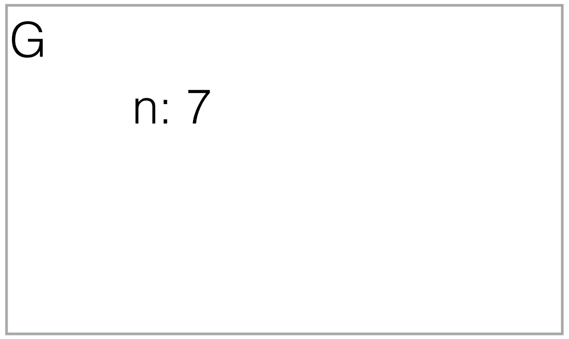
Discussion 02

Environment Diagrams and Recursion

```python
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)
```

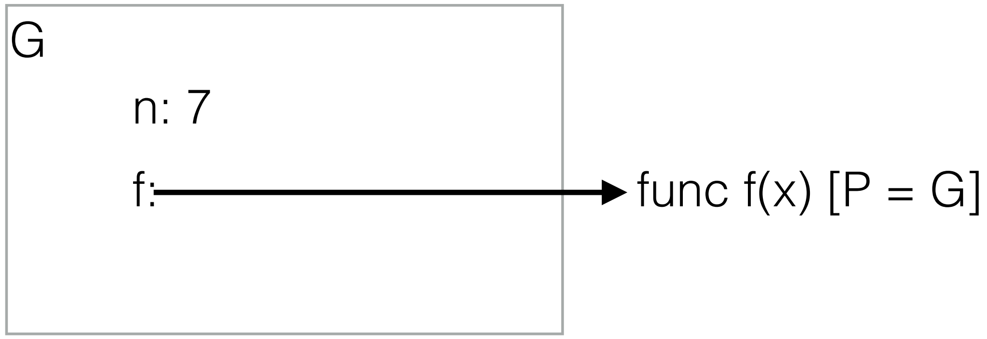
```
n = 7
def f(x):
   n = 8
   return x + 1
def g(x):
   n = 9
   def h():
      return x + 1
   return h
def f(f, x):
   return f(x + n)
f = f(g, n)
g = (lambda y: y())(f)
```



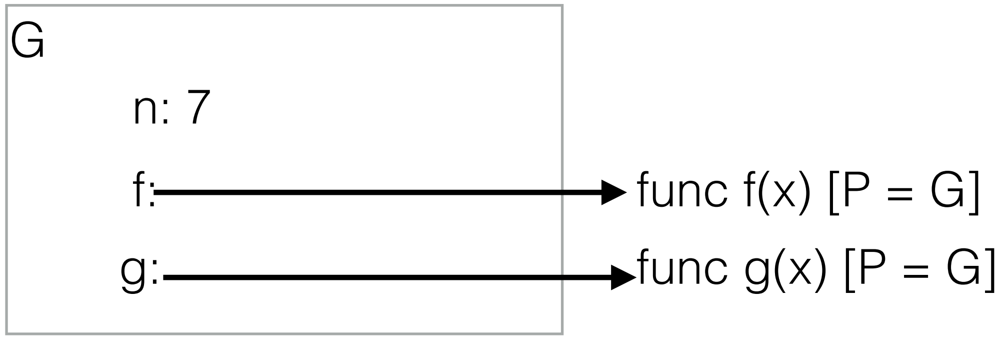
G

n: 7

```
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)
```



```
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)
```



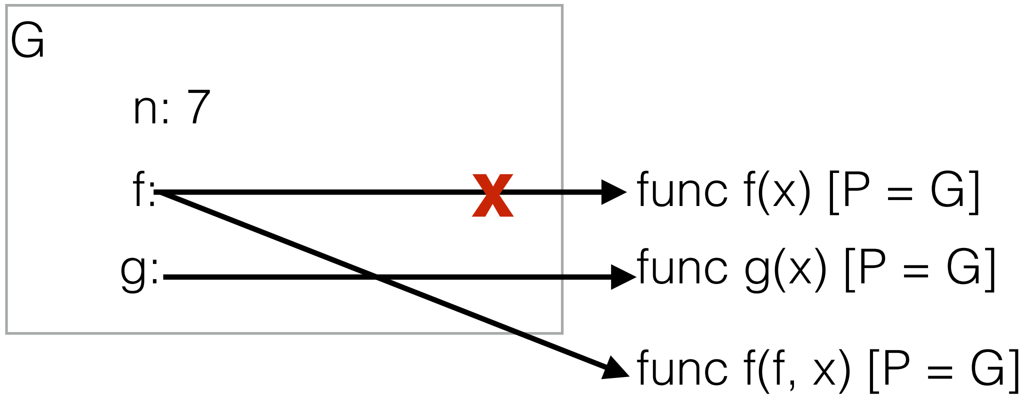
G

n: 7

f: → func f(x) [P = G]

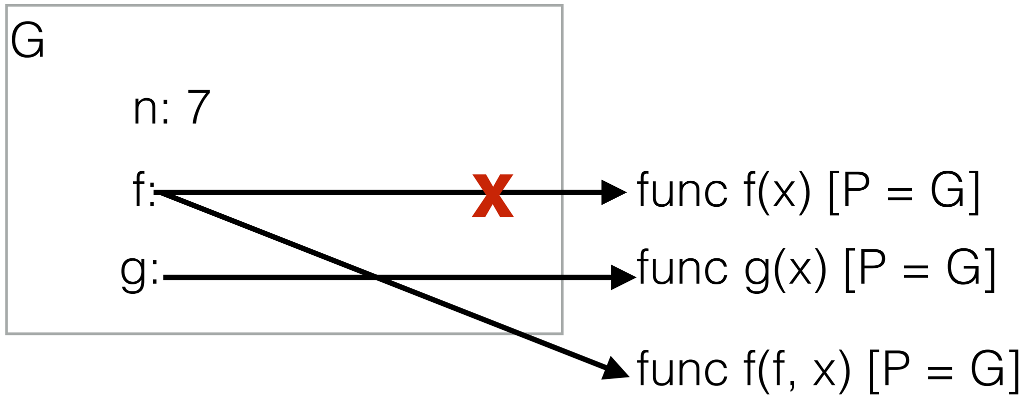
g: → func g(x) [P = G]

```
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)
```



```
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)
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame



G

n: 7

f:

g:

func f(x) [P = G]

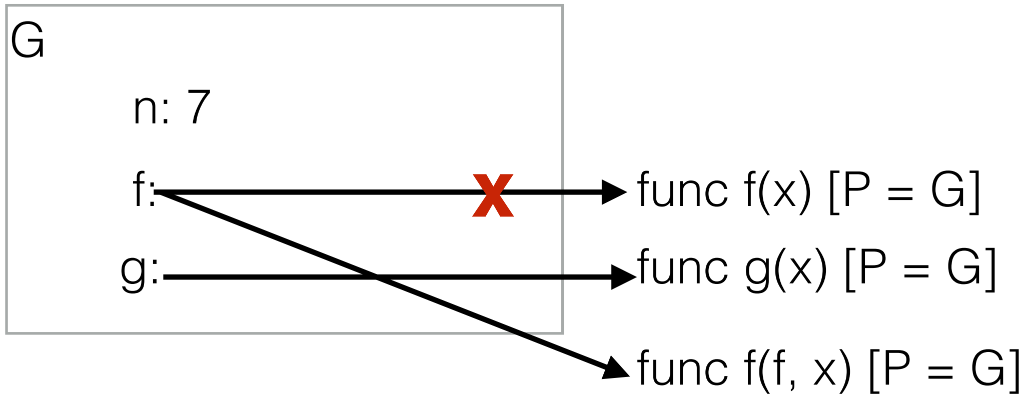
func g(x) [P = G]

func f(f, x) [P = G]

```
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)
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!



G

n: 7

f:

g:

func f(x) [P = G]

func g(x) [P = G]

func f(f, x) [P = G]

```
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)
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f

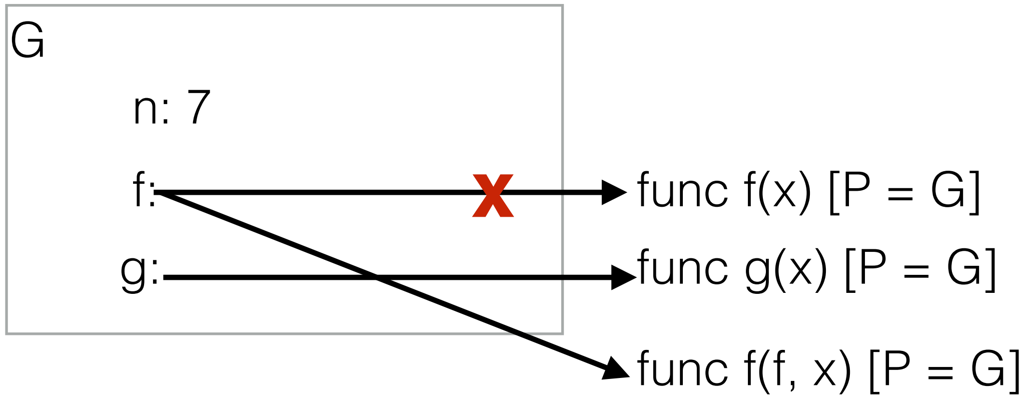
look for what the name g is bound to in the current frame

evaluate g

look for what the name g is bound to in the current frame

evaluate n

look for what the name n is bound to in the current frame

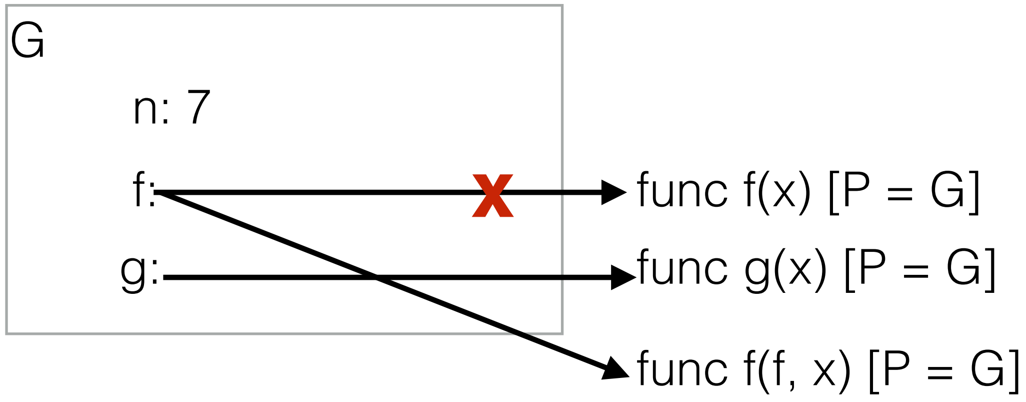


