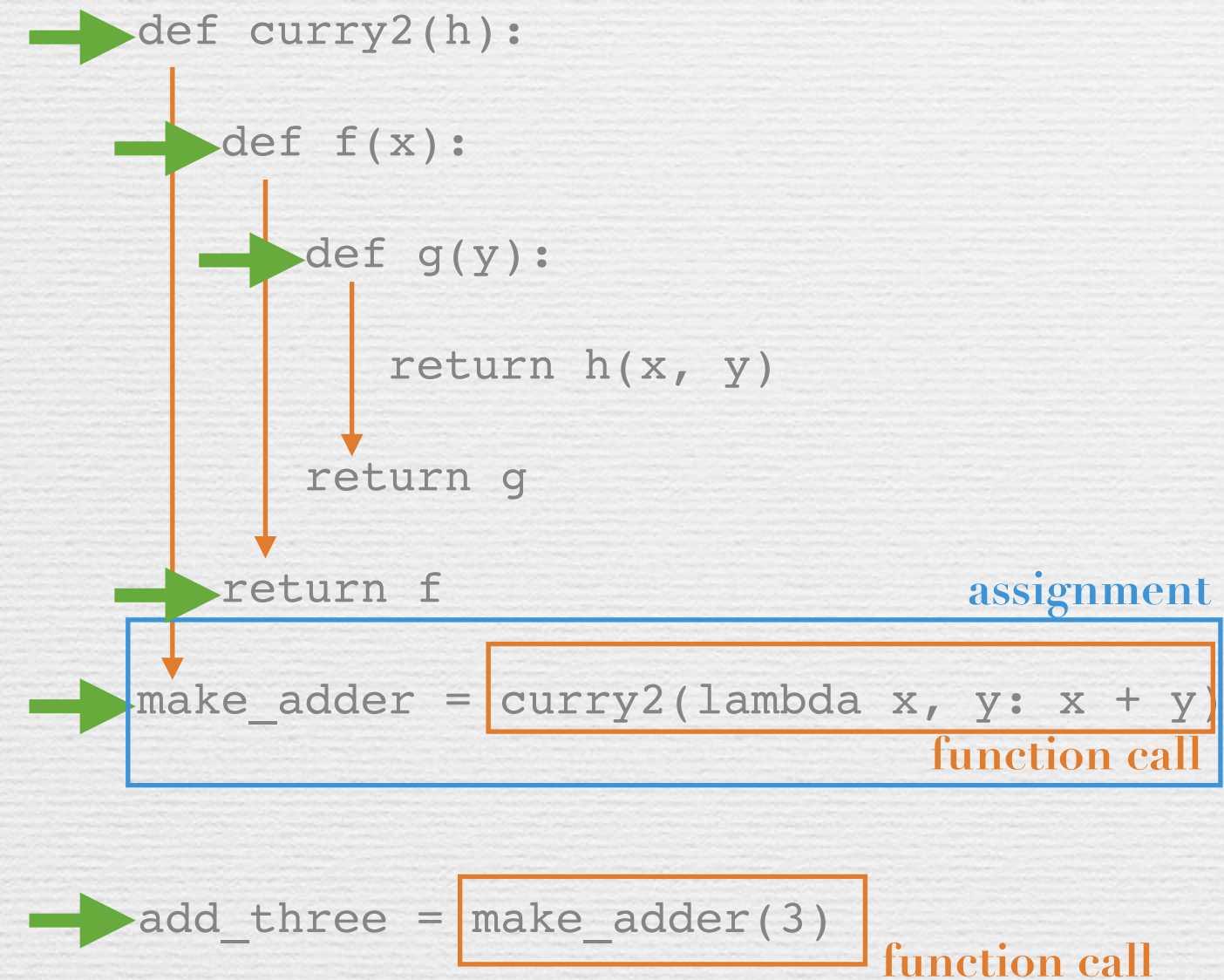
function call



```
five = add_three(2)
```



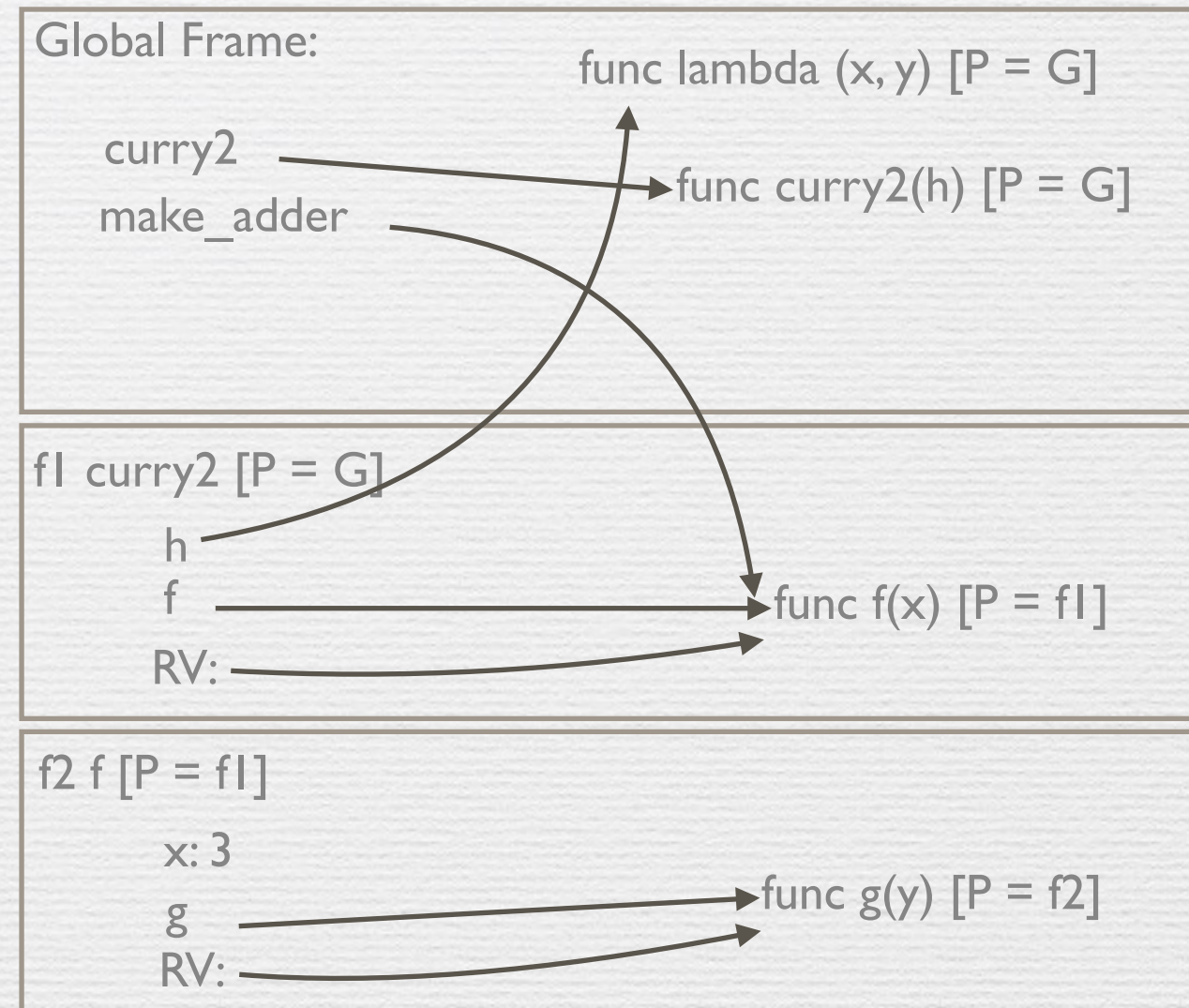
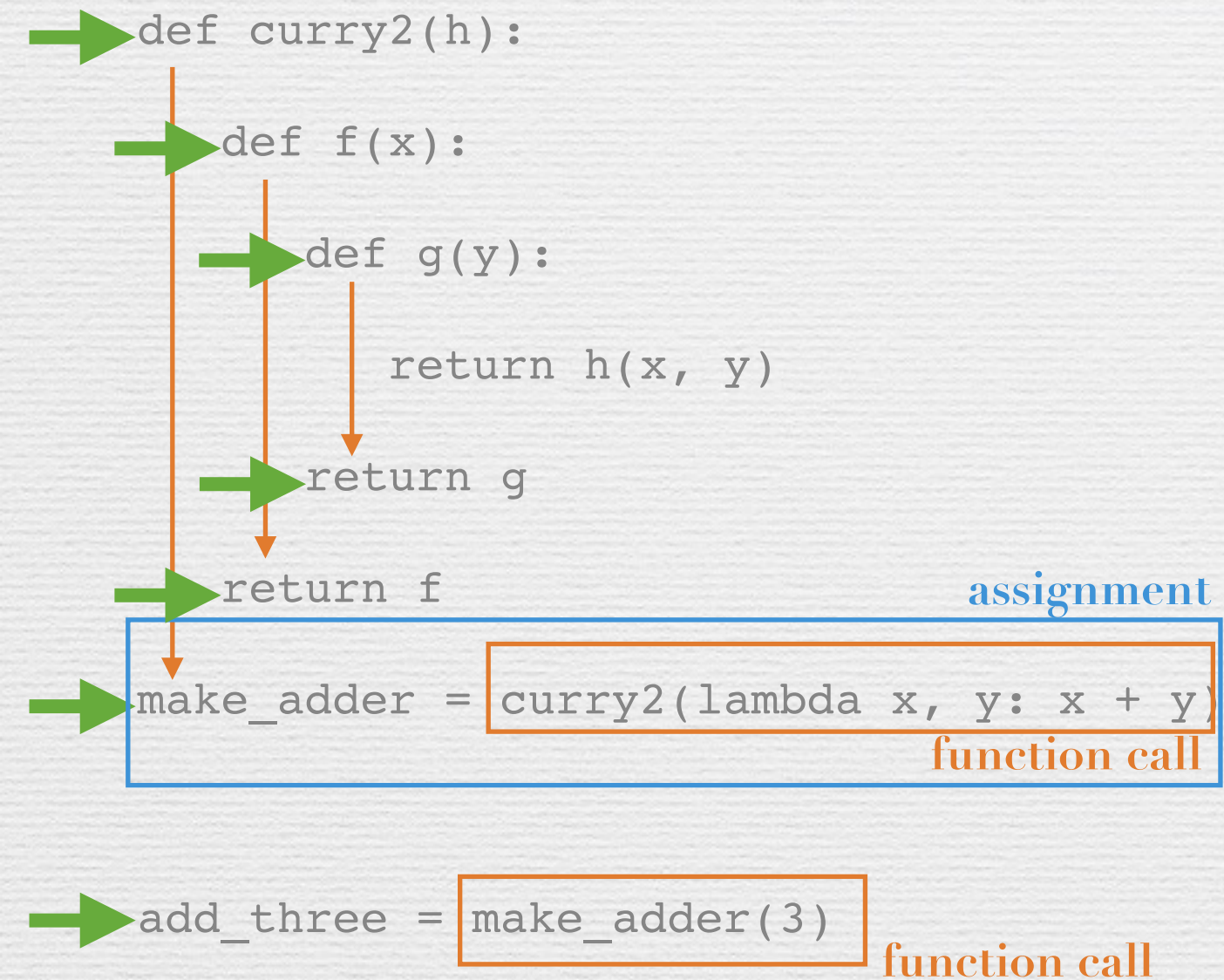
# 1.5 #1



five = add\_three(2)



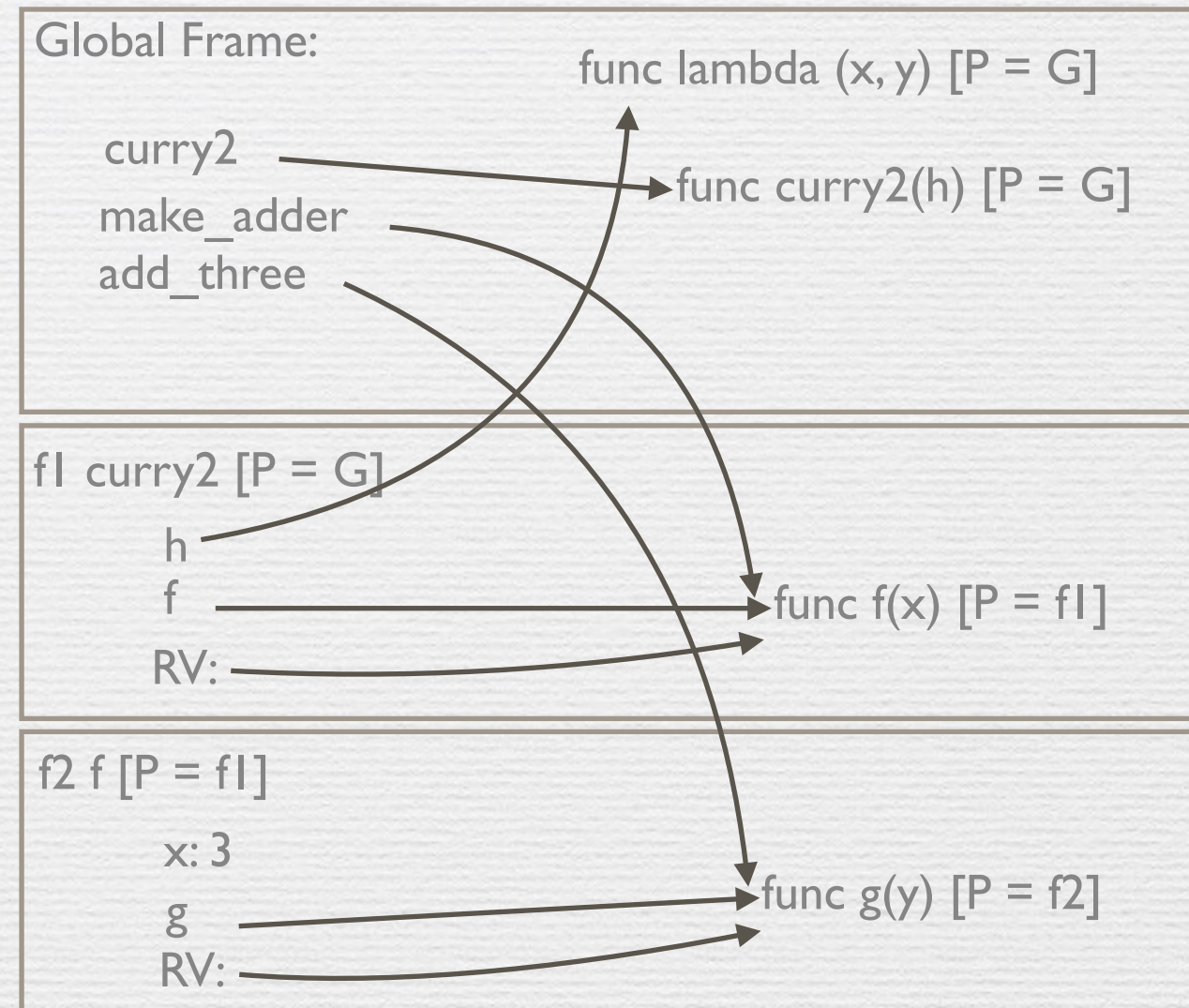
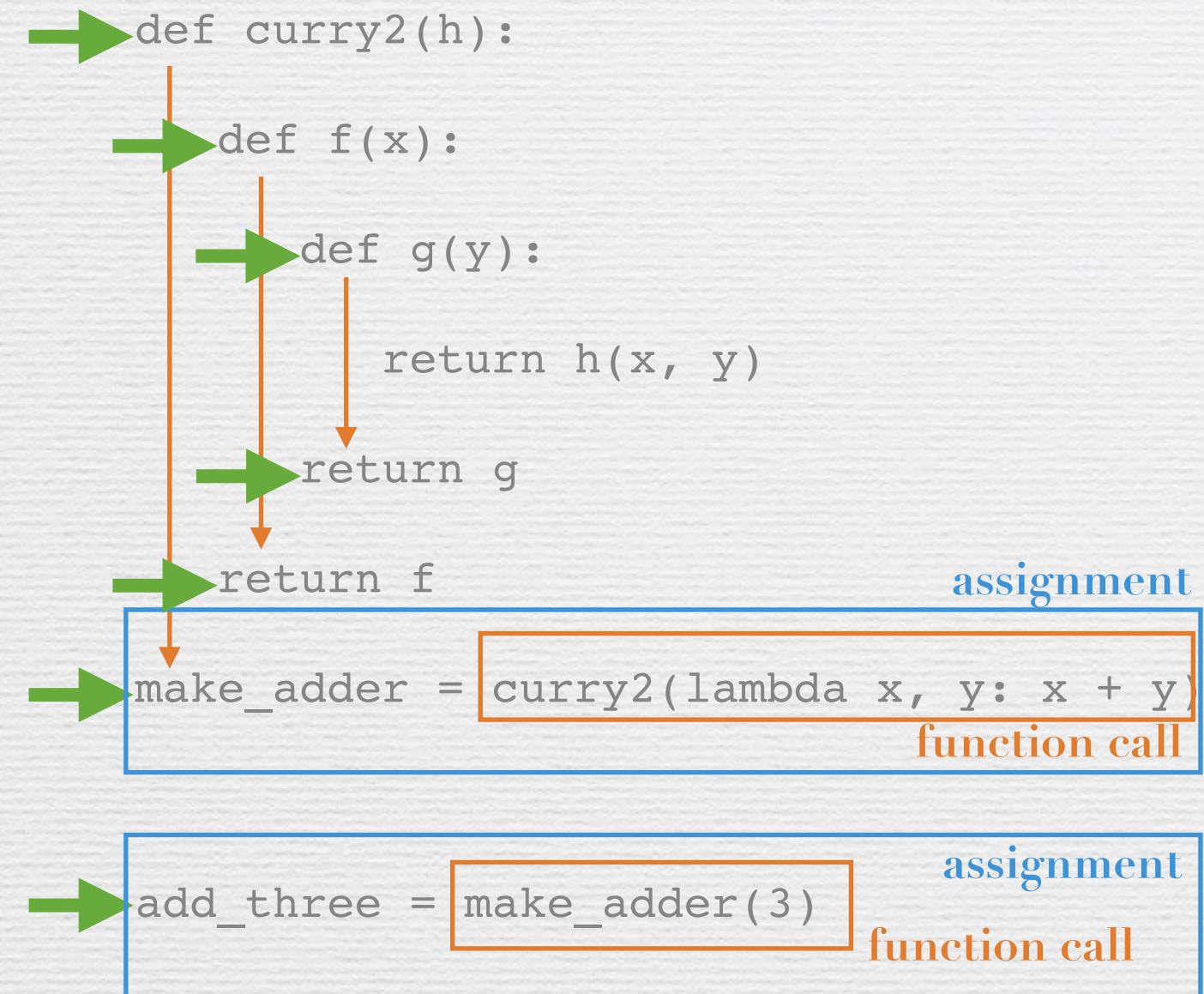
# 1.5 #1



`five = add_three(2)`



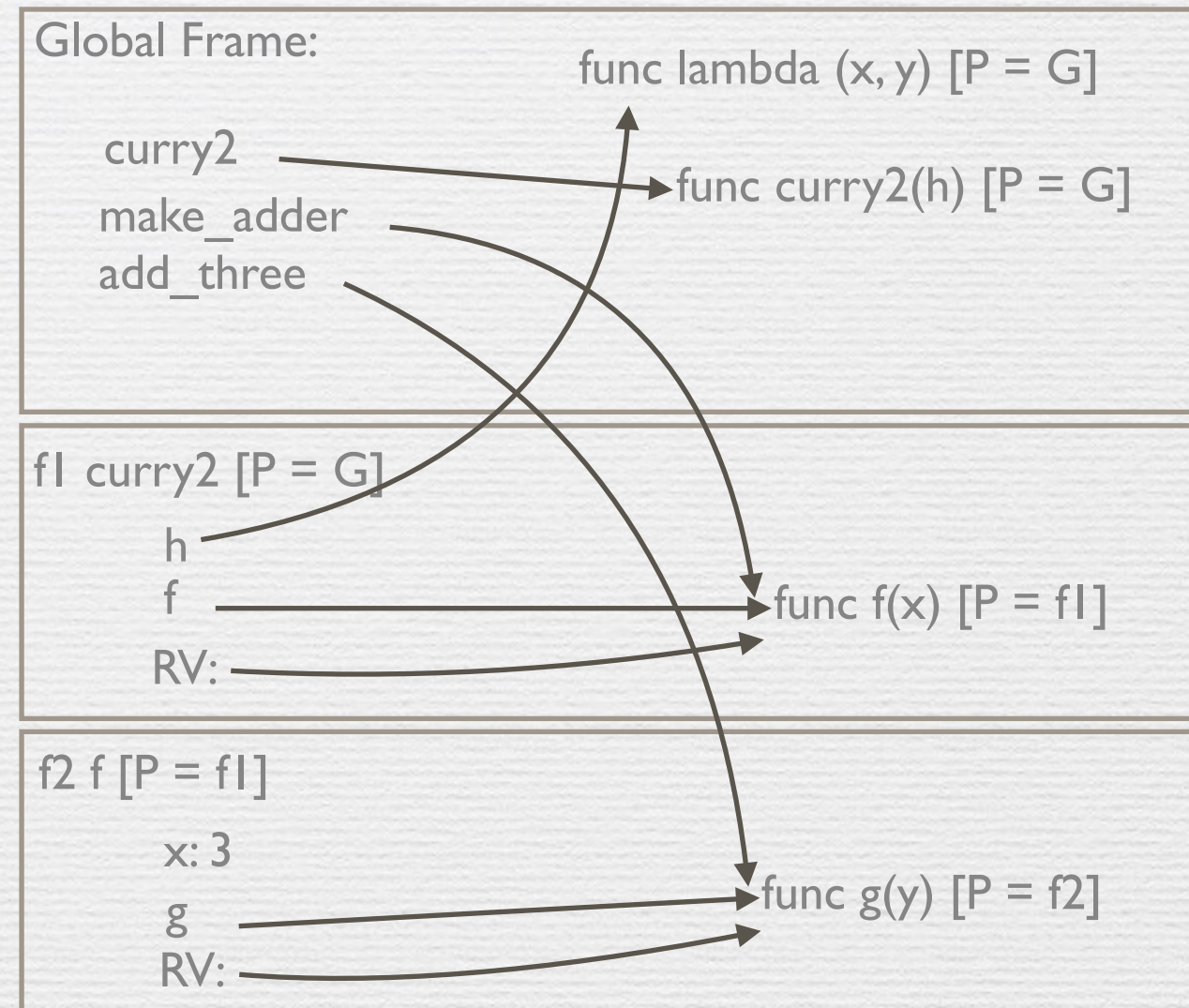
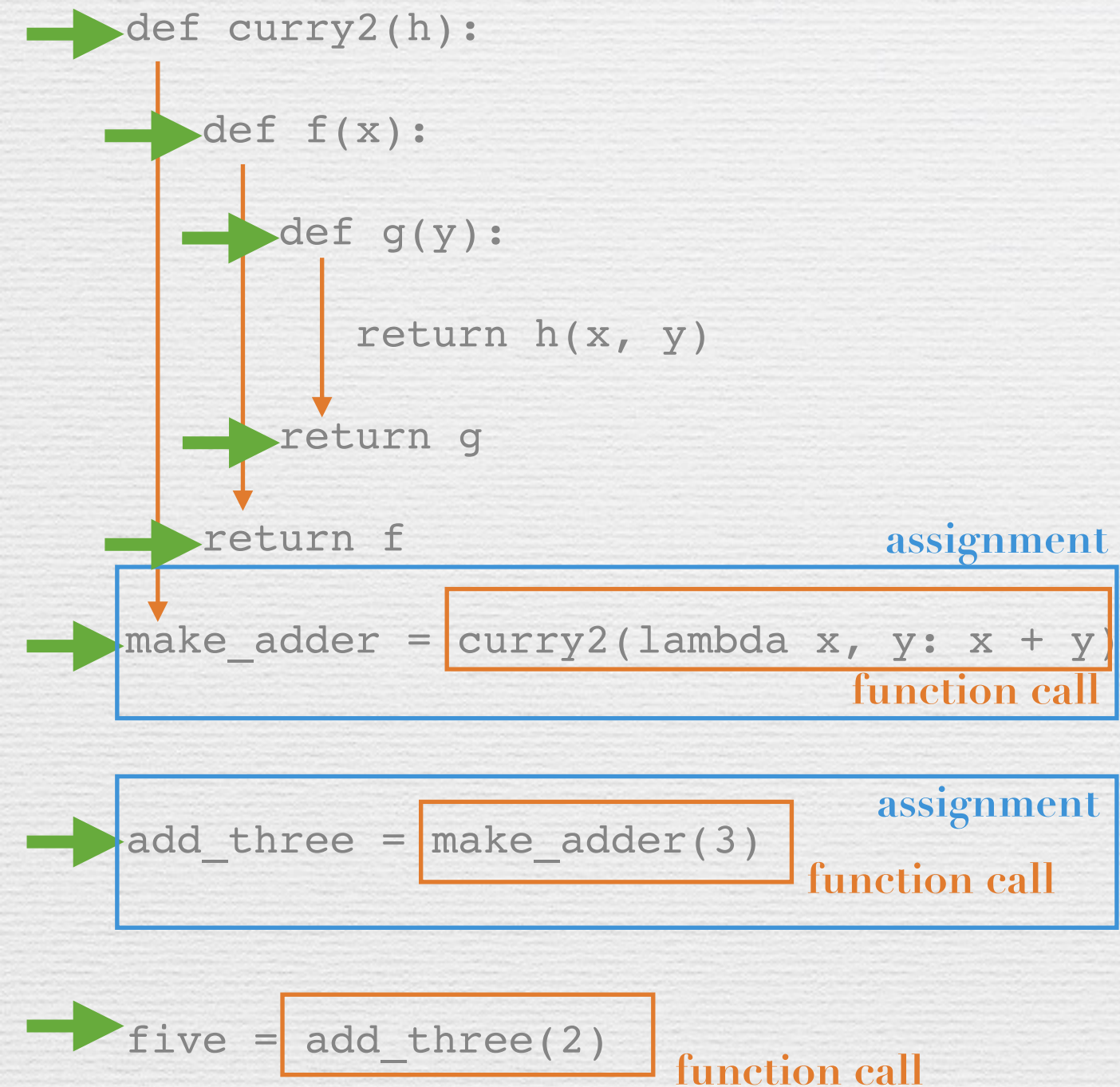
# 1.5 #1



`five = add_three(2)`

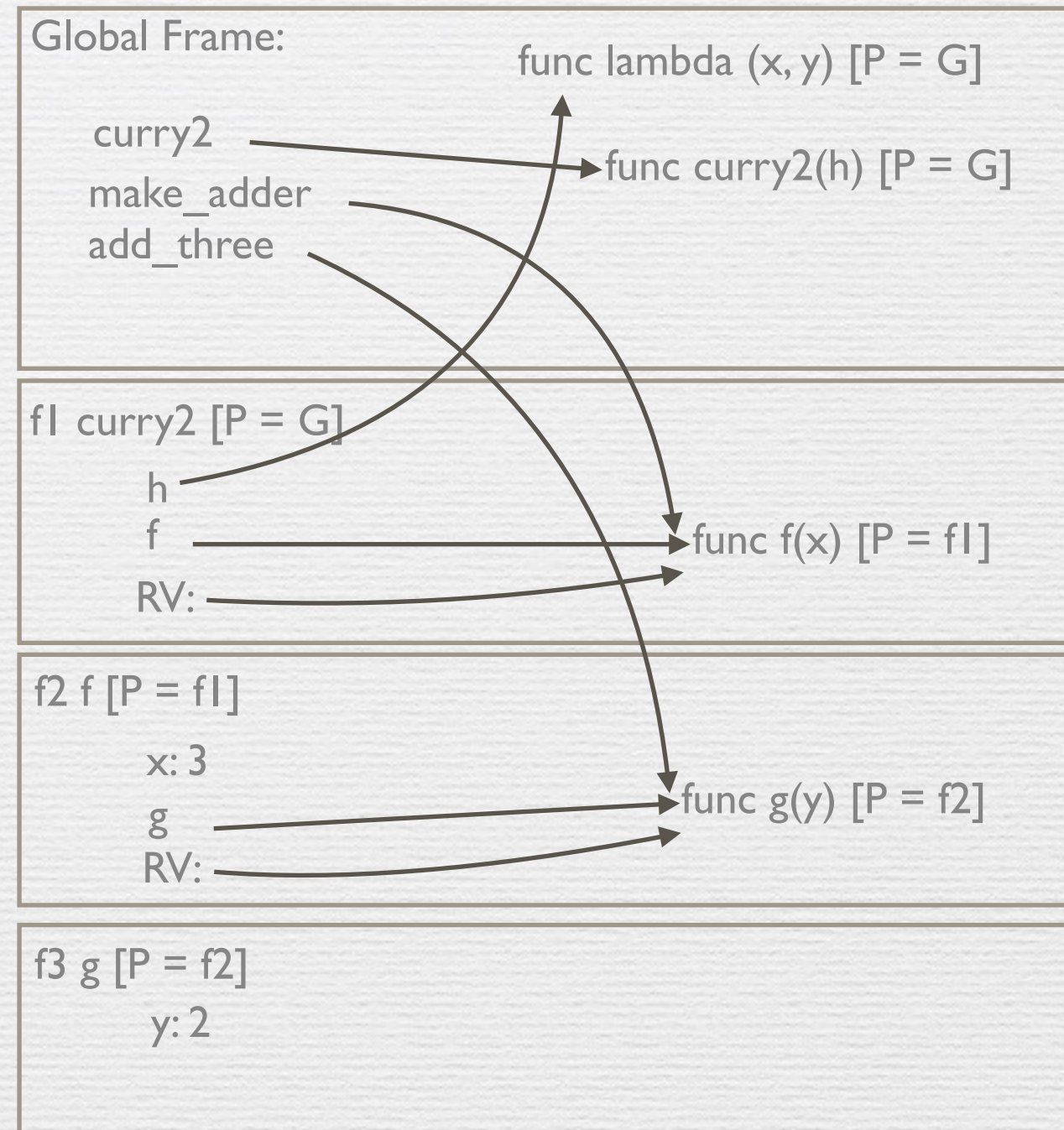
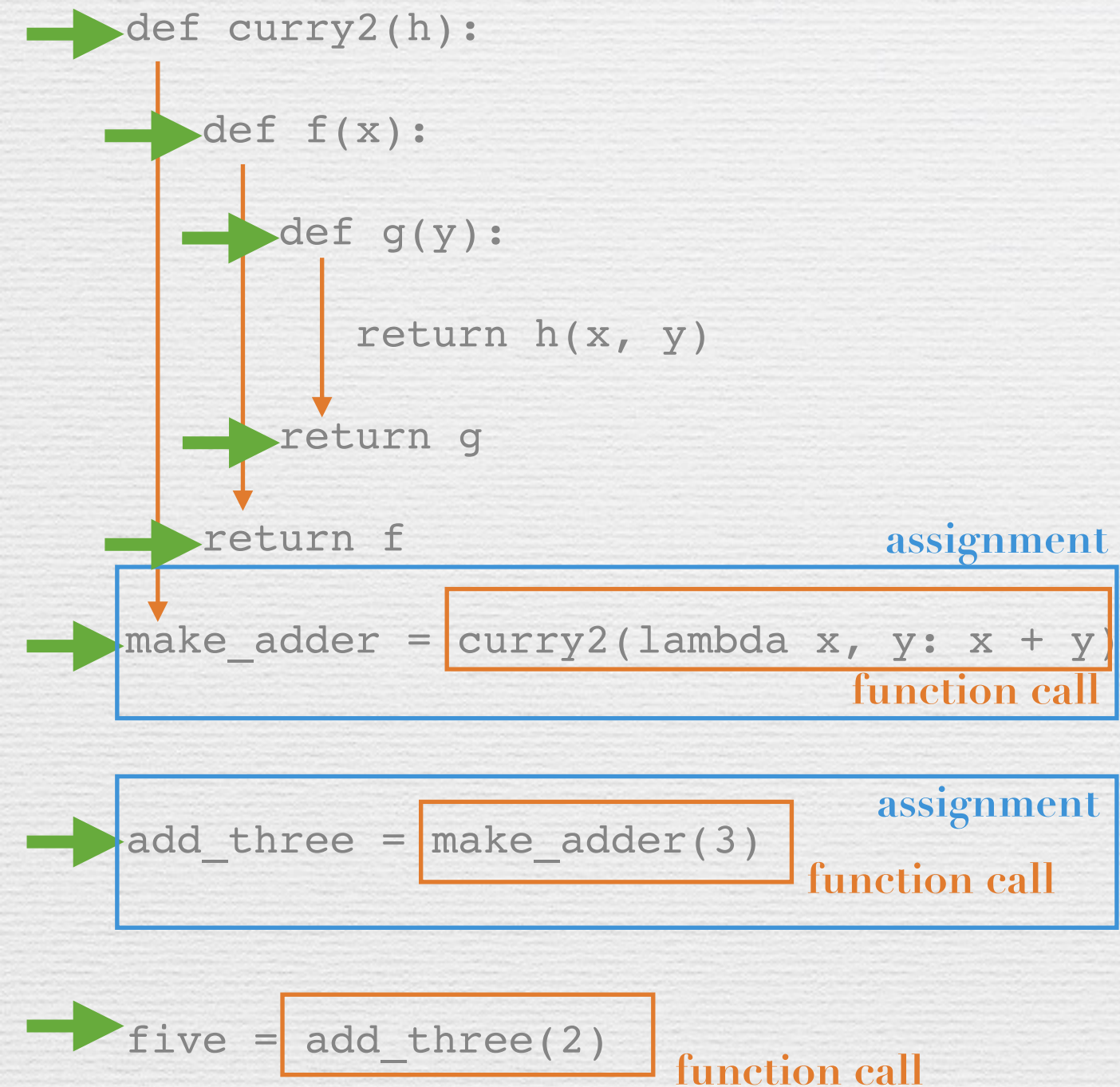