```
n = 7

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)
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS



```
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)
```

f1

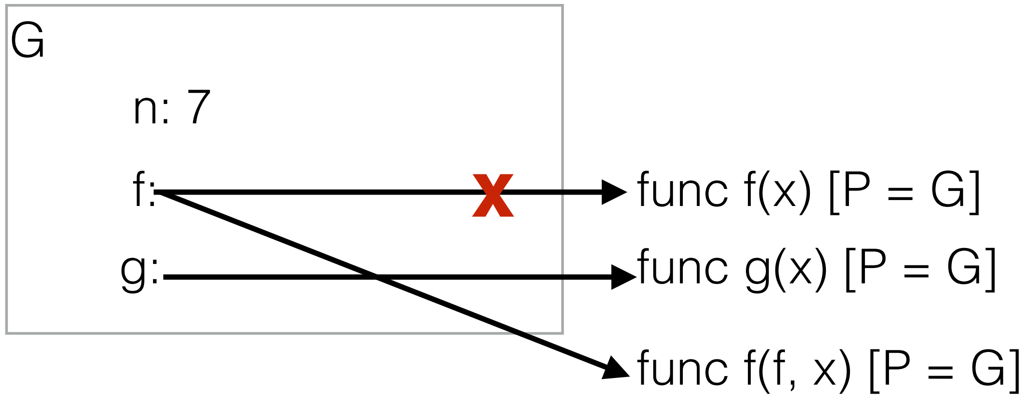
assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame

DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call



```
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)    f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

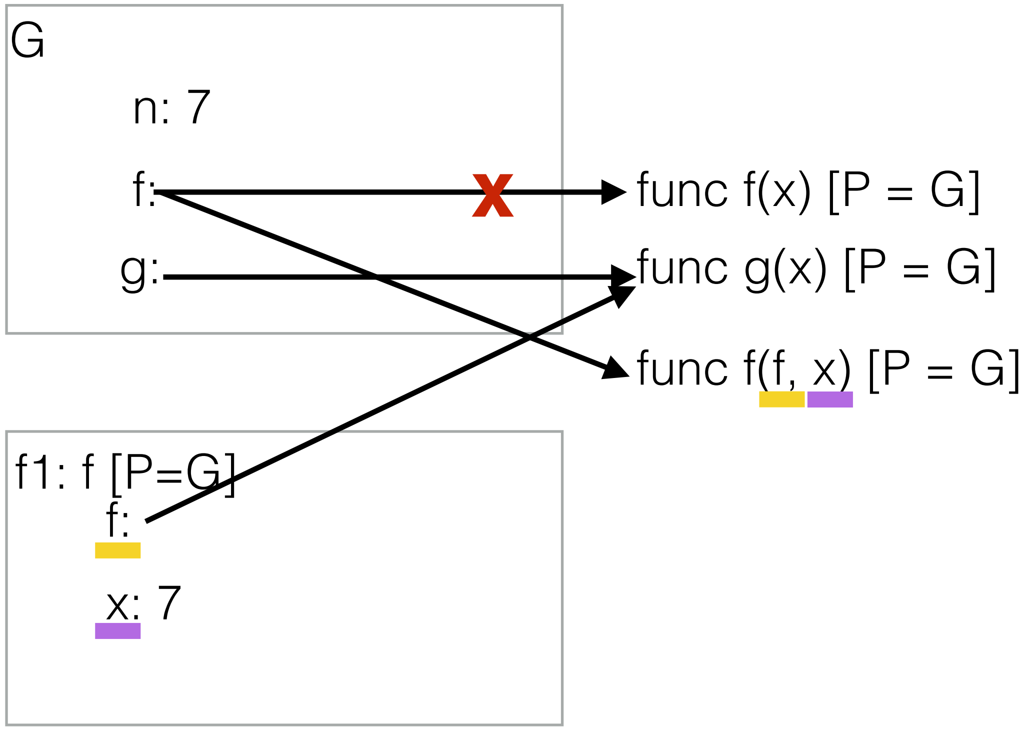
function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame

DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated



```
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) f2

f = f(g, n)  f1

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

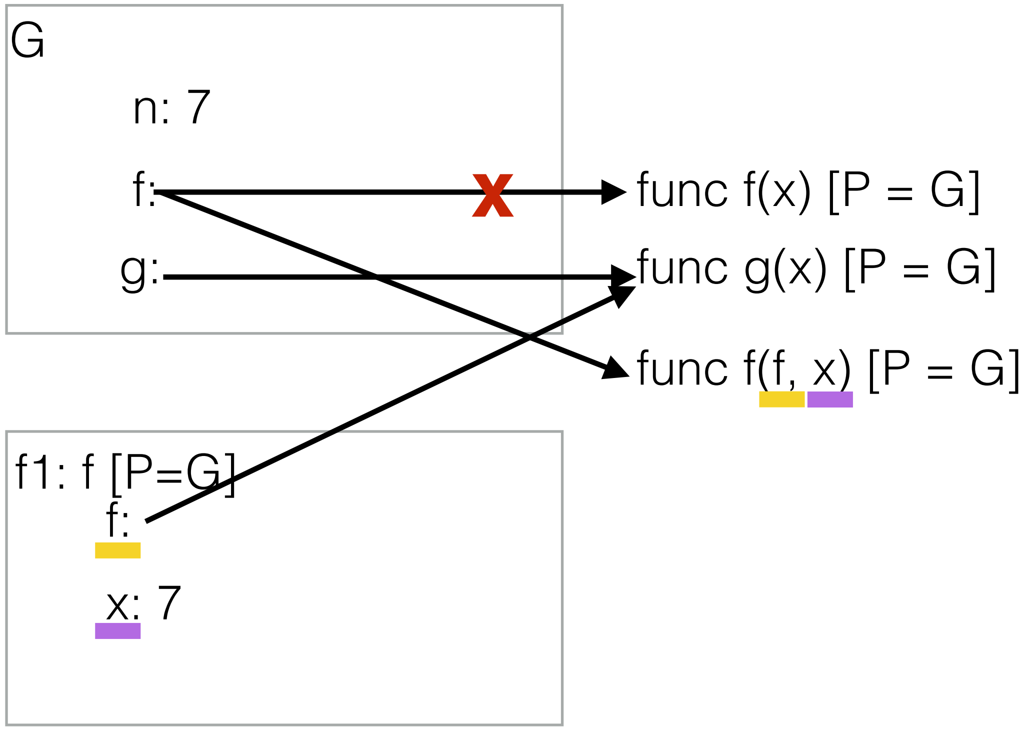
assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2

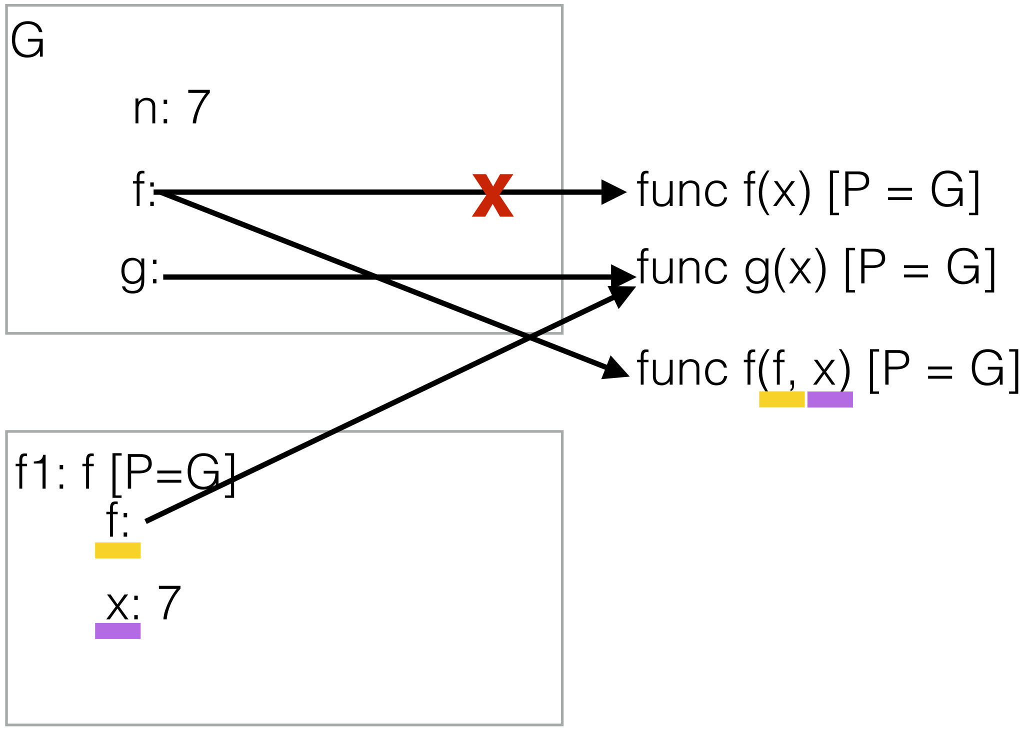


```
n = 7

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) f2

f = f(g, n)  f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
Leave an indication that this is where in the code you will come back to after completing the function call
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands



```
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) f2

f = f(g, n)  f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f

look for what the name g is bound to in the current frame

evaluate g

look for what the name g is bound to in the current frame

evaluate n

look for what the name n is bound to in the current frame

DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

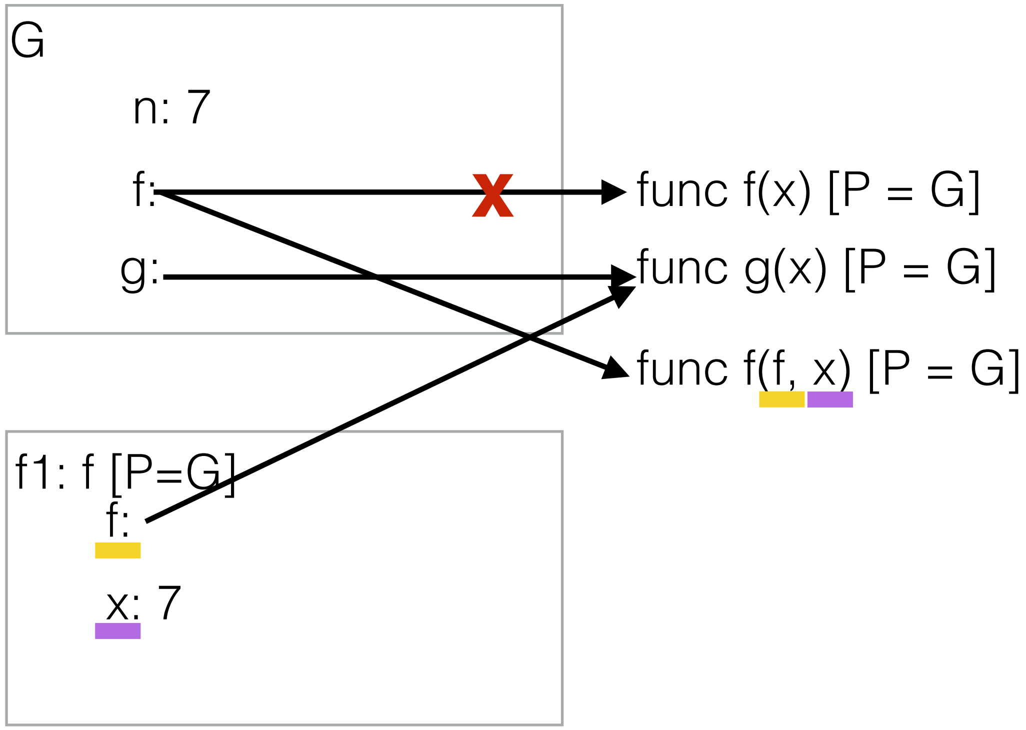
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated

whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2

evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in

note that x is also just passed in and we must look up n in G



```
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) f2

f = f(g, n)   f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f

look for what the name g is bound to in the current frame

evaluate g

look for what the name g is bound to in the current frame

evaluate n

look for what the name n is bound to in the current frame

DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

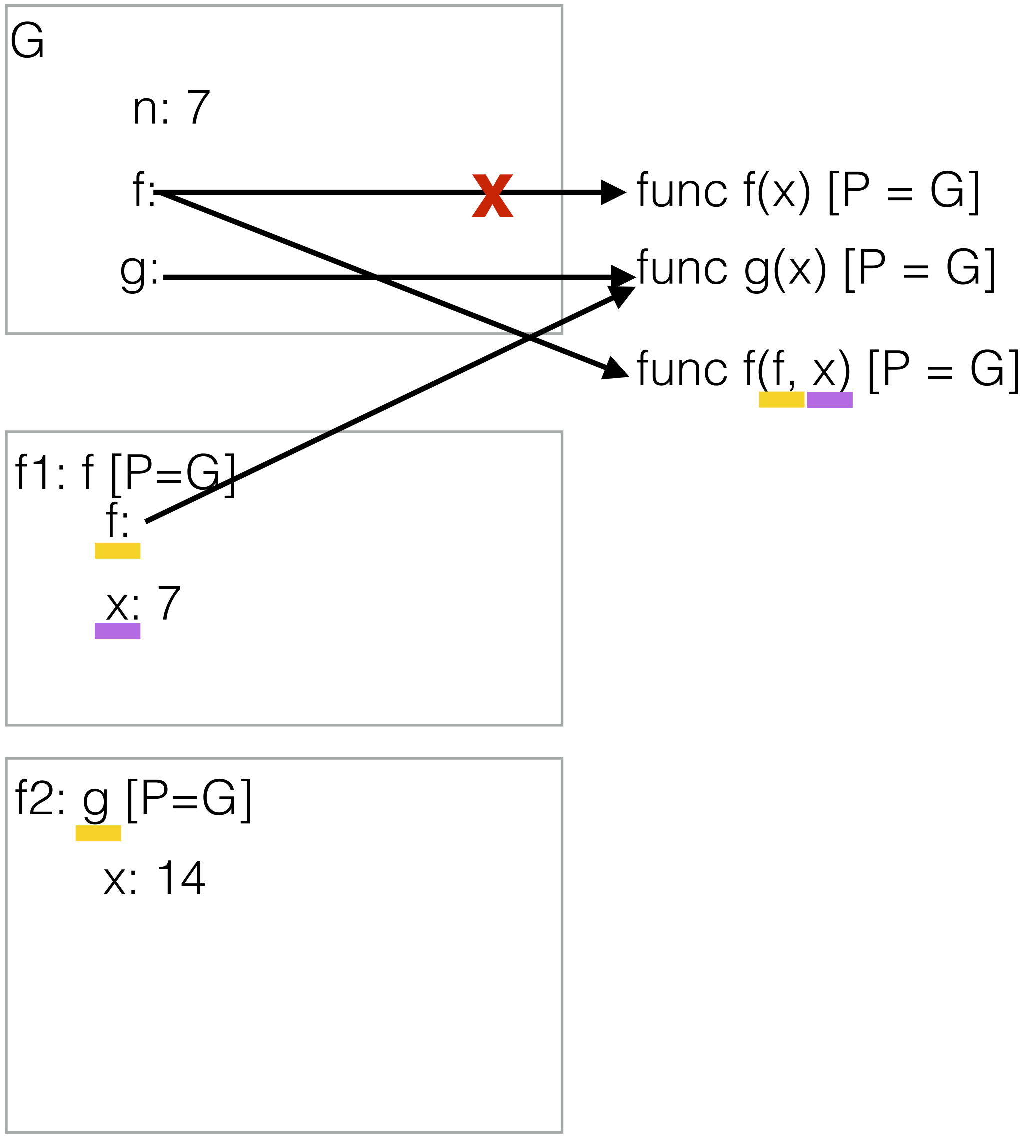
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated

whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2

evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in

note that x is also just passed in and we must look up n in G



G

n: 7

f:

g:

func f(x) [P = G]

func g(x) [P = G]

func f(f, x) [P = G]

f1: f [P=G]

f:

x: 7

f2: g [P=G]

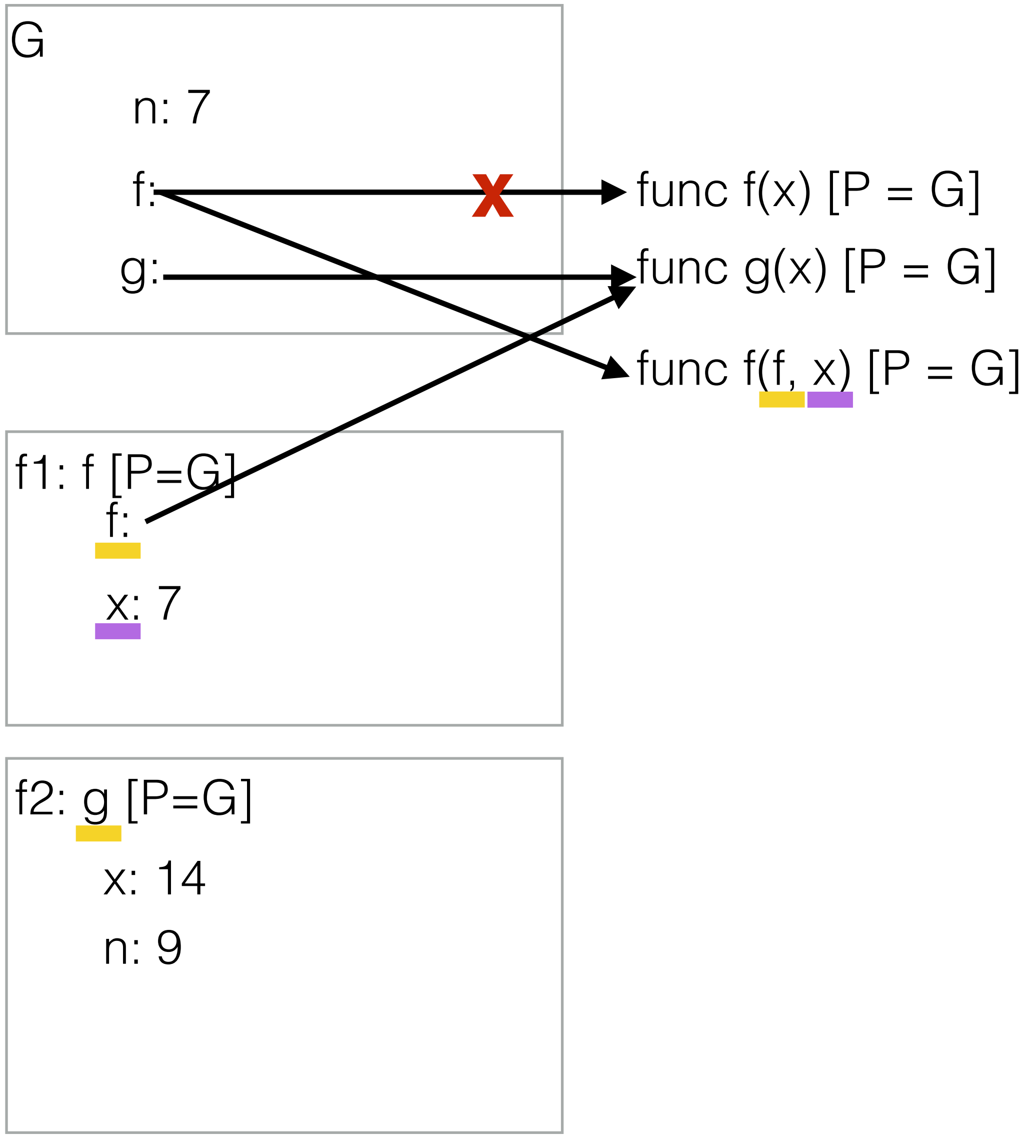
x: 14