# 1.5 #1





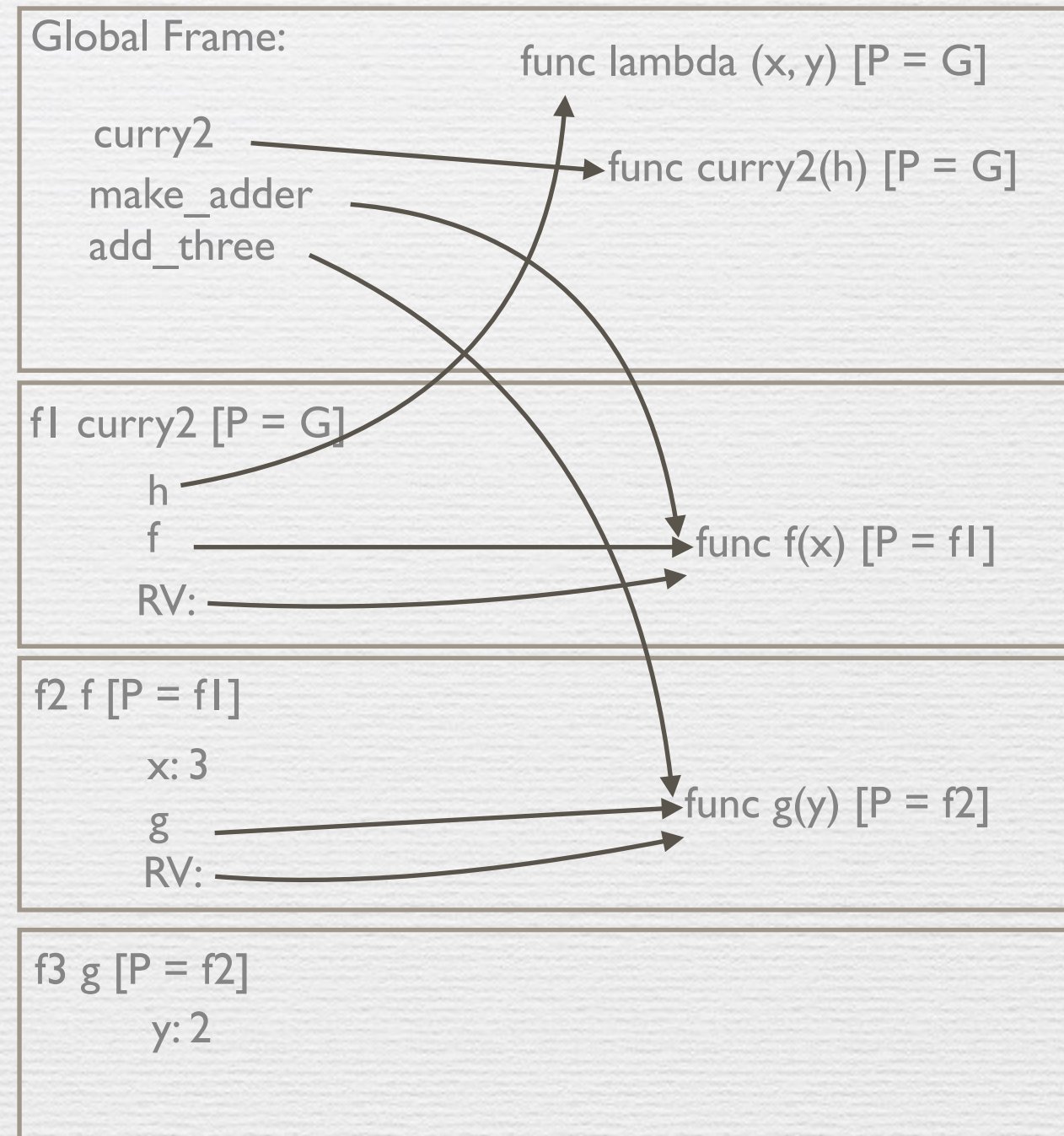
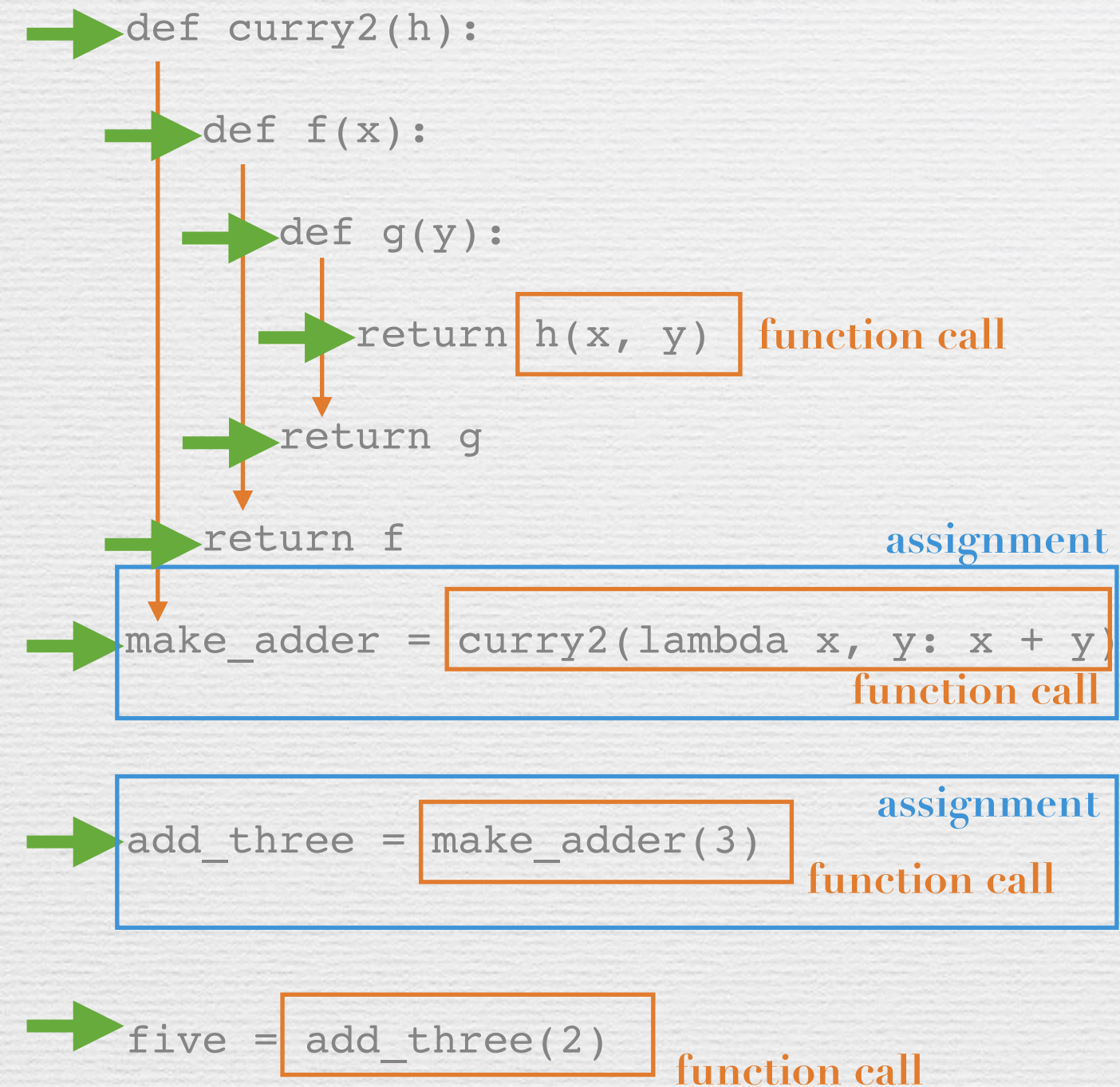
# 1.5 #1



tip: when you start doing a function call, mark where you were before so that you know which line to go back to



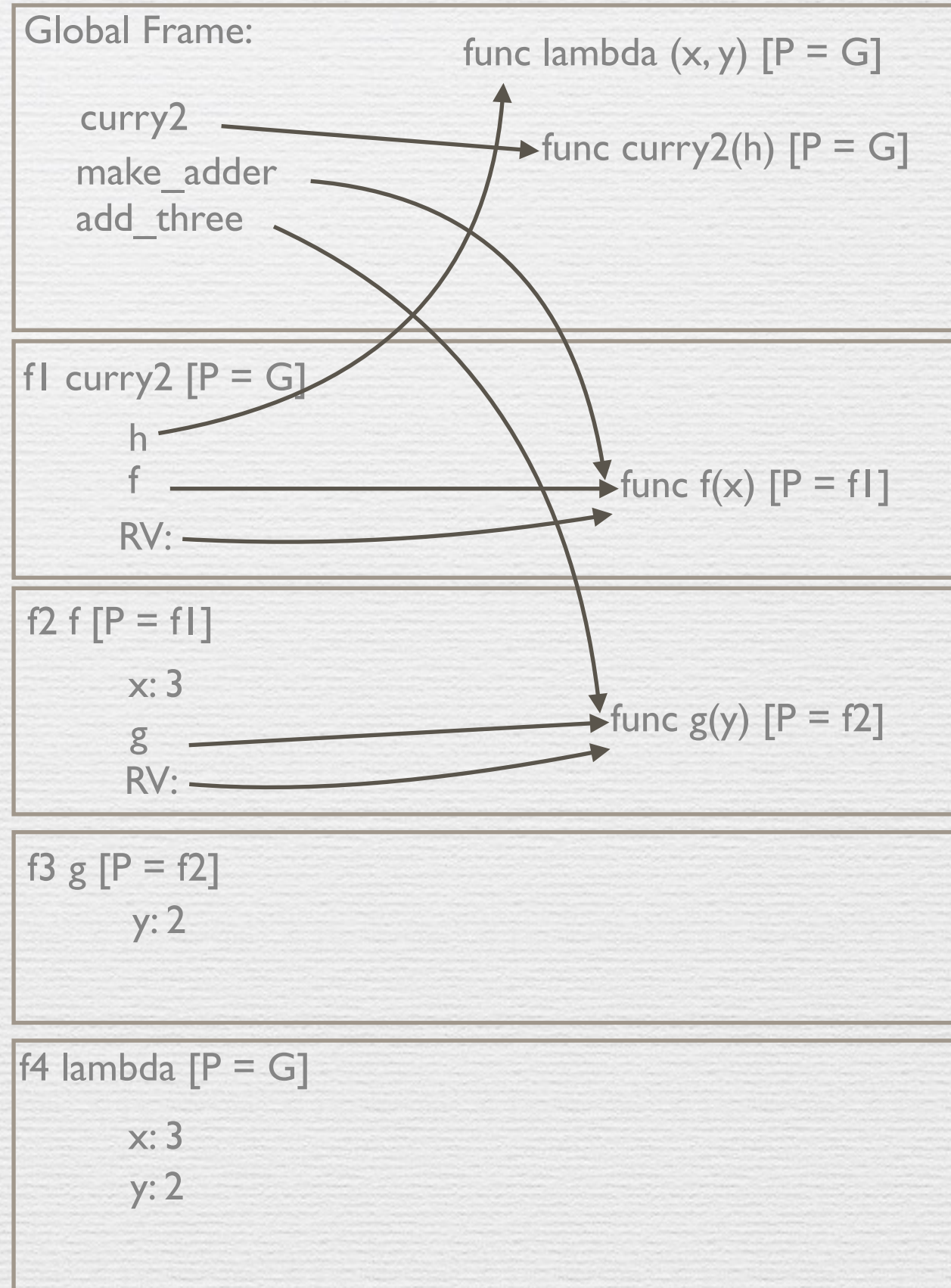
# 1.5 #1





# 1.5 #1

```
→ def curry2(h):  
    → def f(x):  
        → def g(y):  
            → return h(x, y) function call  
        → return g  
    → return f assignment  
→ make_adder = curry2(lambda x, y: x + y) function call  
  
→ add_three = make_adder(3) assignment  
  
→ five = add_three(2) function call
```



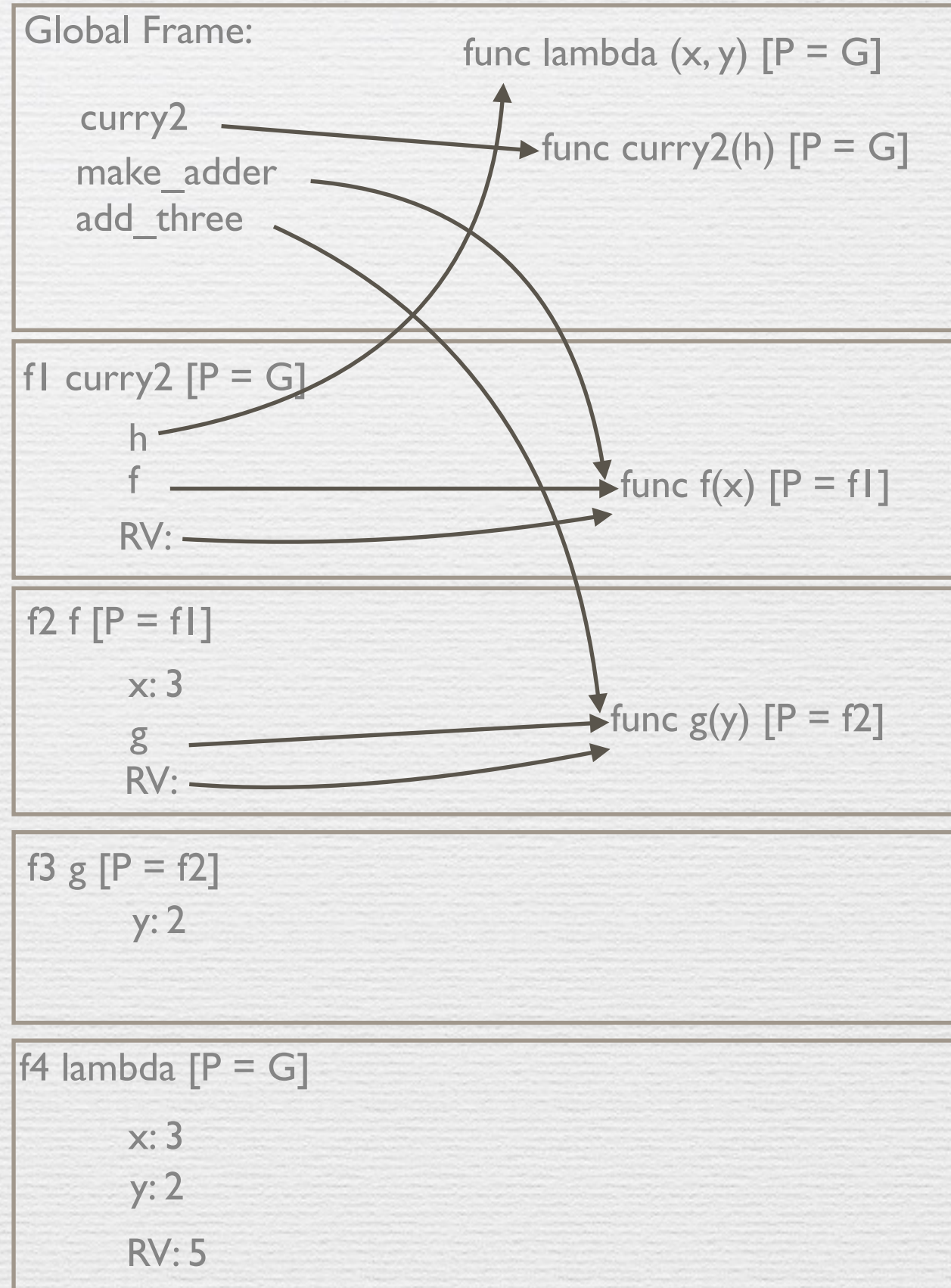
tip: when you start doing a function call, mark where you were before so that you know which line to go back to



# 1.5 #1

```
→ def curry2(h):  
    → def f(x):  
        → def g(y):  
            → return h(x, y) function call  
        → return g  
    → return f assignment  
→ make_adder = curry2(lambda x, y: x + y) function call  
  
→ add_three = make_adder(3) assignment  
  
→ five = add_three(2) function call
```

tip: when you start doing a function call, mark where you were before so that you know which line to go back to