```
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)  f2
f = f(g, n)   f1
g = (lambda y: y())(f)
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
Leave an indication that this is where in the code you will come back to after completing the function call
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands
note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

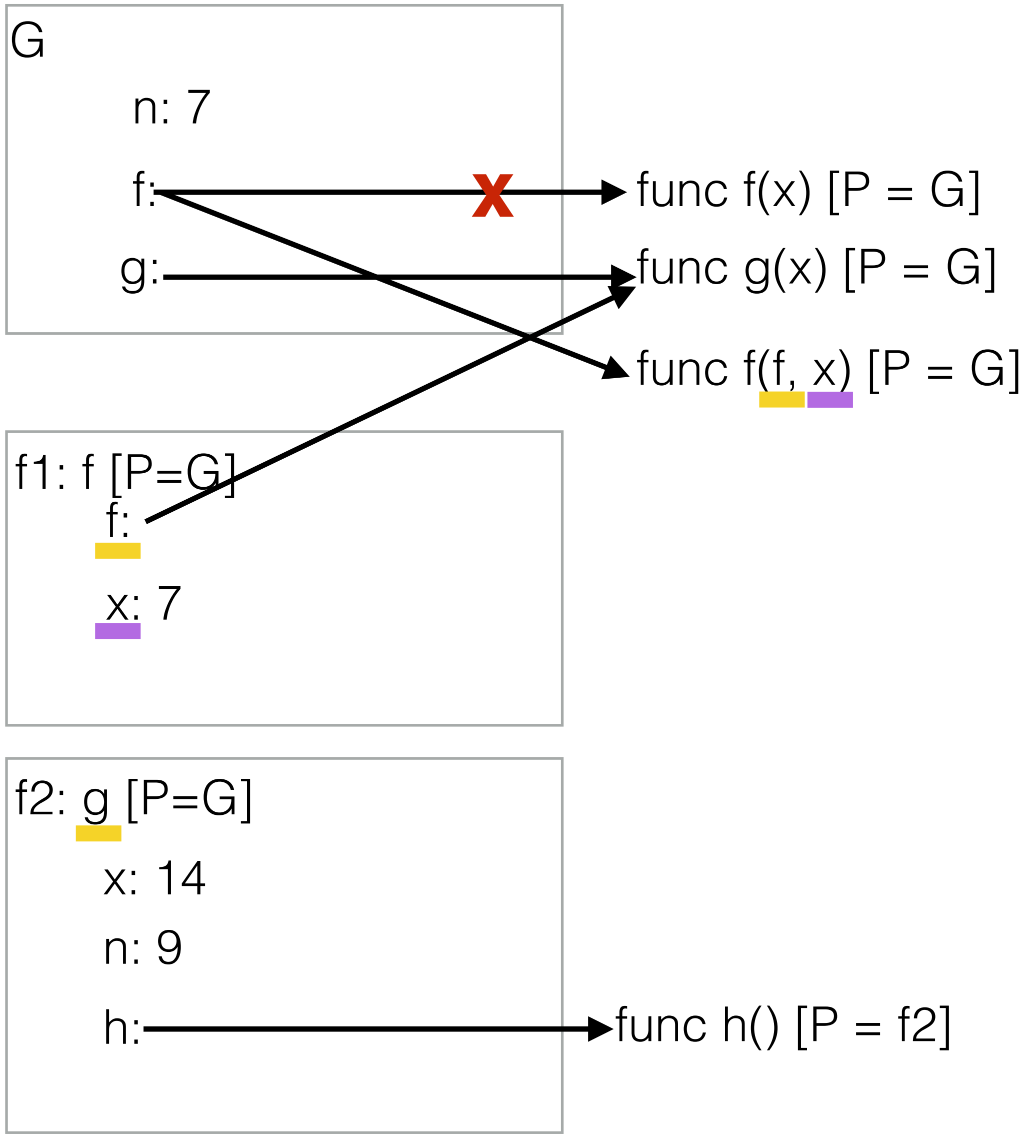


```
n = 7

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) f2

f = f(g, n)    f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
Leave an indication that this is where in the code you will come back to after completing the function call
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands
note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G



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

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

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

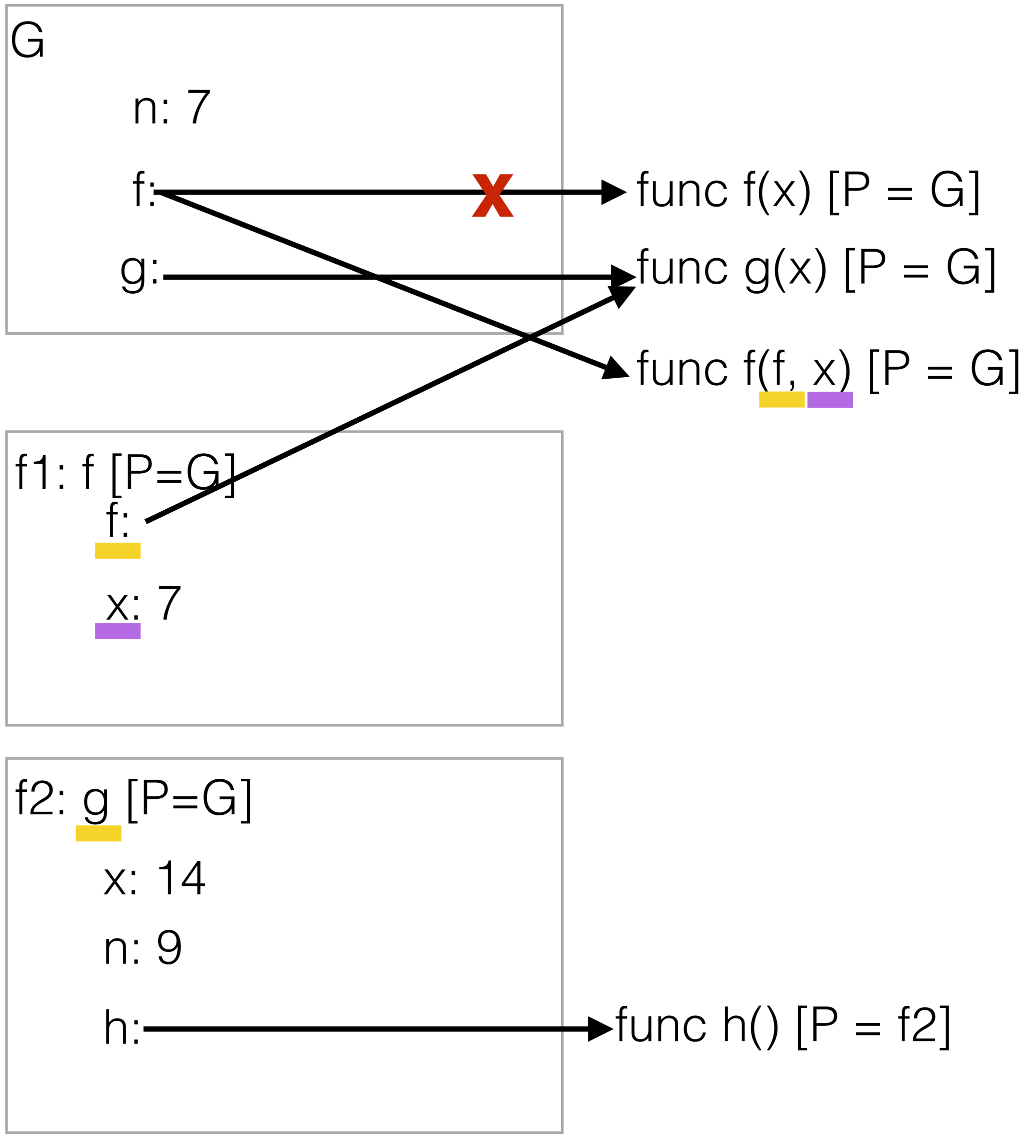
```
n = 7

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) f2

f = f(g, n)   f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
Leave an indication that this is where in the code you will come back to after completing the function call
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands
note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?



G

n: 7

f:

g:

func f(x) [P = G]

func g(x) [P = G]

func f(f, x) [P = G]

f1: f [P=G]

f:

x: 7

f2: g [P=G]

x: 14

n: 9

h:

func h() [P = f2]

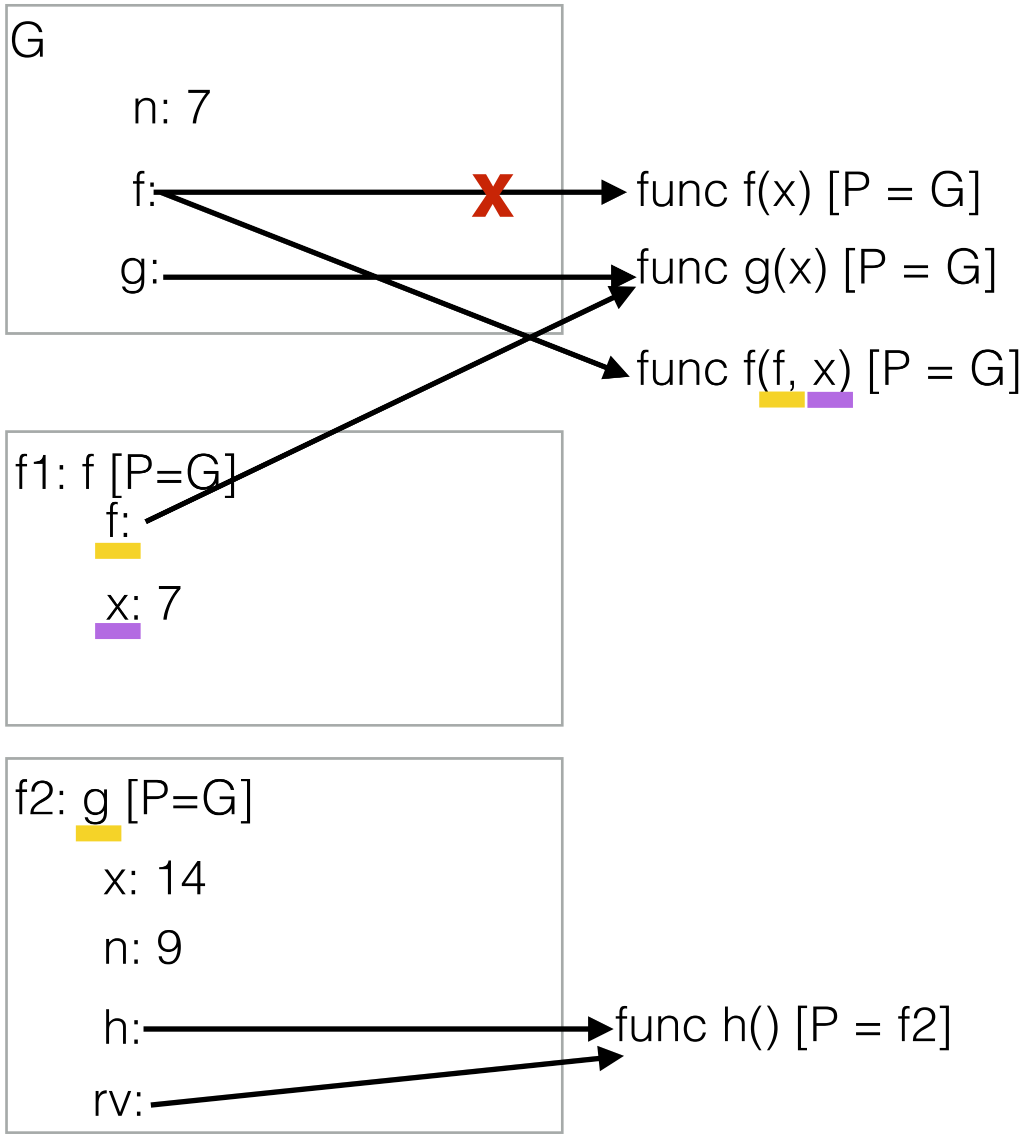
```
n = 7
def f(x):
    n = 8
    return x + 1
def g(x):
    n = 9
    def h():
        return x + 1
    return h
def f(f, x):
    return f(x + n)  f2
f = f(g, n)  f1
g = (lambda y: y())(f)
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
Leave an indication that this is where in the code you will come back to after completing the function call
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands
note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?



```
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) f2

▶f = f(g, n)   f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

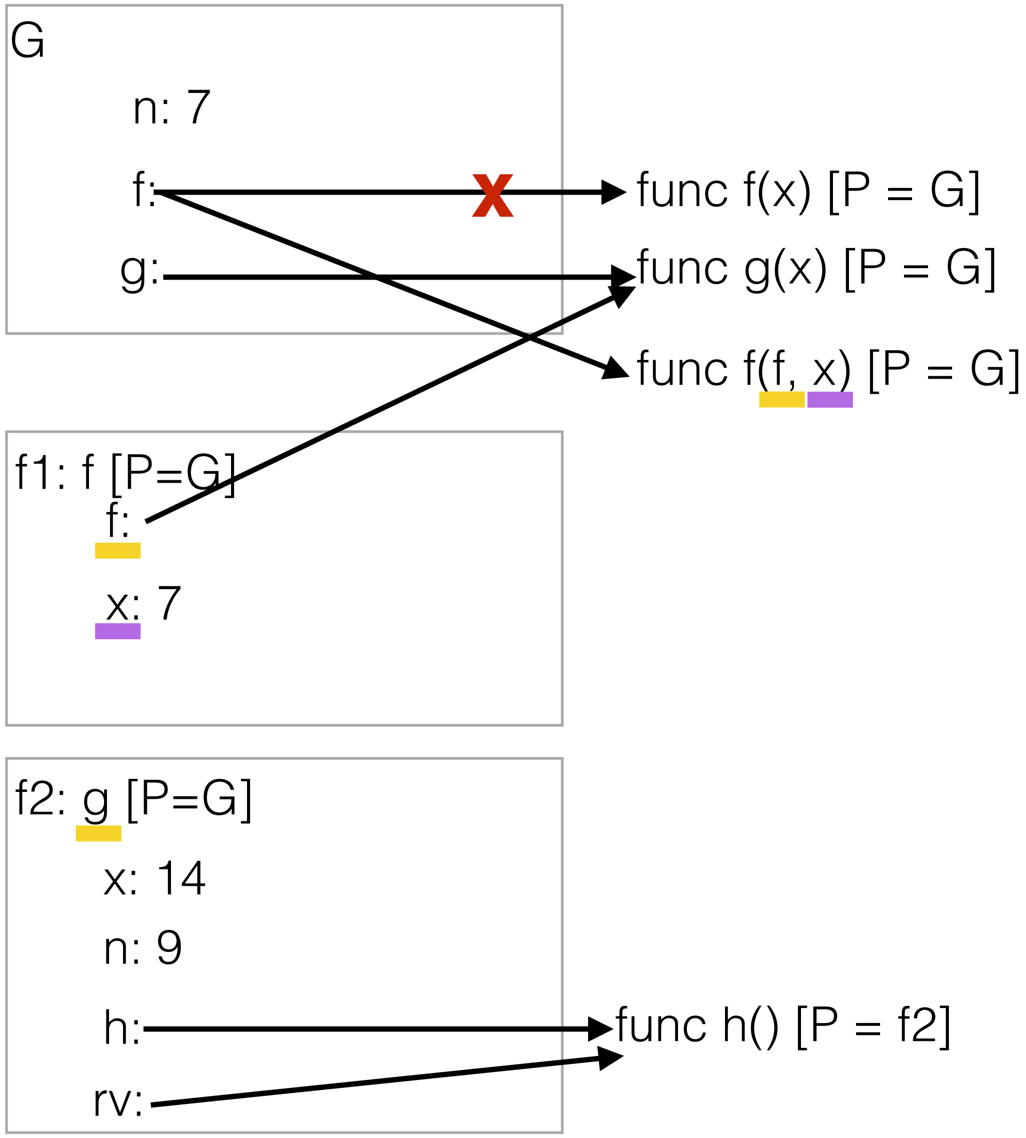
Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)



```
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) f2

f = f(g, n)     f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

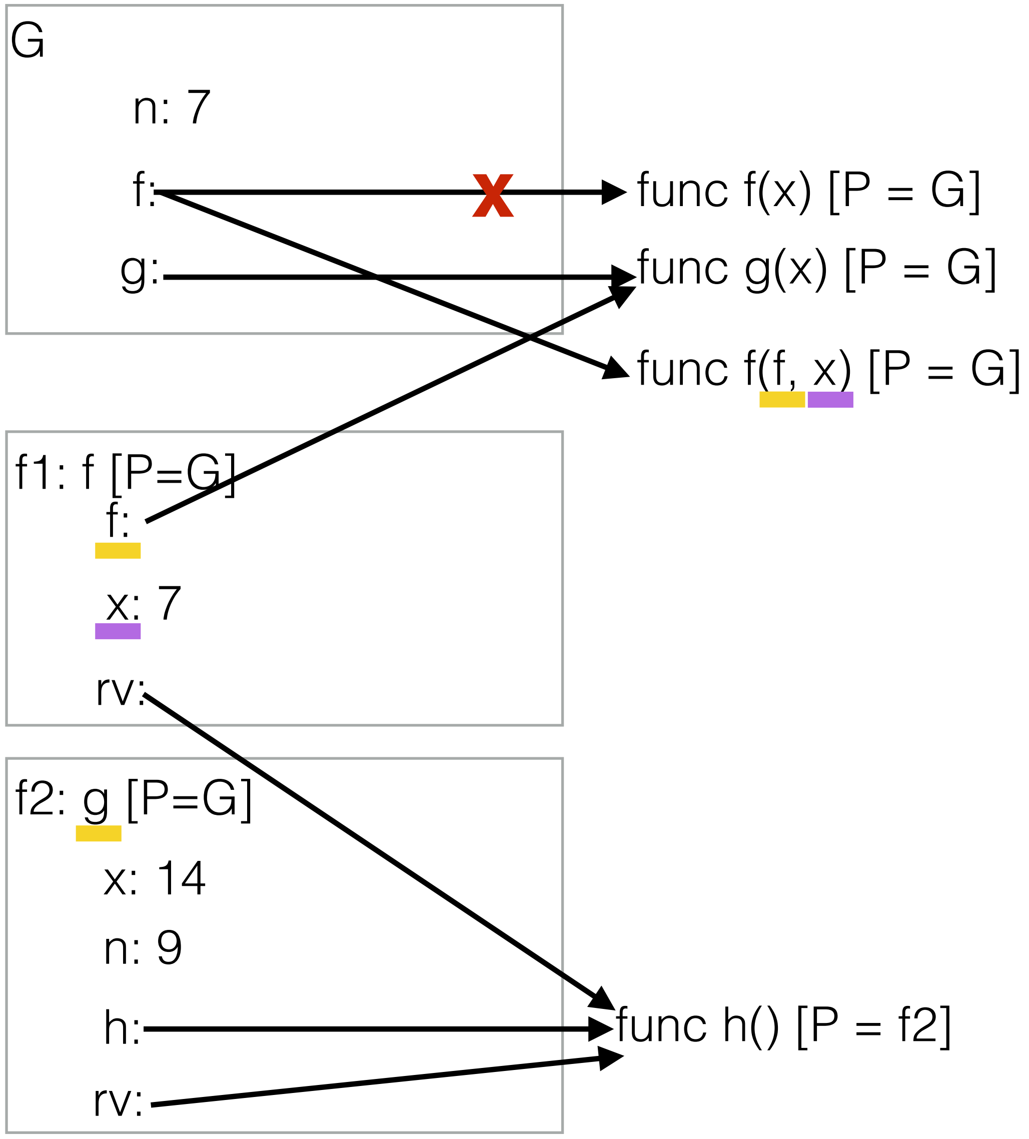
Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)



G

n: 7

f:

g:

func f(x) [P = G]

func g(x) [P = G]

func f(f, x) [P = G]

f1: f [P=G]

f:

x: 7

rv:

f2: g [P=G]

x: 14

n: 9

h:

rv:

func h() [P = f2]