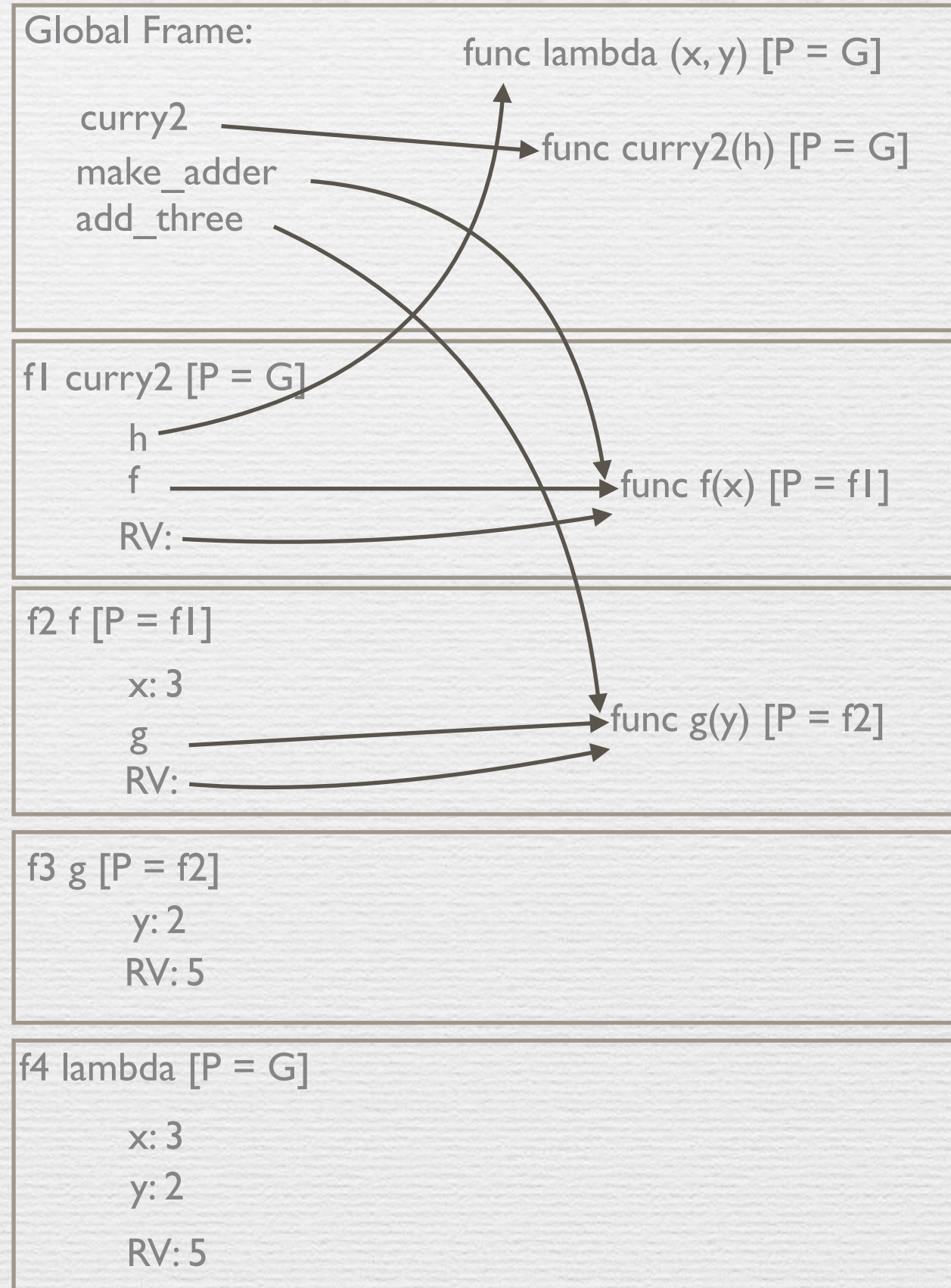




# 1.5 #1

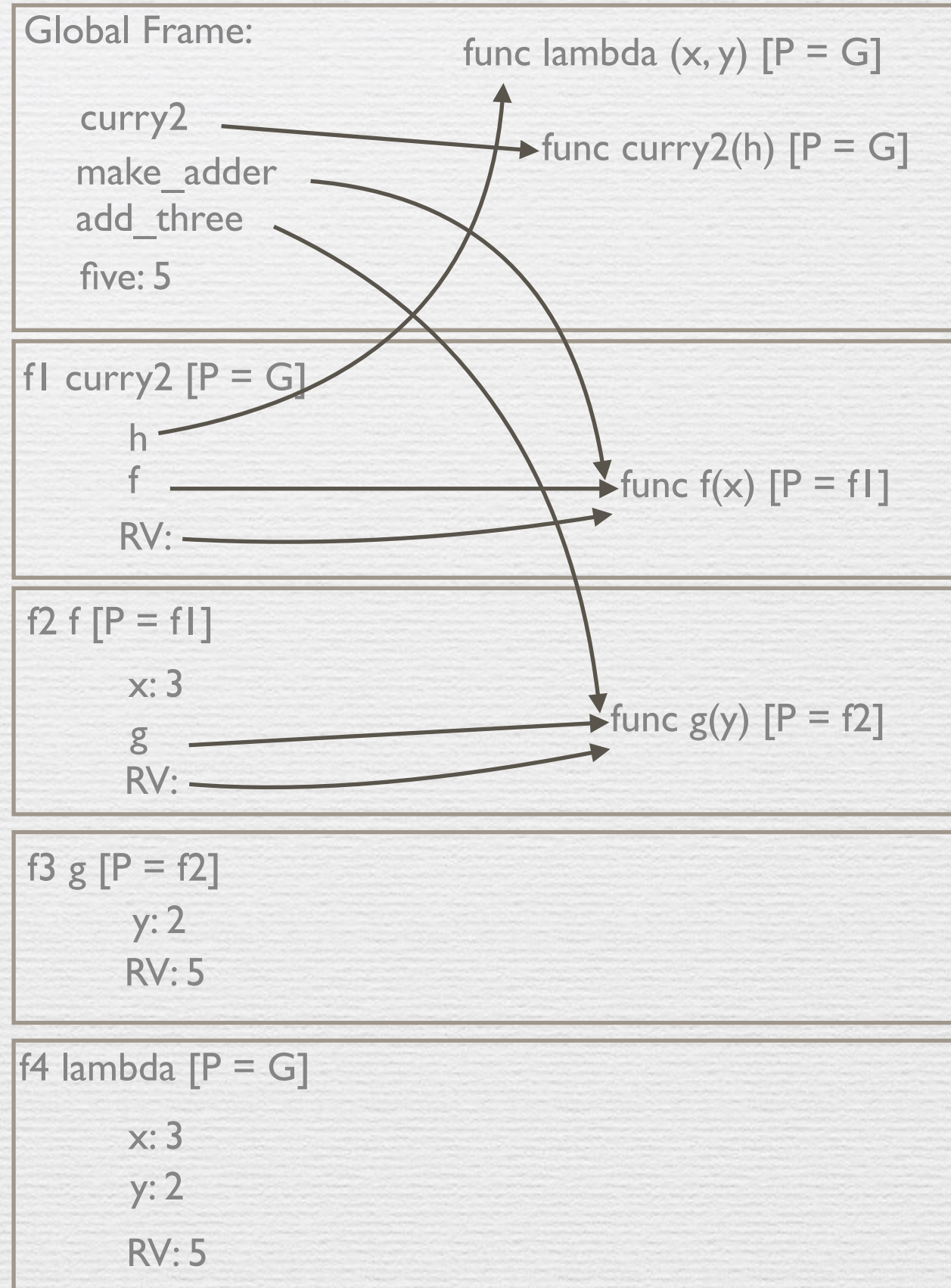
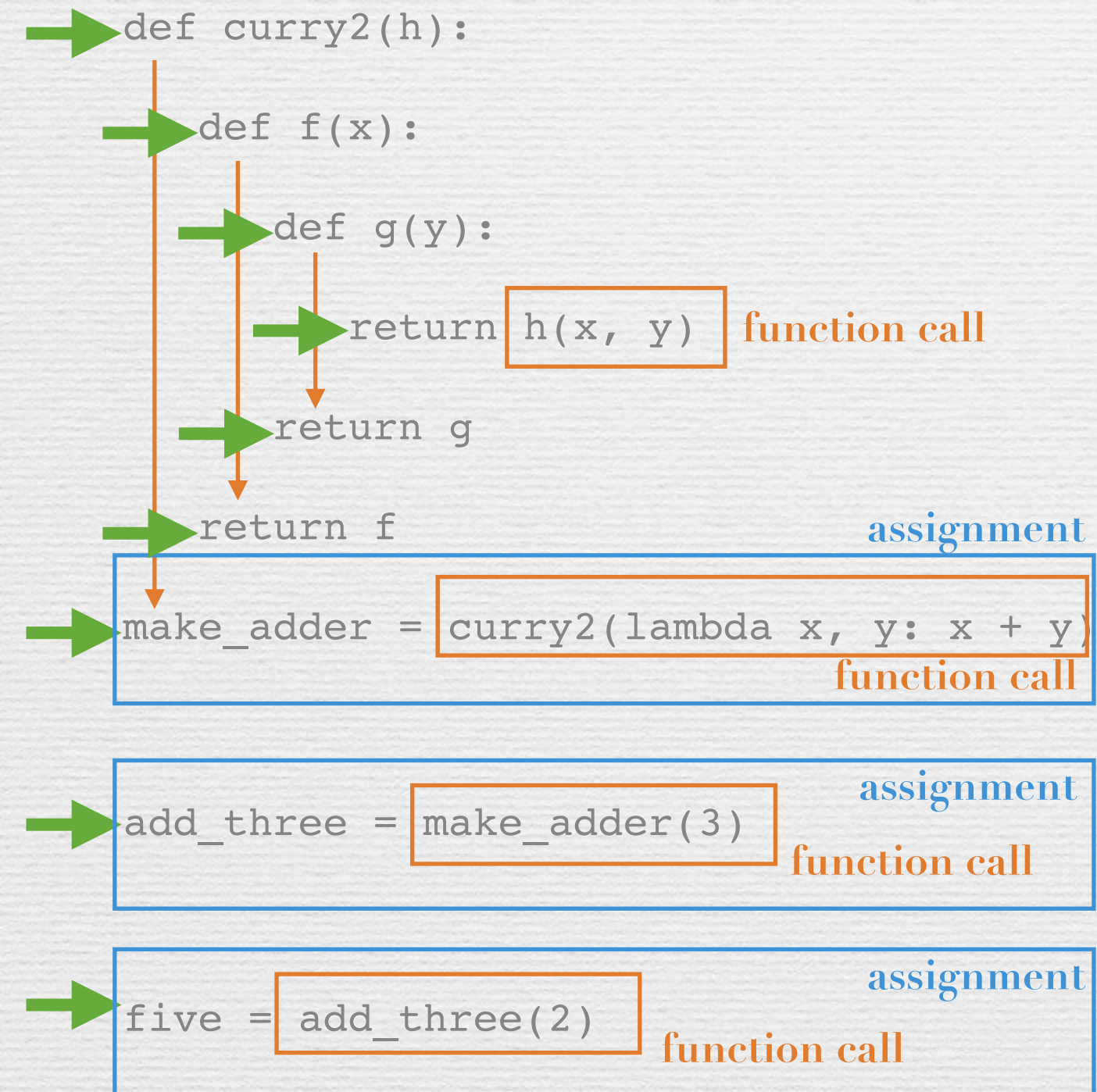
```
→ def curry2(h):  
    → def f(x):  
        → def g(y):  
            → return h(x, y) function call  
        → return g  
    → return f assignment  
→ make_adder = curry2(lambda x, y: x + y) function call  
→ add_three = make_adder(3) assignment  
→ five = add_three(2) function call
```

tip: when you start doing a function call, mark where you were before so that you know which line to go back to





# 1.5 #1



tip: when you start doing a function call, mark where you were before so that you know which line to go back to



# 1.5 #2

```
n = 7

def f(x):
    n = 8
    return x + 1

def g(x):
    n = 9
    def h():
        return x + 1
    return h

def f(f, x):
    return f(x + n)

f = f(g, n)

g = (lambda y: y())(f)
```



# 1.5 #2



```
n = 7
def f(x):
    n = 8
    return x + 1
def g(x):
    n = 9
    def h():
        return x + 1
    return h
def f(f, x):
    return f(x + n)
```

```
f = f(g, n)
```

```
g = (lambda y: y())(f)
```

Global Frame:

n: 7



# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
def g(x):
    n = 9
    def h():
        return x + 1
    return h
def f(f, x):
    return f(x + n)

f = f(g, n)

g = (lambda y: y())(f)
```

Global Frame:

n: 7

f → func f(x) [P = G]



# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
def g(x):
    n = 9
    def h():
        return x + 1
    return h
def f(f, x):
    return f(x + n)

f = f(g, n)

g = (lambda y: y())(f)
```

Global Frame:

n: 7

f → func f(x) [P = G]



# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    n = 9
    def h():
        return x + 1
    return h
def f(f, x):
    return f(x + n)

f = f(g, n)

g = (lambda y: y())(f)
```

Global Frame:

n: 7  
f → func f(x) [P = G]  
g → func g(x) [P = G]



# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    n = 9
    def h():
        return x + 1
    return h
def f(f, x):
    return f(x + n)
```

```
f = f(g, n)
```

```
g = (lambda y: y())(f)
```

Global Frame:

```
n: 7
f → func f(x) [P = G]
g → func g(x) [P = G]
```



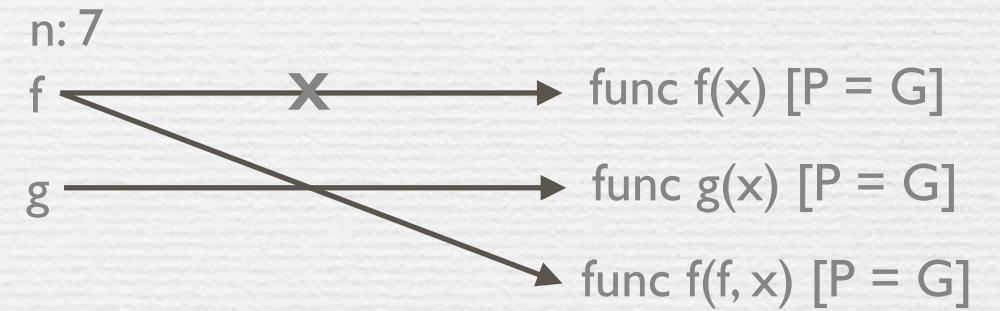
# I.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    n = 9
    def h():
        return x + 1
    return h
→ def f(f, x):
    return f(x + n)
```

```
f = f(g, n)
```

```
g = (lambda y: y())(f)
```

Global Frame:





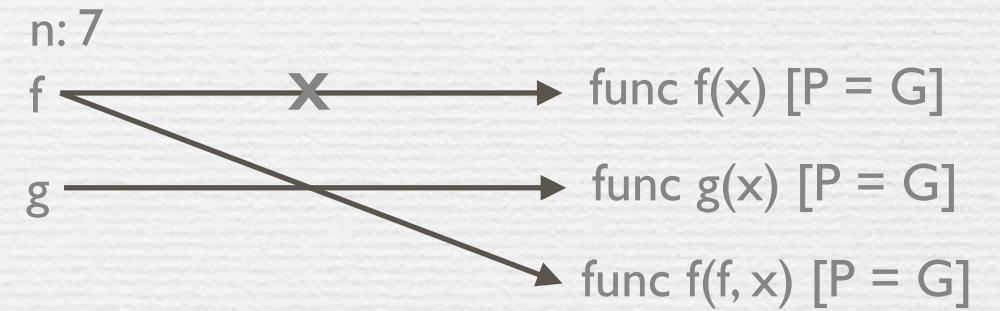
# I.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    n = 9
    def h():
        return x + 1
    return h
→ def f(f, x):
    return f(x + n)
```

```
f = f(g, n)
```

```
g = (lambda y: y())(f)
```

Global Frame:





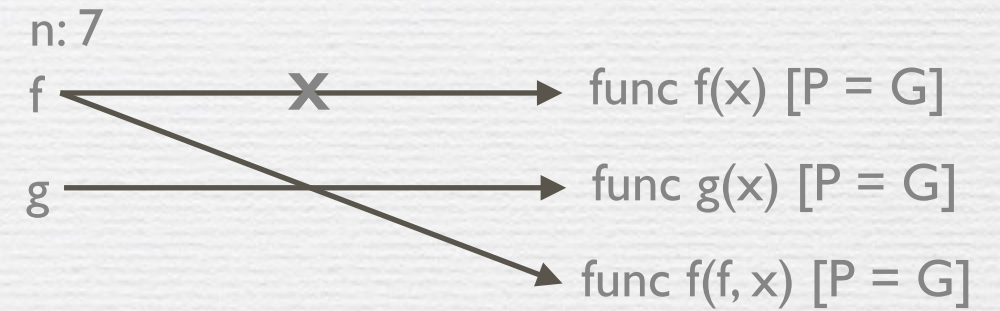
# I.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    n = 9
    def h():
        return x + 1
    return h
→ def f(f, x):
    return f(x + n)

→ f = f(g, n) function call
```

```
g = (lambda y: y())(f)
```

Global Frame:





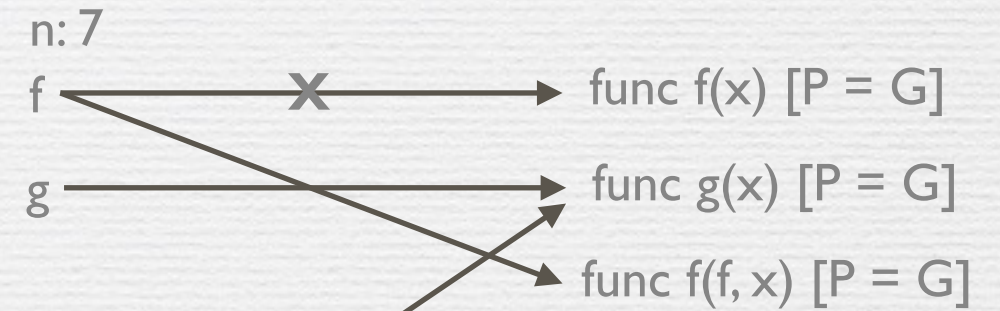
# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    n = 9
    def h():
        return x + 1
    return h
→ def f(f, x):
    return f(x + n)

→ f = f(g, n) function call
```

```
g = (lambda y: y())(f)
```

Global Frame:



f1: f [P = G]  
f  
x: 7



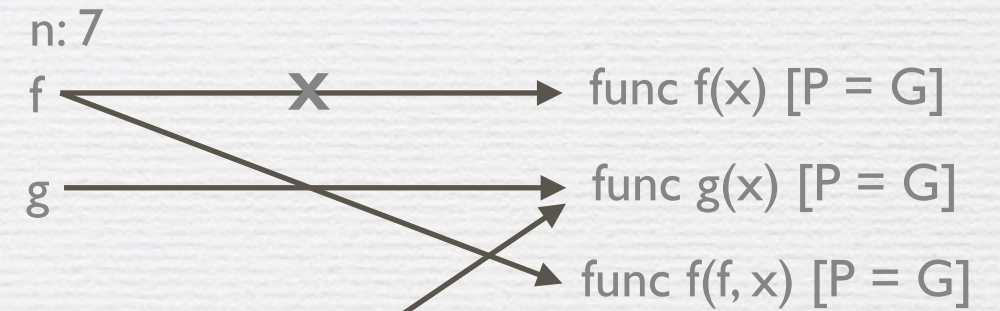
# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    n = 9
    def h():
        return x + 1
    return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call
```

```
g = (lambda y: y())(f)
```

Global Frame:



`f1: f [P = G]`  
`f`  
`x: 7`



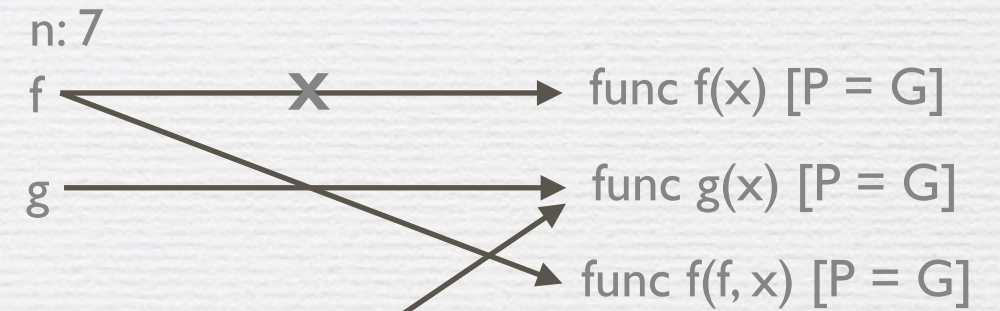
# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    n = 9
    def h():
        return x + 1
    return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call
```

```
g = (lambda y: y())(f)
```

Global Frame:



`f1: f [P = G]`  
`f`  
`x: 7`

`f2: g [P = G]`  
`x: 14`



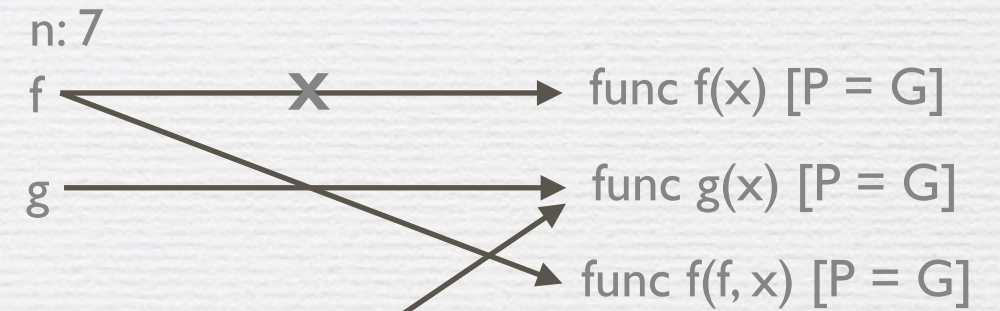
# I.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    def h():
        return x + 1
    return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call
```

```
g = (lambda y: y())(f)
```

Global Frame:



`f1: f [P = G]`  
`f`  
`x: 7`

`f2: g [P = G]`  
`x: 14`  
`n: 9`



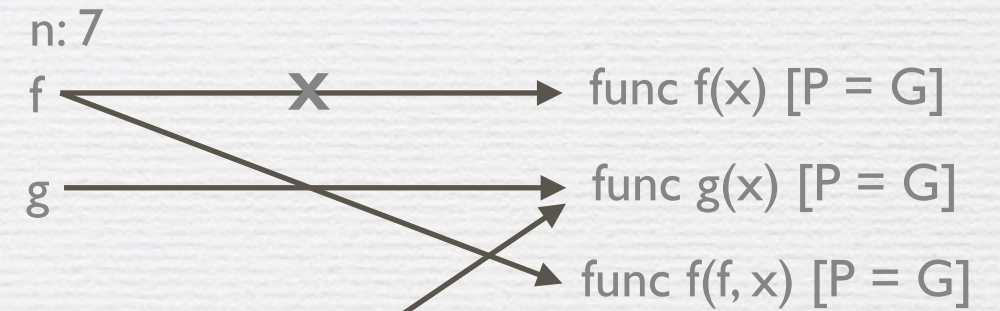
# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        return x + 1
    return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call
```

```
g = (lambda y: y())(f)
```

Global Frame:



`f1: f [P = G]`  
`f`  
`x: 7`

`f2: g [P = G]`  
`x: 14`  
`n: 9`  
`h` points to `func h() [P = f2]`



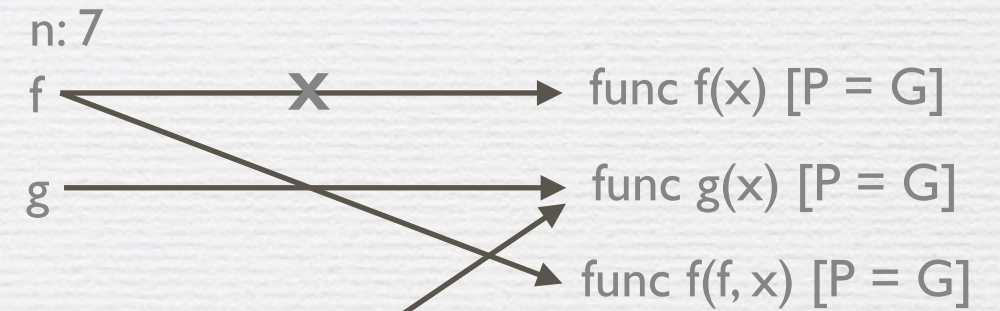
# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
        return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call
```

```
g = (lambda y: y())(f)
```

Global Frame:



f1: f [P = G]  
f  
x: 7

f2: g [P = G]  
x: 14  
n: 9  
h → func h() [P = f2]



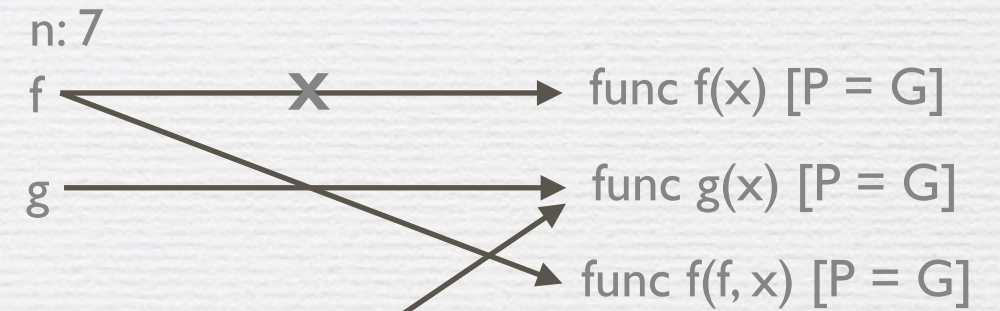
# I.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call
```

```
g = (lambda y: y())(f)
```

Global Frame:

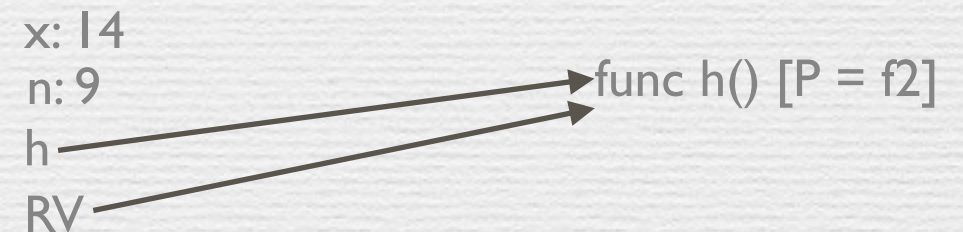


f1: f [P = G]

f1 frame diagram showing local bindings:

- `f` points to `func f(f, x) [P = G]` (from the Global Frame)
- `x: 7`

f2: g [P = G]





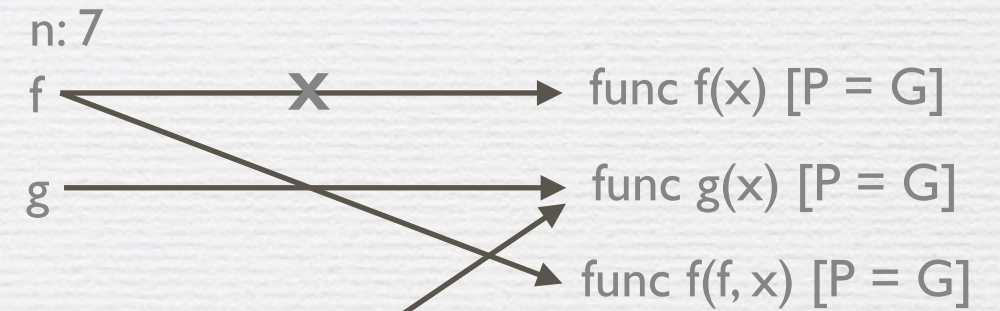
# I.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call
```

```
g = (lambda y: y())(f)
```

Global Frame:



f1: f [P = G]

f  
x: 7  
RV

f2: g [P = G]

x: 14  
n: 9  
h  
RV

func h() [P = f2]



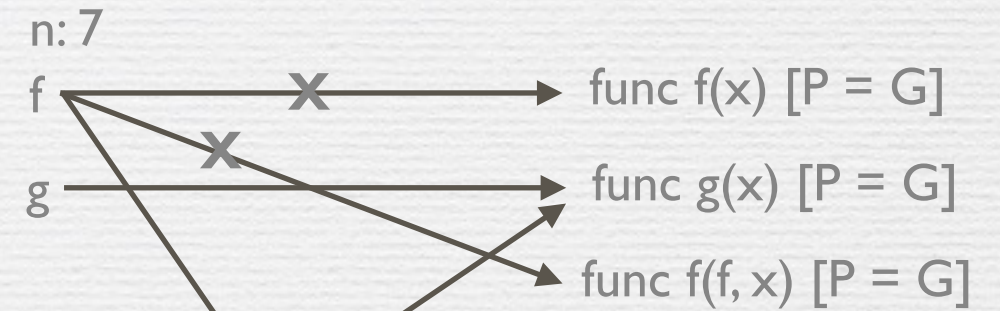
# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call assignment
```

```
g = (lambda y: y())(f)
```

Global Frame:



f1: f [P = G]

f  
x: 7  
RV

f2: g [P = G]

x: 14  
n: 9  
h  
RV  
func h() [P = f2]



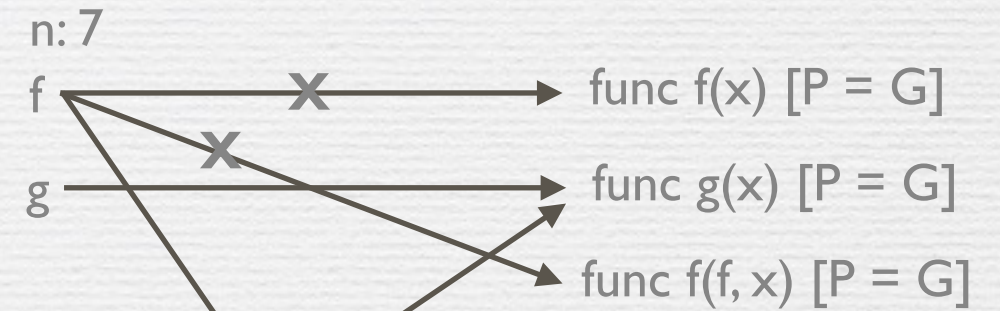
# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call assignment
```

```
→ g = (lambda y: y())(f) function call
```

Global Frame:



f1: f [P = G]

f  
x: 7  
RV

f2: g [P = G]

x: 14  
n: 9  
h  
RV  
func h() [P = f2]

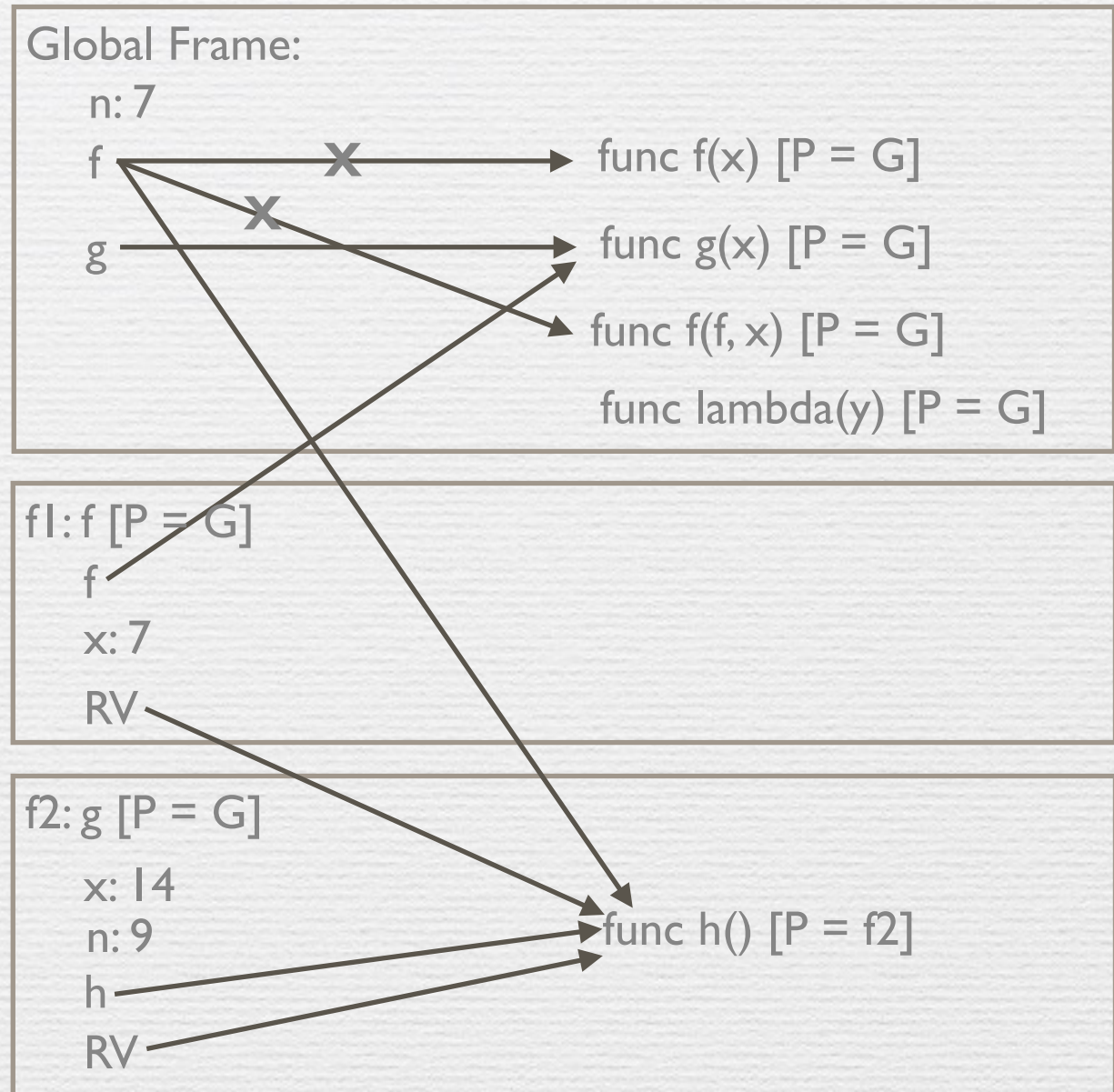


# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call assignment
```

```
→ g = (lambda y: y())(f) function call
```