```
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) f2

f = f(g, n)   f1

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

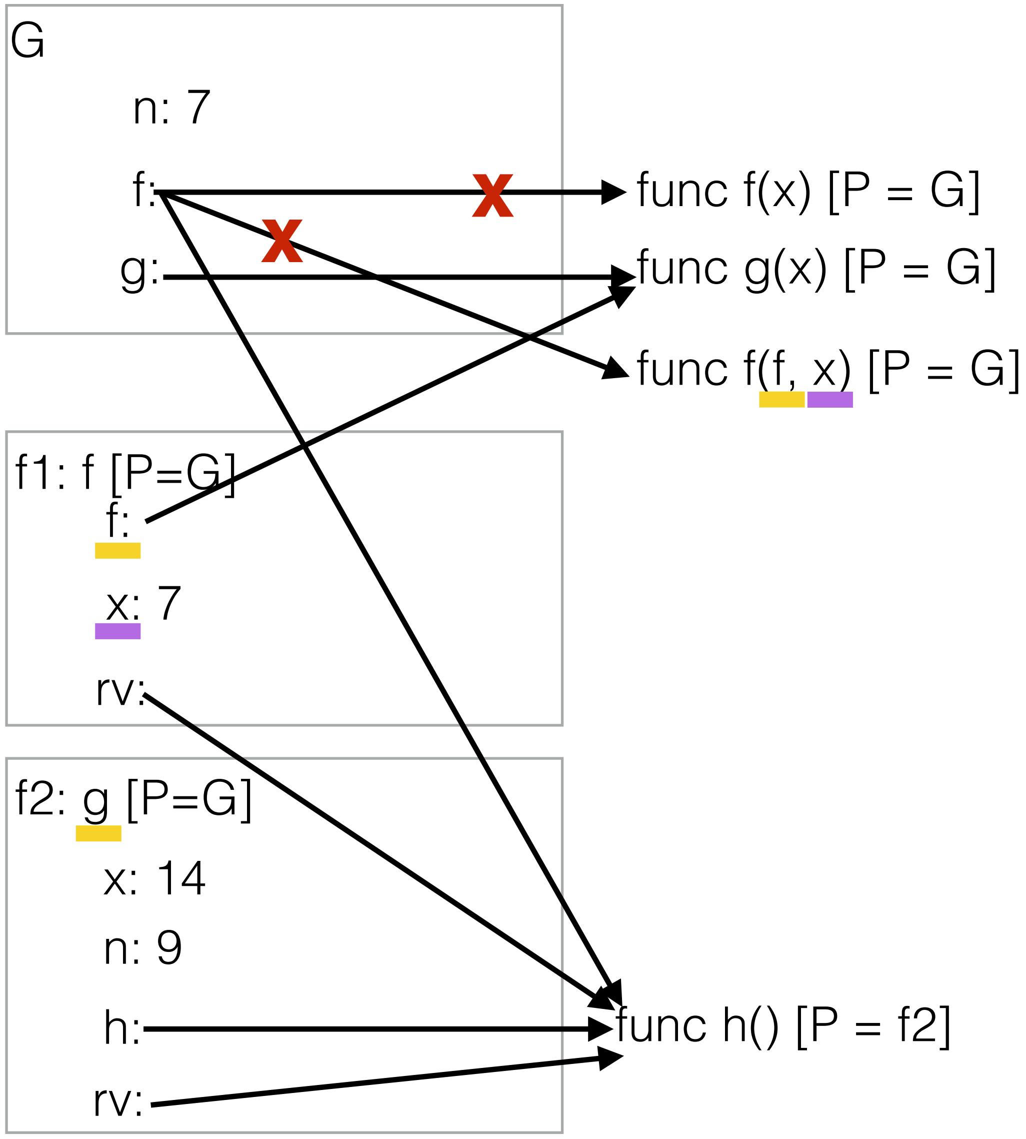
assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
Leave an indication that this is where in the code you will come back to after completing the function call
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands
note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)



```
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)
```

f2

f1

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

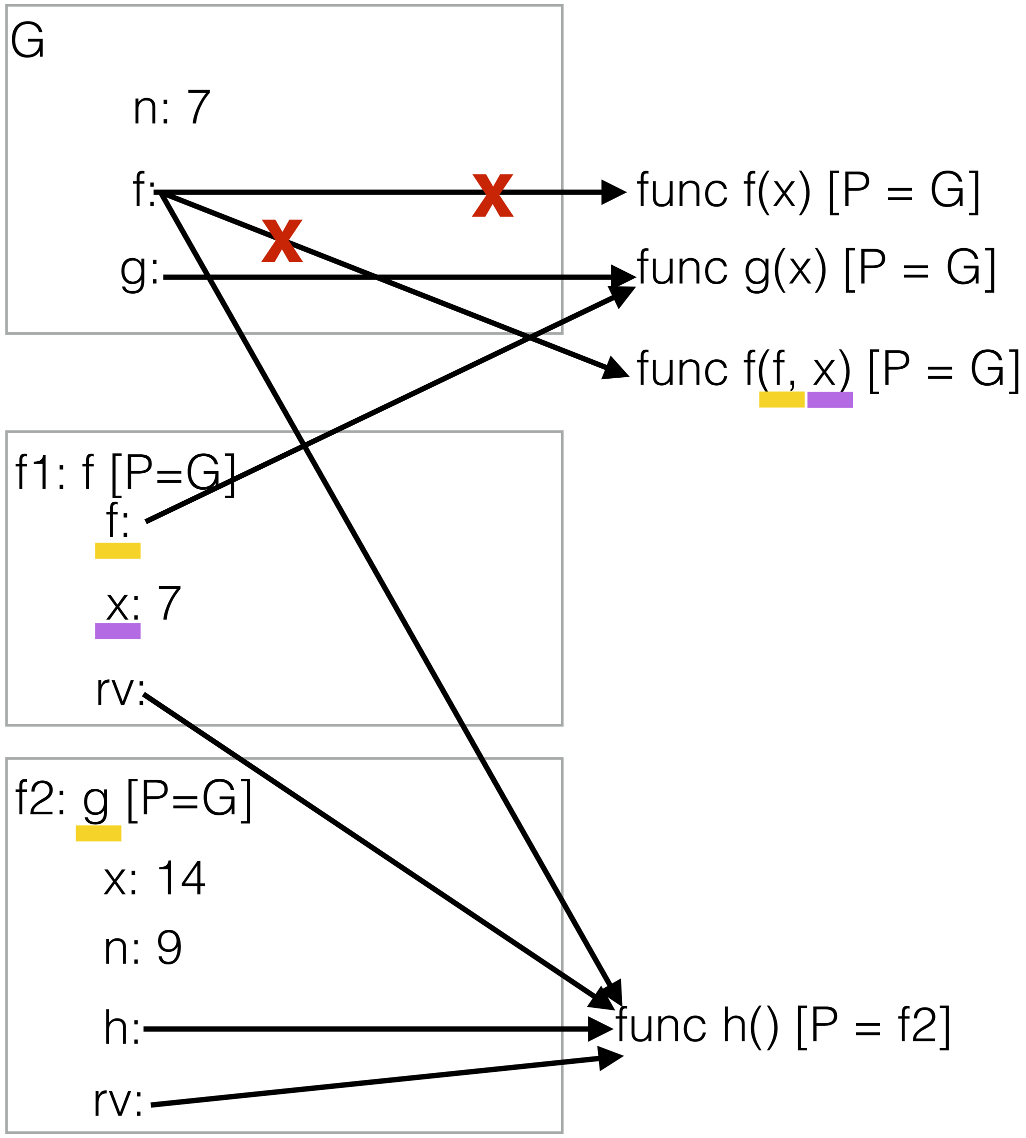
evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands
note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)