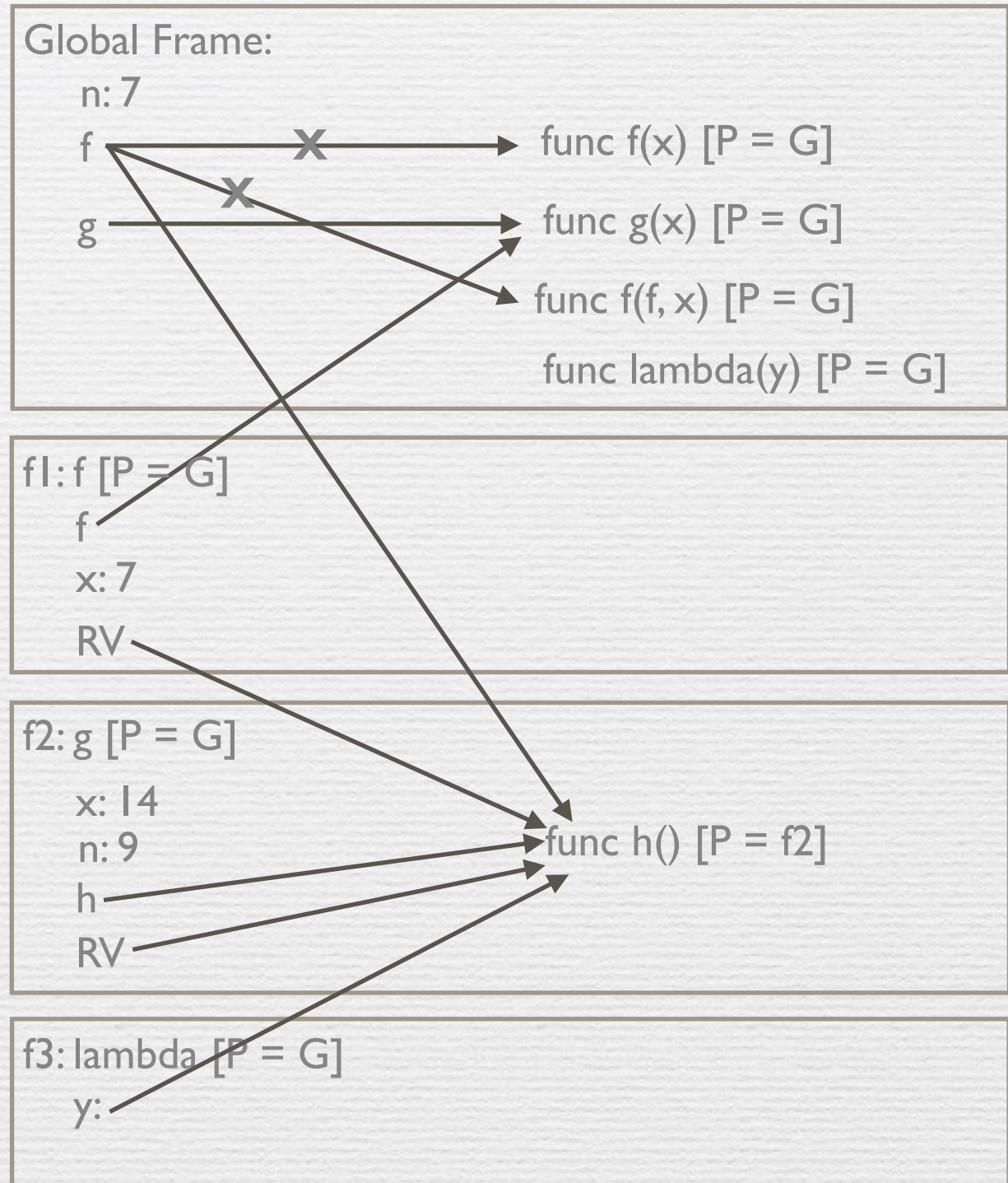


# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call assignment
```

```
→ g = (lambda y: y())(f) function call
```



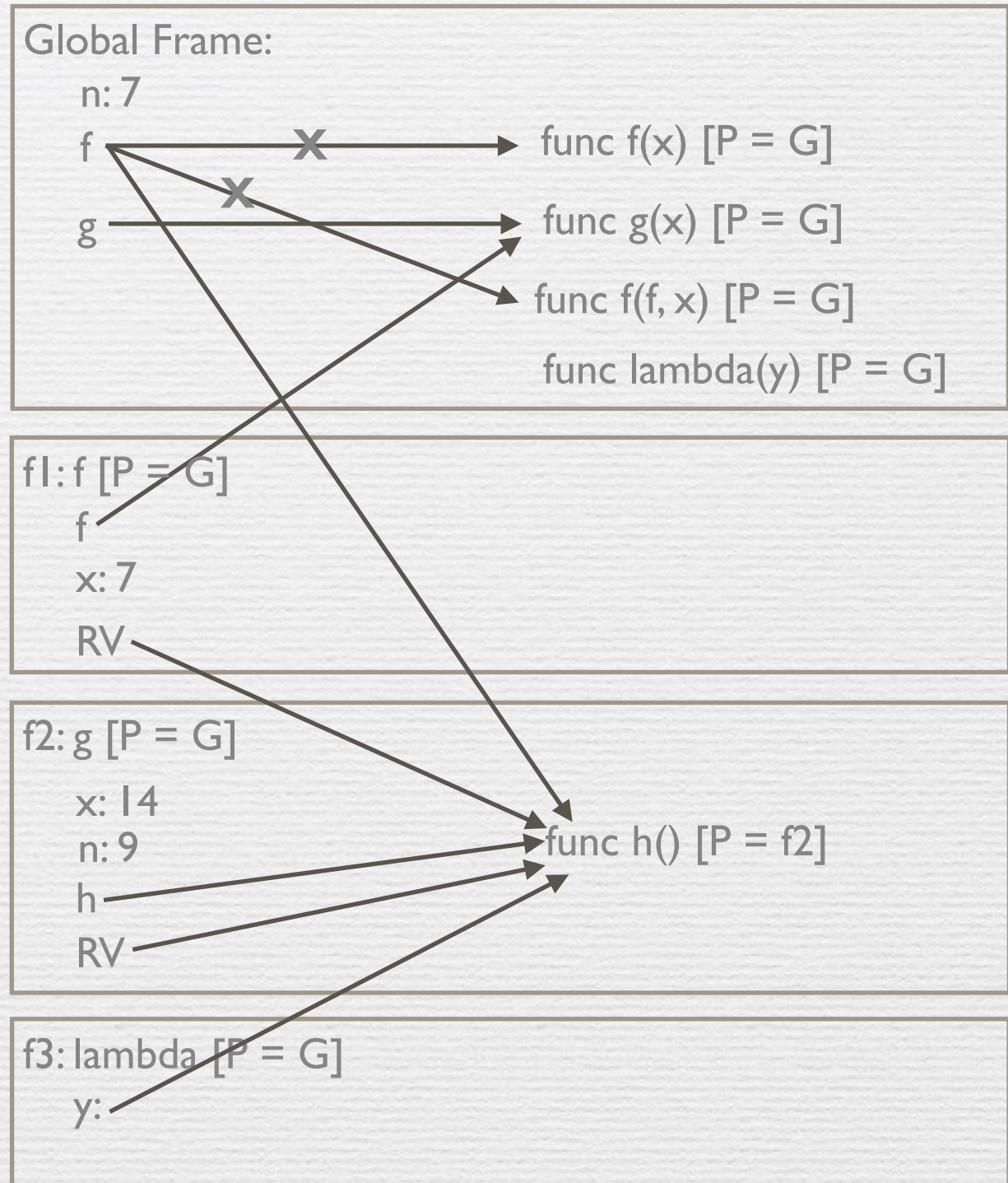


# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call assignment
```

```
→ g = (lambda y: y())(f) function call
function call
```



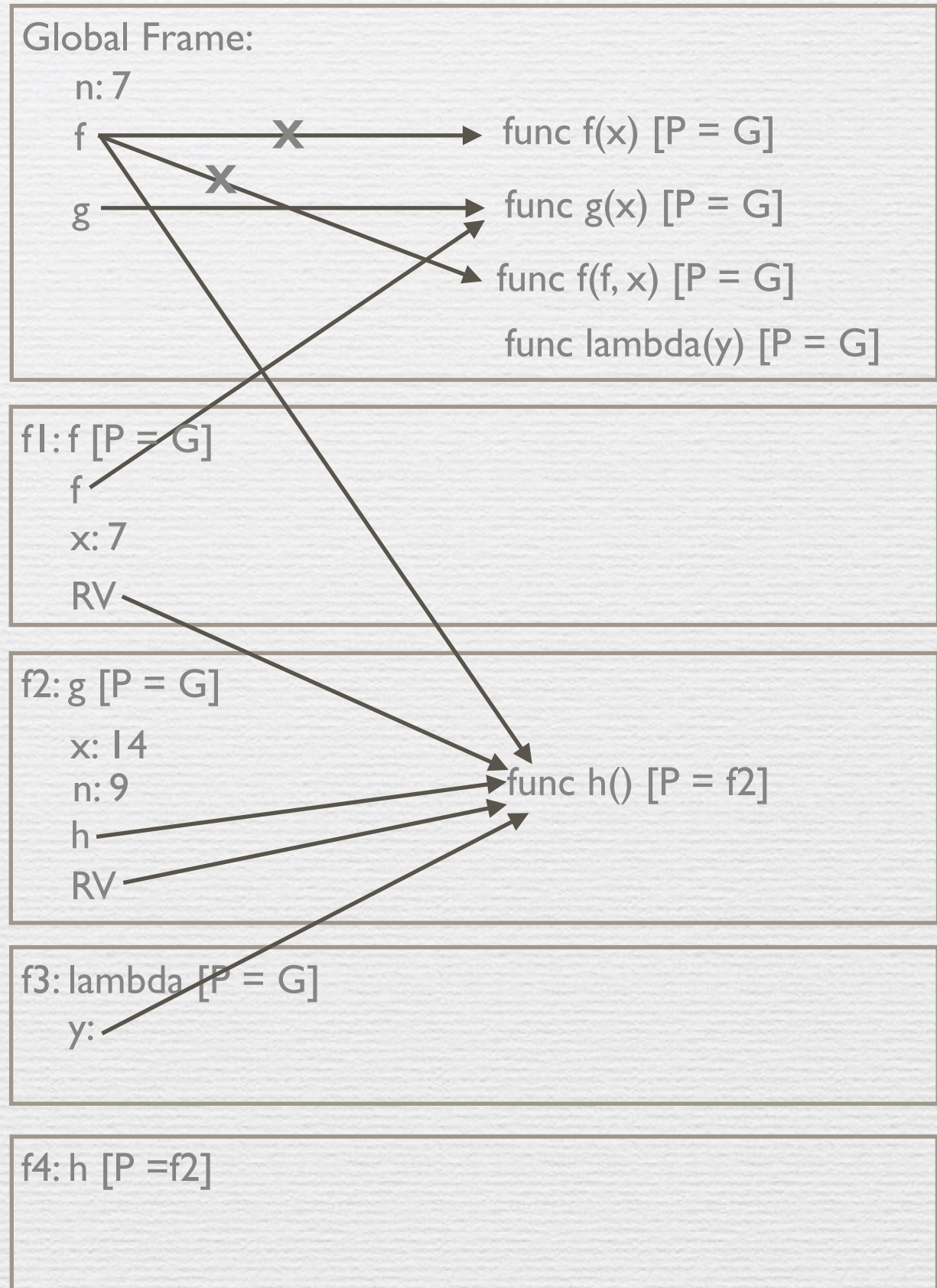


# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call assignment
```

```
→ g = (lambda y: y())(f) function call
function call
```



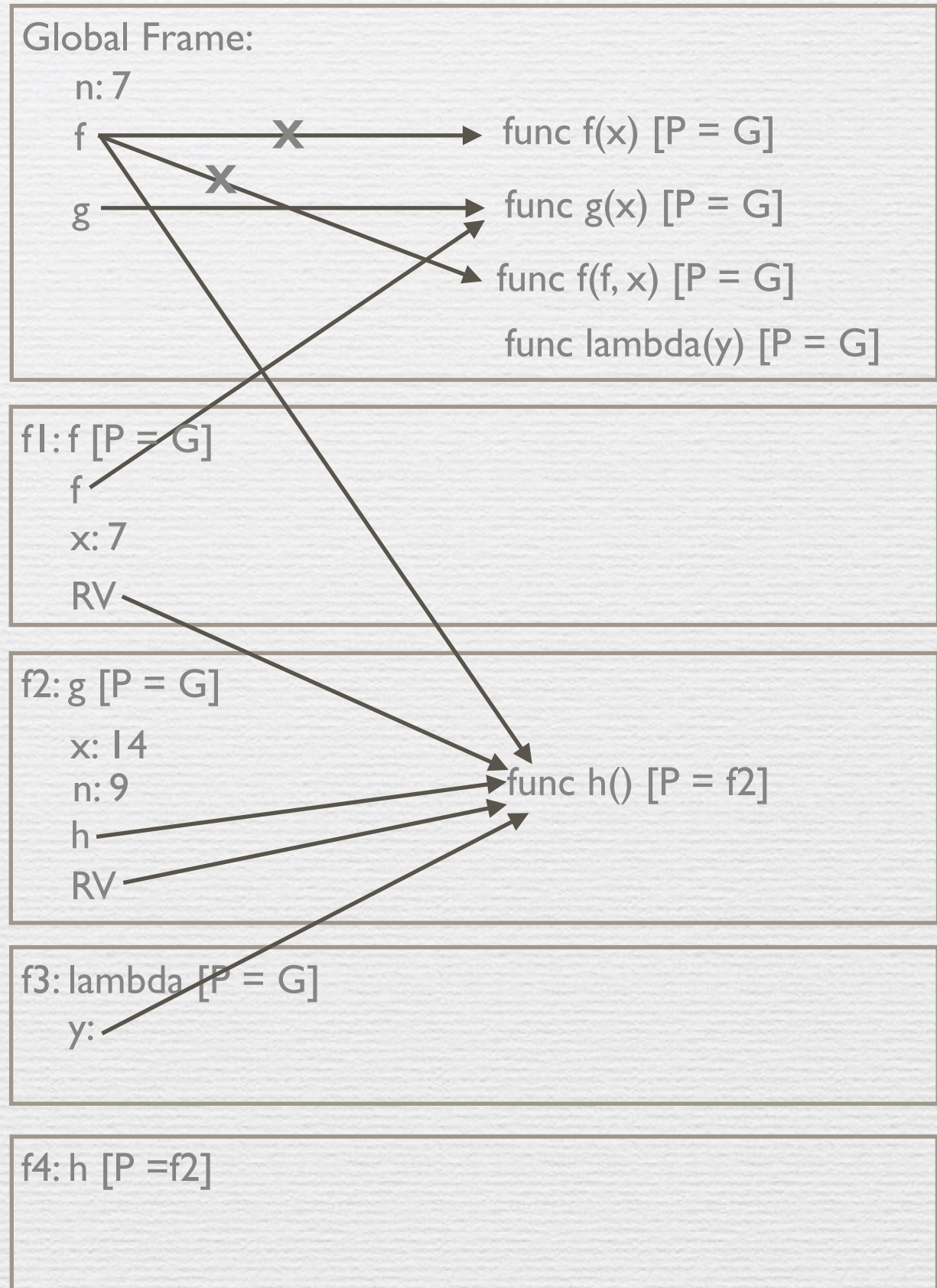


# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call assignment
```

```
→ g = (lambda y: y())(f) function call
function call
```



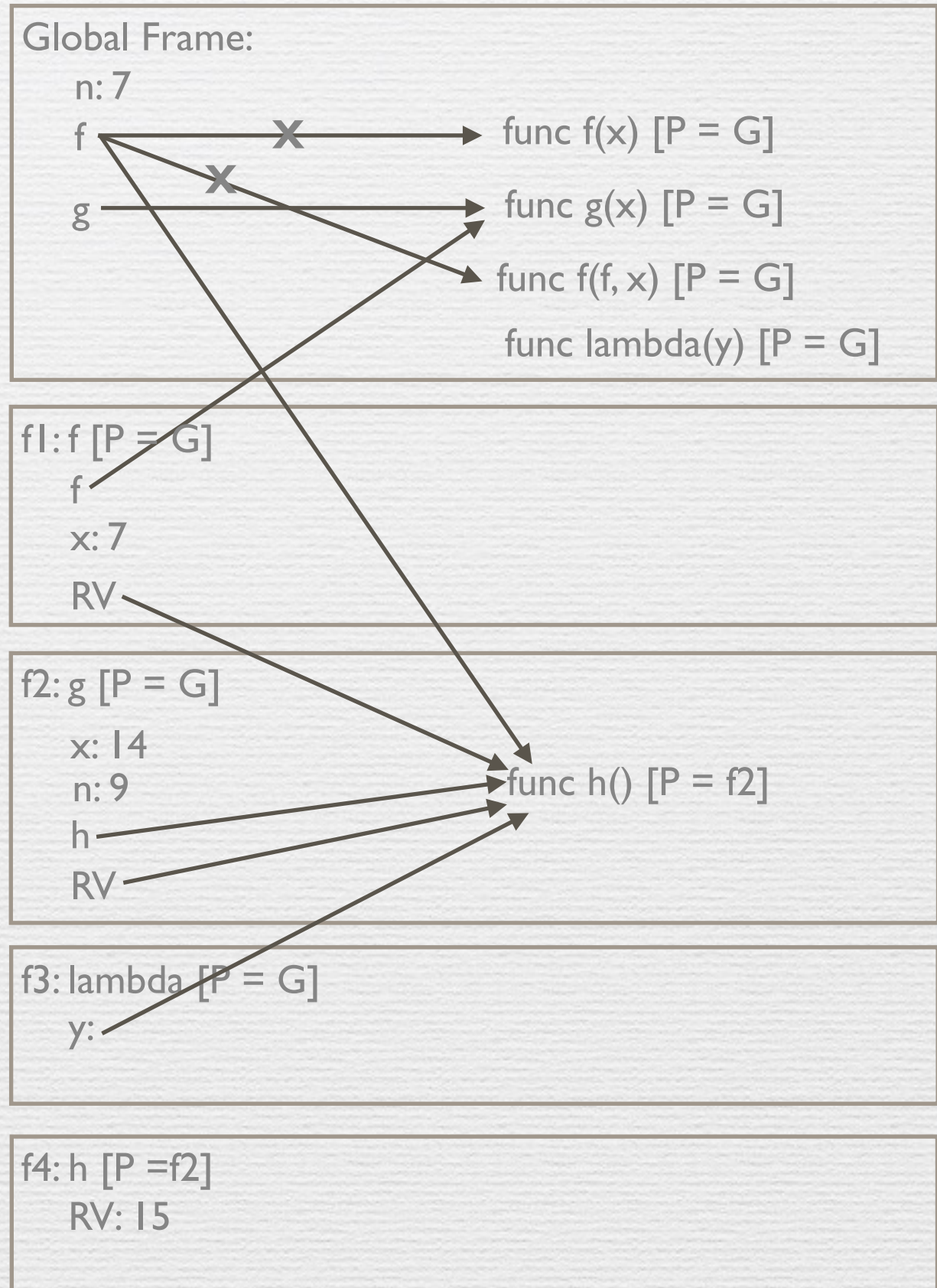


# I.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call assignment
```

```
→ g = (lambda y: y())(f) function call
function call
```





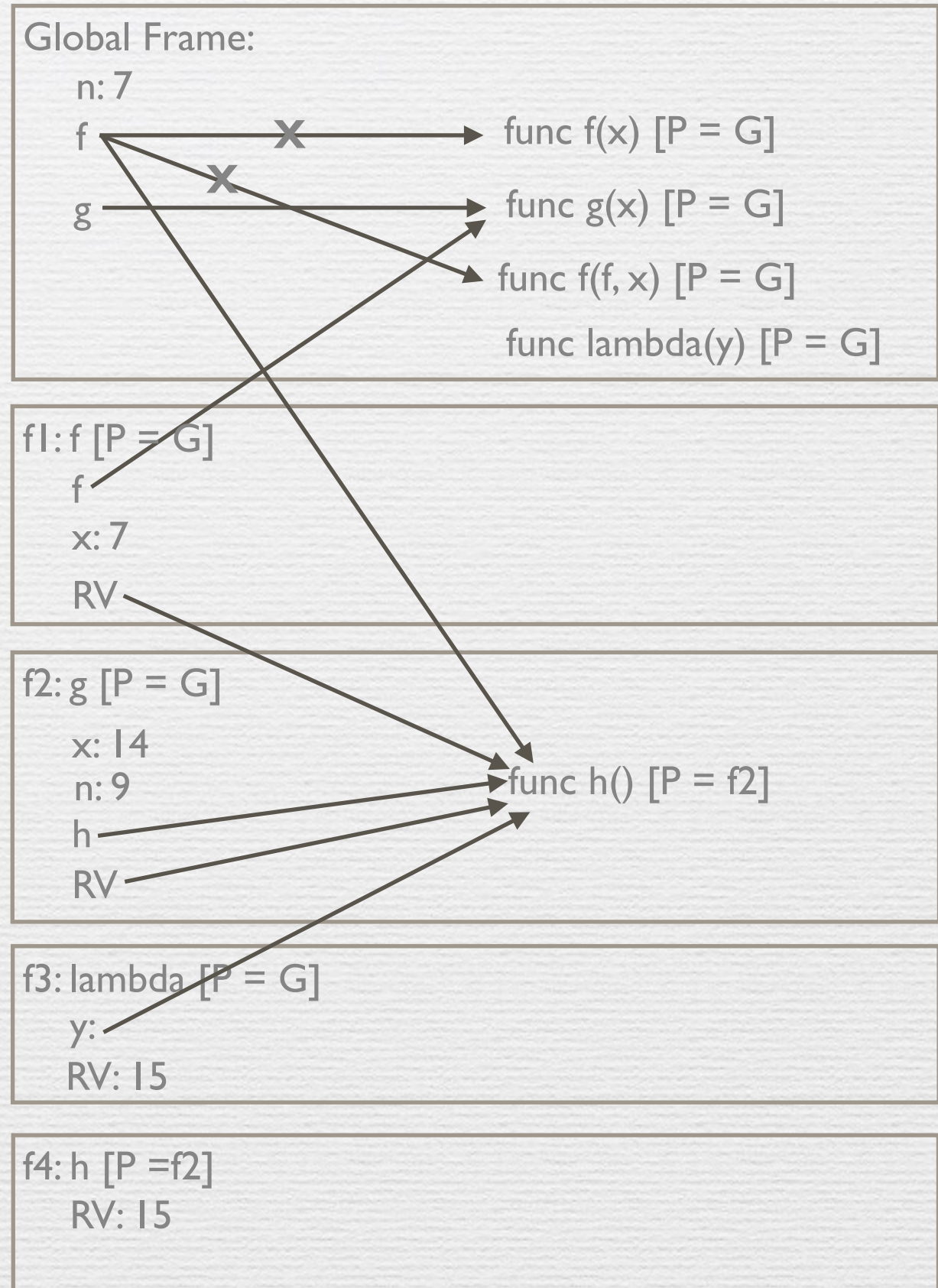
# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

→ `f = f(g, n)` **function call** **assignment**

→ g = (lambda y: y())(f)      function call

function call





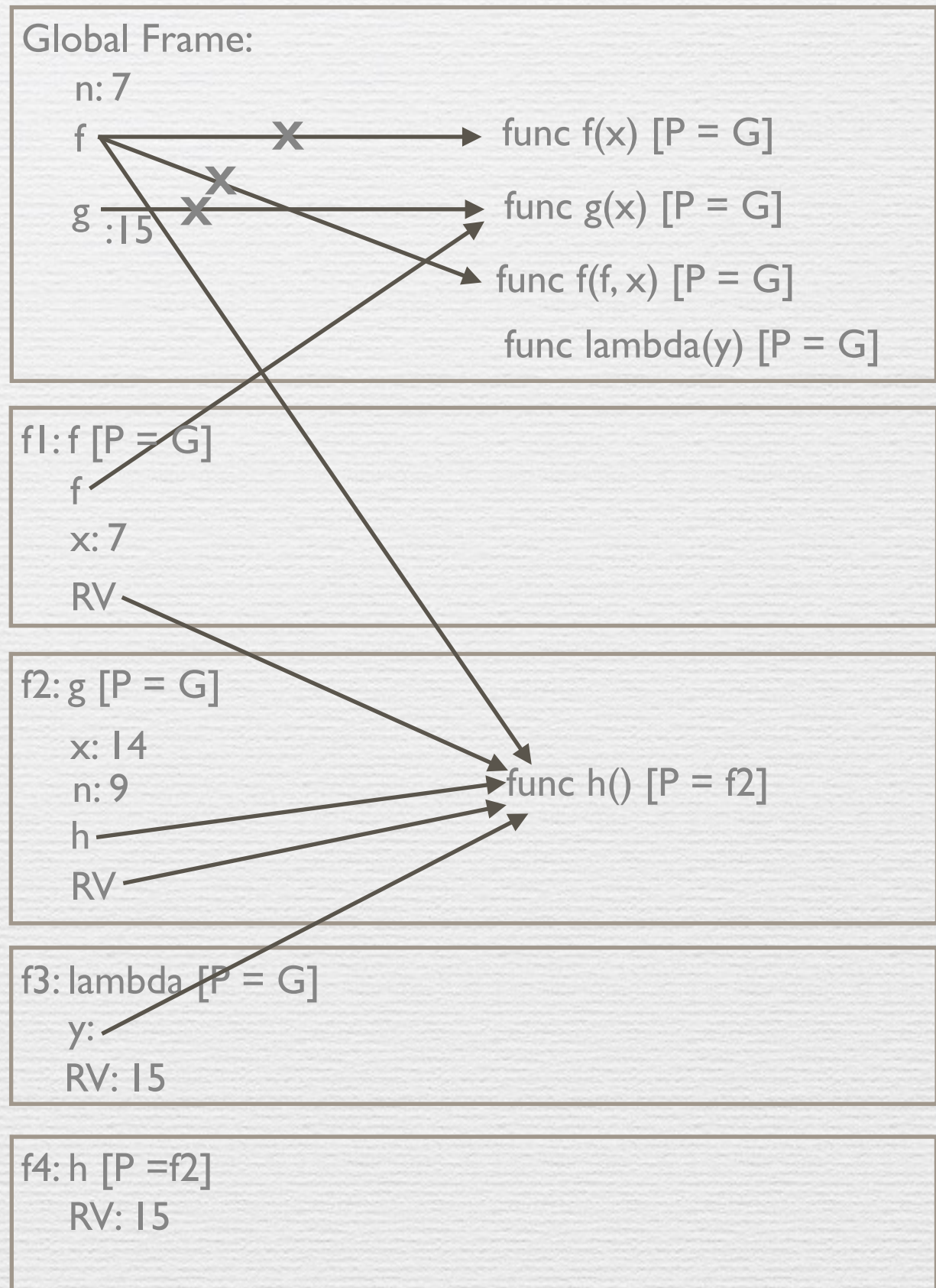
# 1.5 #2

```
→ n = 7
→ def f(x):
    n = 8
    return x + 1
→ def g(x):
    → n = 9
    → def h():
        → return x + 1
    → return h
→ def f(f, x):
    → return f(x + n) function call
```

```
→ f = f(g, n) function call assignment
```

```
→ g = (lambda y: y())(f) function call assignment
```

**function call**





# Challenge Problem



# I.5 #3

```
y = "y"
```

```
h = y
```

```
def y(y):
```

```
    h = "h"
```

```
    if y == h:
```

```
        return y + "i"
```

```
    y = lambda y: y(h)
```

```
    return lambda h: y(h)
```

```
y = y(y)(y)
```



# 1.5 #3

→ `y = "y"`

`h = y`

`def y(y):`

`h = "h"`

`if y == h:`

`return y + "i"`

`y = lambda y: y(h)`

`return lambda h: y(h)`

`y = y(y)(y)`

Global Frame:

`y: "y"`



# I.5 #3

→ `y = "y"`

→ `h = y`

Global Frame:

y: "y"

h: "y"