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

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

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

```
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)  f2

f = f(g, n)  f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

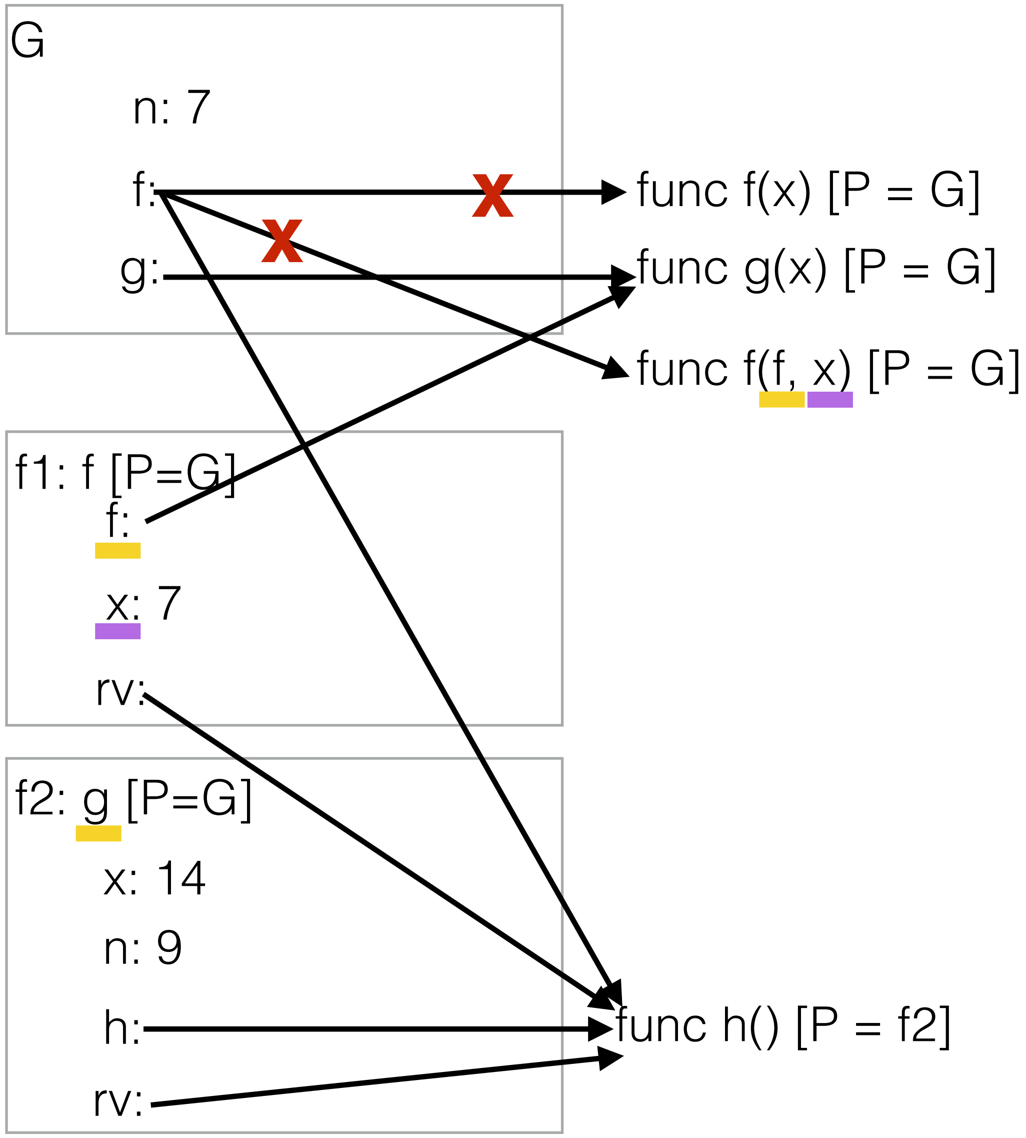
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?



```
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) f2

f = f(g, n)  f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

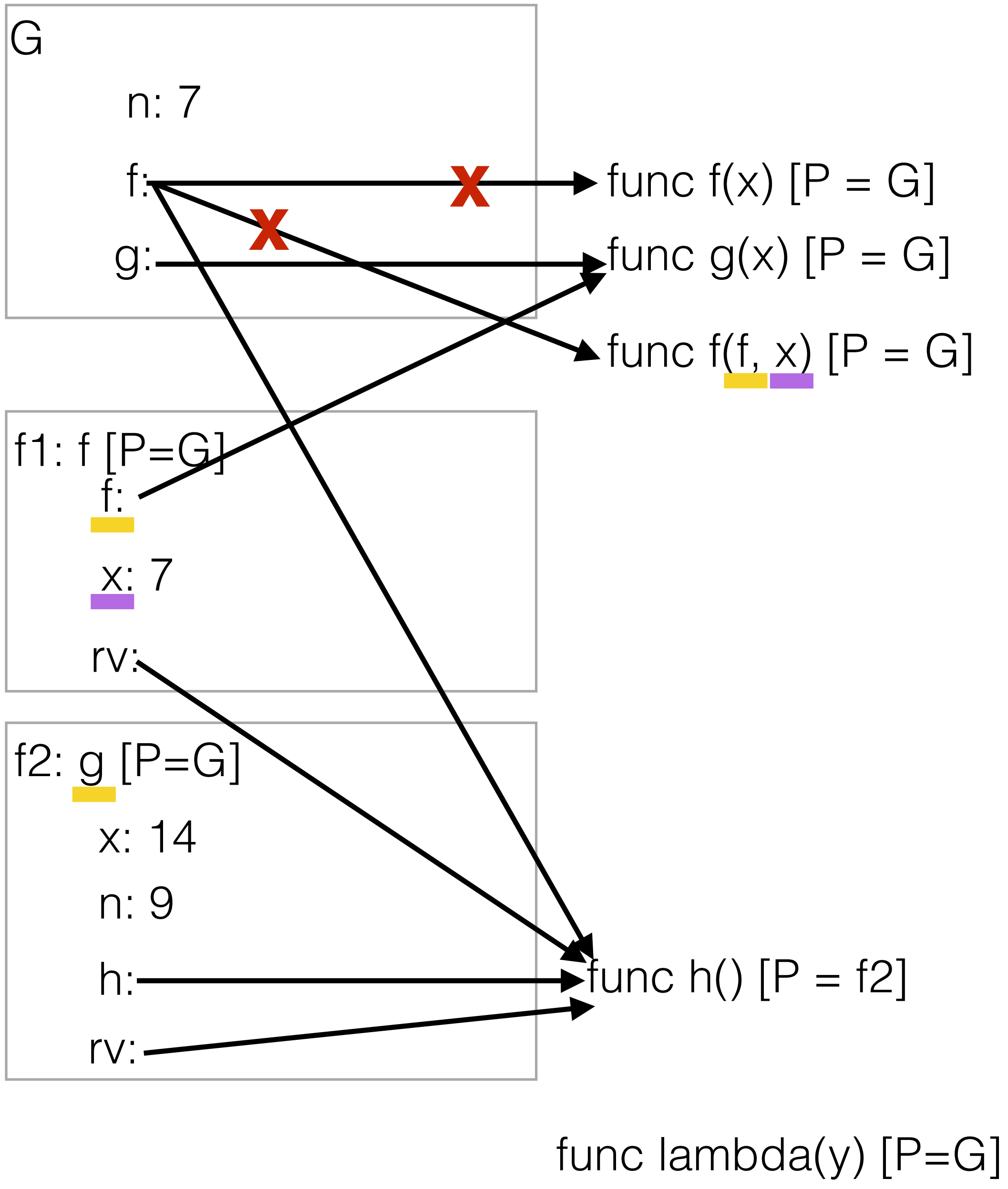
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?



G
n: 7
f:
g:

func f(x) [P = G]
func g(x) [P = G]
func f(f, x) [P = G]

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

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

func h() [P = f2]

func lambda(y) [P=G]

```
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)  f2
f = f(g, n)  f1
g = (lambda y: y())(f)
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

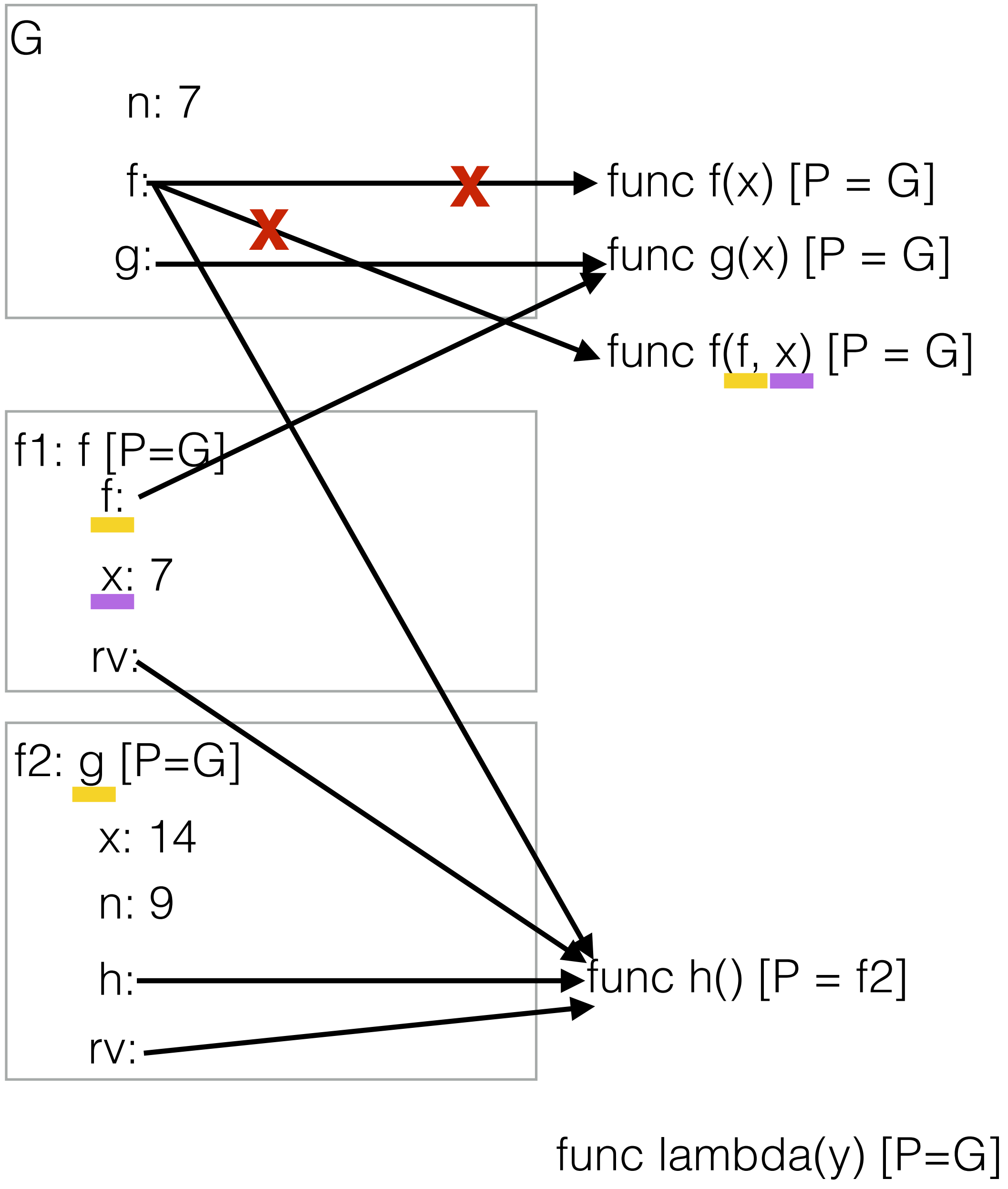
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?
we just assigned f to point to the function h, so we pass in the function h as y