```
def y(y):
```

```
    h = "h"
```

```
    if y == h:
```

```
        return y + "i"
```

```
    y = lambda y: y(h)
```

```
    return lambda h: y(h)
```

```
y = y(y)(y)
```



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

`h = "h"`

`if y == h:`

`return y + "i"`

`y = lambda y: y(h)`

`return lambda h: y(h)`

`y = y(y)(y)`

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

`h = "h"`

`if y == h:`

`return y + "i"`

`y = lambda y: y(h)`

`return lambda h: y(h)`

`y = y(y)(y)`

Global Frame:

`y: "y"`

`h: "y"`

`func y(y) [P = G]`



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

`h = "h"`

`if y == h:`

`return y + "i"`

`y = lambda y: y(h)`

`return lambda h: y(h)`

→ `y = y(y)(y)`

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

`h = "h"`

`if y == h:`

`return y + "i"`

`y = lambda y: y(h)`

`return lambda h: y(h)`

→ `y = y(y)(y)`

Global Frame:

`y: "y"`

`h: "y"`

`func y(y) [P = G]`

`fl: y [P = G]`

`y:`



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

`if y == h:`

`return y + "i"`

`y = lambda y: y(h)`

`return lambda h: y(h)`

→ `y = y(y)(y)`

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

fl: y [P = G]

y:

h: "h"



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

func y

"h"

`return y + "i"`

`y = lambda y: y(h)`

`return lambda h: y(h)`

→ `y = y(y)(y)`

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

fl: y [P = G]

y:

h: "h"



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

func y

"h"

`return y + "i"`

→ `y = lambda y: y(h)` assignment

`return lambda h: y(h)`

→ `y = y(y)(y)`

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

fl: y [P = G]

y:

h: "h"



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

func y

"h"

`return y + "i"`

→ `y = lambda y: y(h)` **assignment**

`return lambda h: y(h)`

→ `y = y(y)(y)`

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

fl: y [P = G]

y: ~~"y"~~

h: "h"

func lambda l(y) [P = fl]



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

func y

"h"

`return y + "i"`

→ `y = lambda y: y(h)` assignment

→ `return lambda h: y(h)`

→ `y = y(y)(y)`

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

fl: y [P = G]

y: ~~"y"~~

h: "h"

RV

func lambda1(y) [P = fl]

func lambda2(h) [P = fl]



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

func y

"h"

`return y + "i"`

→ `y = lambda y: y(h)` assignment

→ `return lambda h: y(h)`

→ `y = y(y)(y)`

We just completed the first `y(y)` function call: now we know what the operator is for the second function call

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

fl: y [P = G]

y: ~~"y"~~

h: "h"

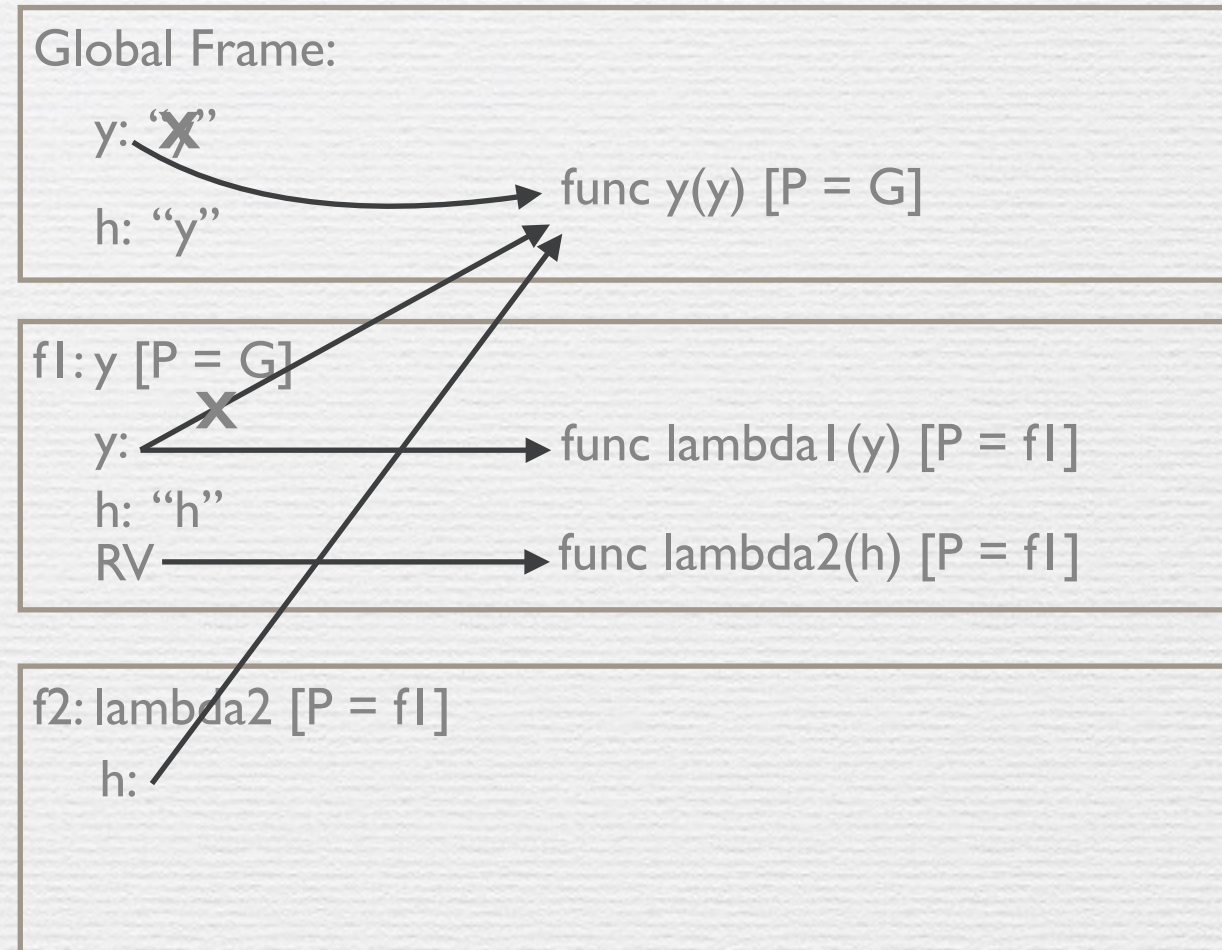
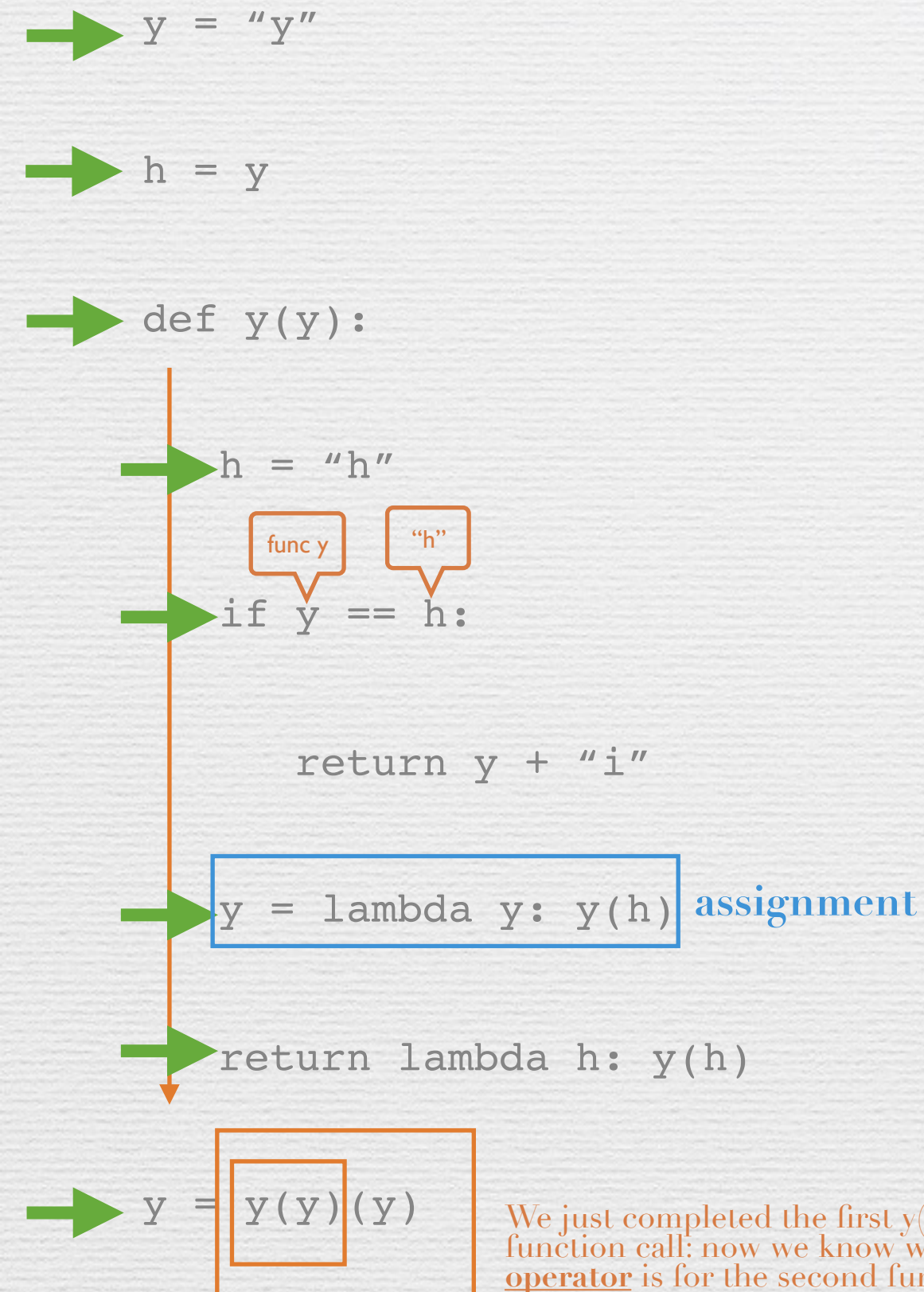
RV

func lambda1(y) [P = fl]

func lambda2(h) [P = fl]



# I.5 #3





# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

func y

"h"

`return y + "i"`

→ `y = lambda y: y(h)` **assignment**

→ `return lambda h: y(h)`

y is not defined here,  
we must look for it in  
the **parent** frames  
h was passed in; in  
this case h is the  
function y (check f2)

→ `y = y(y)(y)`

We just completed the first `y(y)`  
function call: now we know what the  
**operator** is for the second function call

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

f1: y [P = G]

y: ~~"y"~~

h: "h"

RV

func lambda1(y) [P = f1]

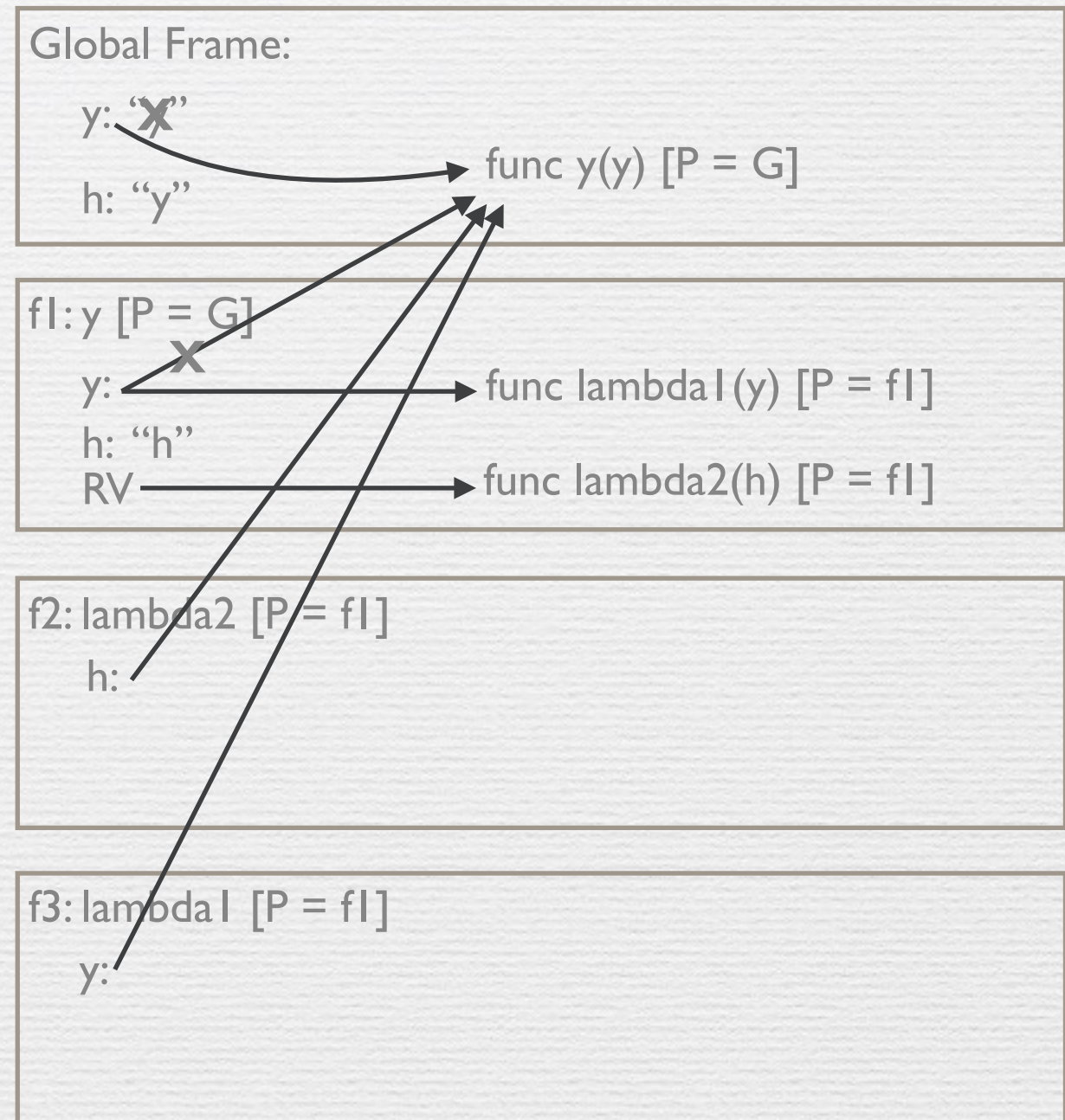
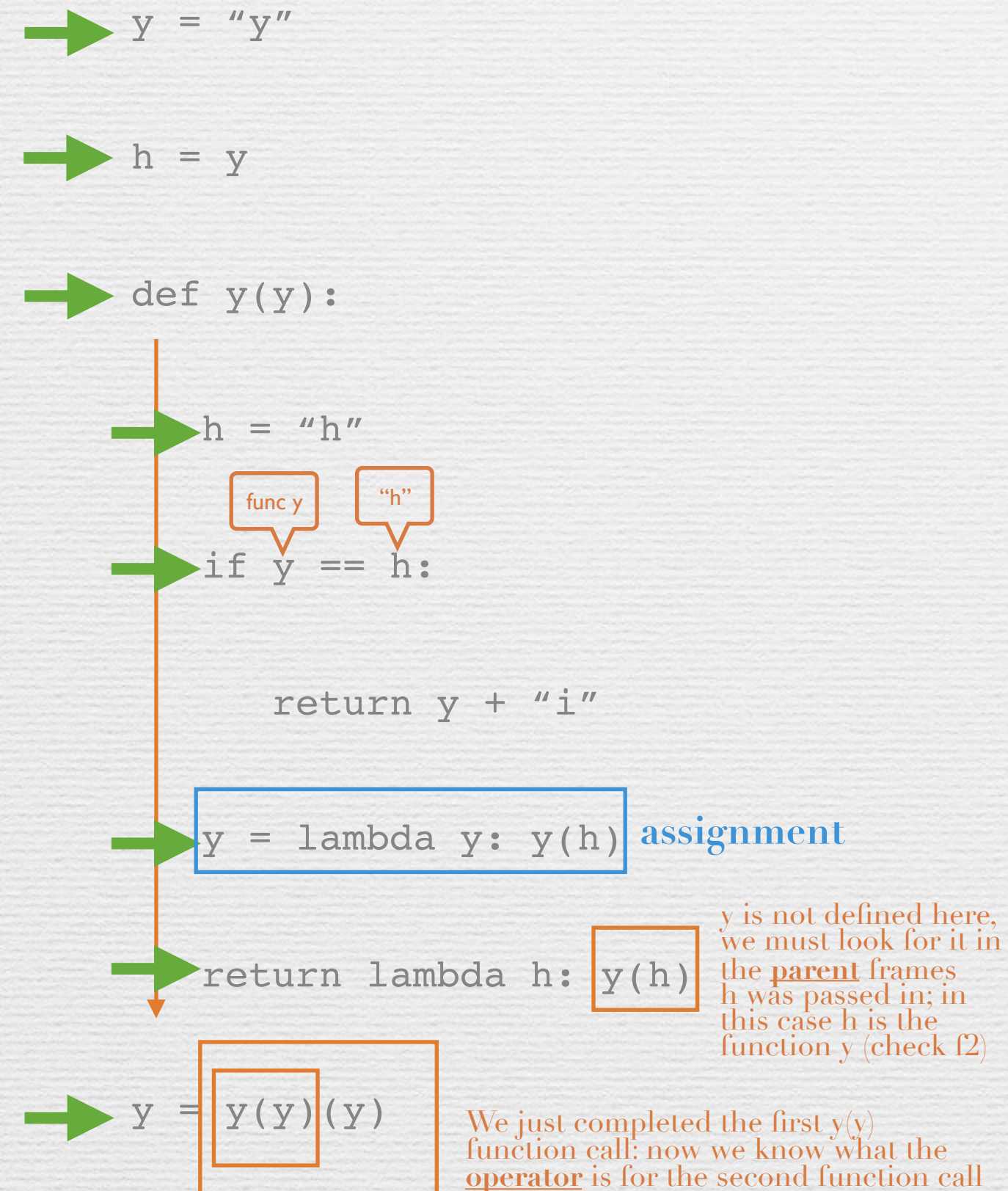
func lambda2(h) [P = f1]

f2: lambda2 [P = f1]

h:



# I.5 #3





# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

`return y + "i"`

→ `y = lambda y: y(h)` **assignment**

→ `return lambda h: y(h)`

→ `y = y(y)(y)`

The return value of lambda1 is the result of calling y (which is what we passed in) on h. Since h is not defined in this frame, we must look at the parent frames

**assignment**

y is not defined here, we must look for it in the parent frames h was passed in; in this case h is the function y (check f2)

We just completed the first y(y) function call: now we know what the operator is for the second function call

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

f1: y [P = G]

y: ~~"y"~~

func lambda1(y) [P = f1]

h: "h"

func lambda2(h) [P = f1]

RV

f2: lambda2 [P = f1]

h:

f3: lambda1 [P = f1]

y:



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

`return y + "i"`

→ `y = lambda y: y(h)`

→ `return lambda h: y(h)`

→ `y = y(y)(y)`

The return value of lambda1 is the result of calling y (which is what we passed in) on h. Since h is not defined in this frame, we must look at the parent frames

**assignment**

y is not defined here, we must look for it in the parent frames h was passed in; in this case h is the function y (check f2)

We just completed the first y(y) function call: now we know what the operator is for the second function call

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

f1: y [P = G]

y: ~~"y"~~

h: "h"

RV

func lambda1(y) [P = f1]

func lambda2(h) [P = f1]

f2: lambda2 [P = f1]

h:

f3: lambda1 [P = f1]

y:

f3: y [P = G]

y: "h"



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

`return y + "i"`

→ `y = lambda y: y(h)`

→ `return lambda h: y(h)`

→ `y = y(y)(y)`

The return value of lambda1 is the result of calling y (which is what we passed in) on h. Since h is not defined in this frame, we must look at the parent frames

**assignment**

y is not defined here, we must look for it in the parent frames h was passed in; in this case h is the function y (check f2)

We just completed the first y(y) function call: now we know what the operator is for the second function call

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

f1: y [P = G]

y: ~~"y"~~

h: "h"

RV

func lambda1(y) [P = f1]

func lambda2(h) [P = f1]

f2: lambda2 [P = f1]

h:

f3: lambda1 [P = f1]

y:

f3: y [P = G]

y: "h"

h: "h"



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

`func y` `"h"`

`"h"`

`return y + "i"`

→ `y = lambda y: y(h)` **assignment**

→ `return lambda h: y(h)`

The return value of `lambda1` is the result of calling `y` (which is what we passed in) on `h`. Since `h` is not defined in this frame, we must look at the parent frames

→ `y = y(y)(y)`

We just completed the first `y(y)` function call: now we know what the operator is for the second function call

Global Frame:

`y: "X"`

`h: "y"`

`func y(y) [P = G]`

`f1: y [P = G]`

`y: "X"`

`h: "h"`

`RV`

`func lambda1(y) [P = f1]`

`func lambda2(h) [P = f1]`

`f2: lambda2 [P = f1]`

`h:`

`f3: lambda1 [P = f1]`

`y:`

`f3: y [P = G]`

`y: "h"`

`h: "h"`



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

→ `return y + "i"`

→ `y = lambda y: y(h)` **assignment**

→ `return lambda h: y(h)`

→ `y = y(y)(y)`

The return value of lambda1 is the result of calling y (which is what we passed in) on h. Since h is not defined in this frame, we must look at the parent frames

**assignment**

y is not defined here, we must look for it in the parent frames h was passed in; in this case h is the function y (check f2)

We just completed the first y(y) function call: now we know what the operator is for the second function call

Global Frame:

y: ~~"y"~~

h: "y"

func y(y) [P = G]

f1: y [P = G]

y: ~~"y"~~

h: "h"

RV

func lambda1(y) [P = f1]

func lambda2(h) [P = f1]

f2: lambda2 [P = f1]

h:

f3: lambda1 [P = f1]

y:

f3: y [P = G]

y: "h"

h: "h"

RV: "hi"



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

→ `return y + "i"`

→ `y = lambda y: y(h)` **assignment**

→ `return lambda h: y(h)`

→ `y = y(y)(y)`

The return value of lambda1 is the result of calling y (which is what we passed in) on h. Since h is not defined in this frame, we must look at the parent frames

**assignment**

y is not defined here, we must look for it in the parent frames h was passed in; in this case h is the function y (check f2)

We just completed the first y(y) function call: now we know what the operator is for the second function call

Global Frame:

y: ~~"y"~~  
h: "y" → func y(y) [P = G]

f1: y [P = G]

y: ~~"y"~~ → func lambda1(y) [P = f1]  
h: "h"  
RV → func lambda2(h) [P = f1]

f2: lambda2 [P = f1]

h:

f3: lambda1 [P = f1]

y:  
RV: "hi"

f3: y [P = G]

y: "h"  
h: "h"  
RV: "hi"



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

→ `return y + "i"`

→ `y = lambda y: y(h)` **assignment**

→ `return lambda h: y(h)`

→ `y = y(y)(y)`

The return value of lambda1 is the result of calling y (which is what we passed in) on h. Since h is not defined in this frame, we must look at the parent frames

**assignment**

y is not defined here, we must look for it in the parent frames h was passed in; in this case h is the function y (check f2)

We just completed the first y(y) function call: now we know what the operator is for the second function call

Global Frame:

y: ~~"y"~~  
h: "y" → func y(y) [P = G]

f1: y [P = G]

y: ~~"y"~~ → func lambda1(y) [P = f1]  
h: "h"  
RV → func lambda2(h) [P = f1]

f2: lambda2 [P = f1]

h:  
RV: "hi"

f3: lambda1 [P = f1]

y:  
RV: "hi"

f3: y [P = G]

y: "h"  
h: "h"  
RV: "hi"



# I.5 #3

→ `y = "y"`

→ `h = y`

→ `def y(y):`

→ `h = "h"`

→ `if y == h:`

→ `return y + "i"`

→ `y = lambda y: y(h)` **assignment**

→ `return lambda h: y(h)`

→ `y = y(y)(y)`

The return value of lambda1 is the result of calling y (which is what we passed in) on h. Since h is not defined in this frame, we must look at the parent frames

**assignment**

y is not defined here, we must look for it in the parent frames h was passed in; in this case h is the function y (check f2)

We just completed the first y(y) function call: now we know what the operator is for the second function call

Global Frame:

y: ~~"y"~~ "hi"  
h: "y" → func y(y) [P = G]

f1: y [P = G]

y: ~~"y"~~ → func lambda1(y) [P = f1]  
h: "h"  
RV → func lambda2(h) [P = f1]

f2: lambda2 [P = f1]

h:  
RV: "hi"

f3: lambda1 [P = f1]

y:  
RV: "hi"

f3: y [P = G]

y: "h"  
h: "h"  
RV: "hi"