```
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) f2

f = f(g, n)   f1

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

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

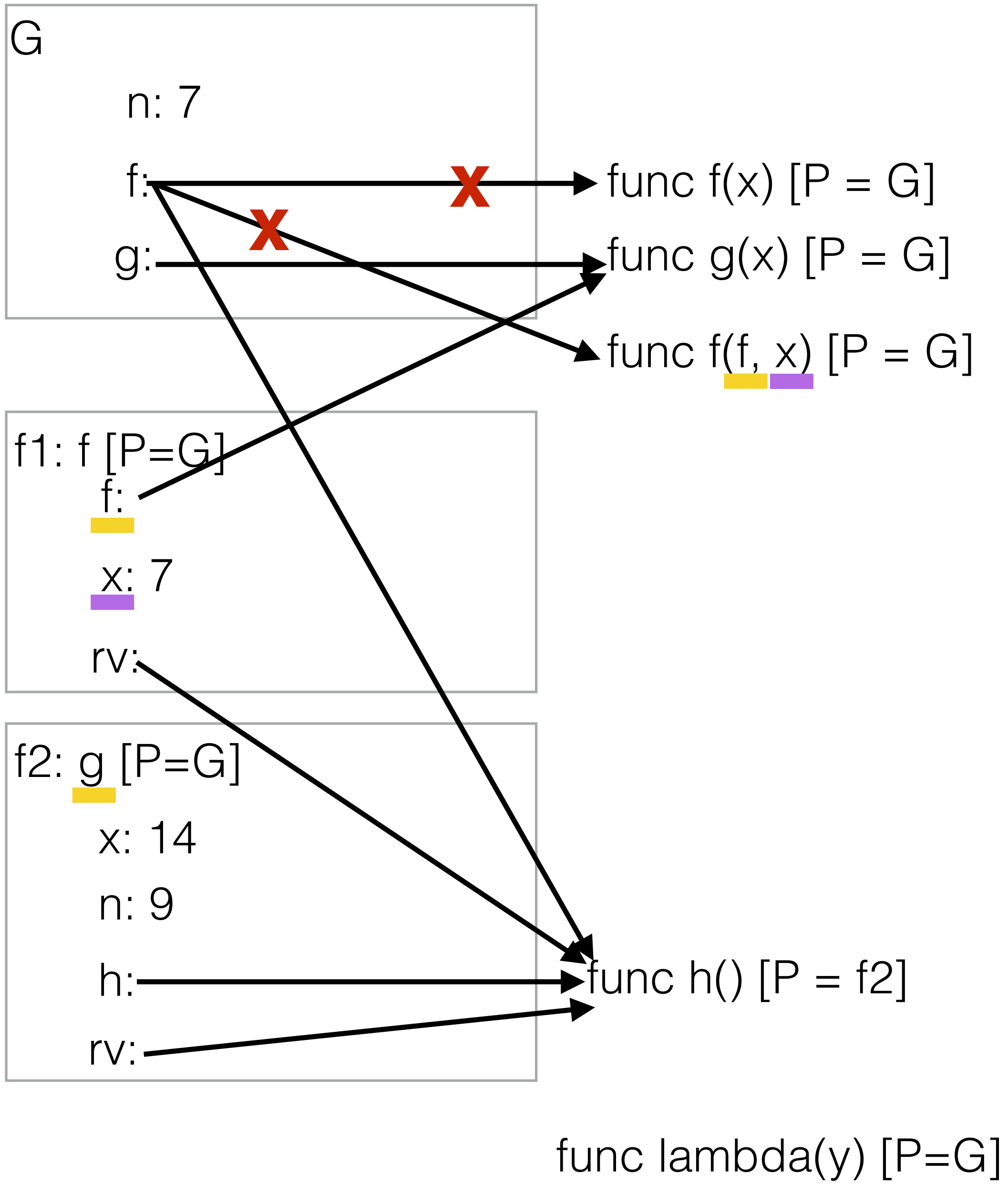
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?
we just assigned f to point to the function h, so we pass in the function h as y
Checkpoint: why is lambda's parent G?



```
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) f2

f = f(g, n)    f1

g = (lambda y: y())(f)
                   f3
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

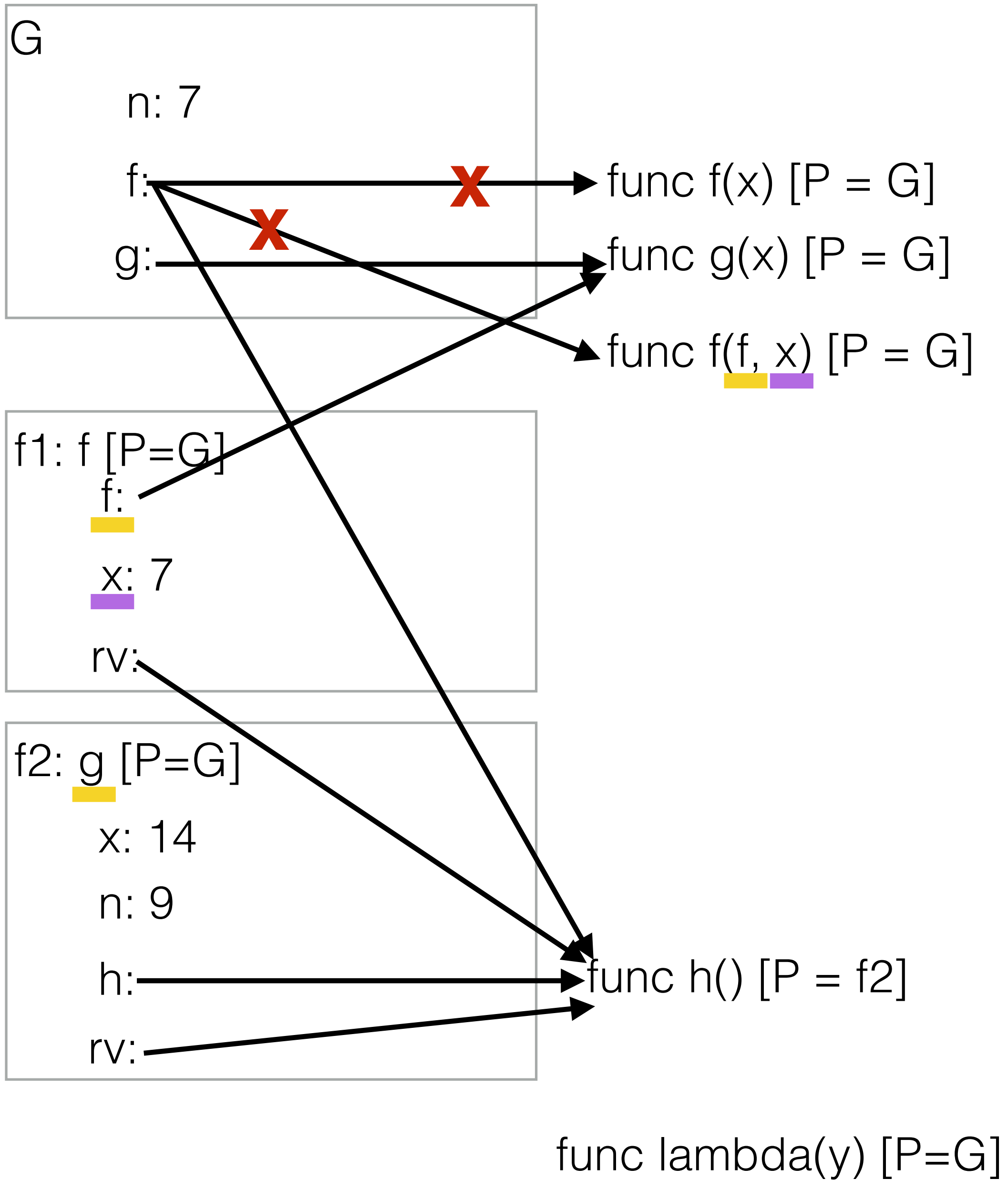
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?
we just assigned f to point to the function h, so we pass in the function h as y
Checkpoint: why is lambda's parent G?



```
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)   f2
f = f(g, n)   f1
g = (lambda y: y())(f)
                     f3
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f

look for what the name g is bound to in the current frame

evaluate g

look for what the name g is bound to in the current frame

evaluate n

look for what the name n is bound to in the current frame

DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated

whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2

evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in

note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

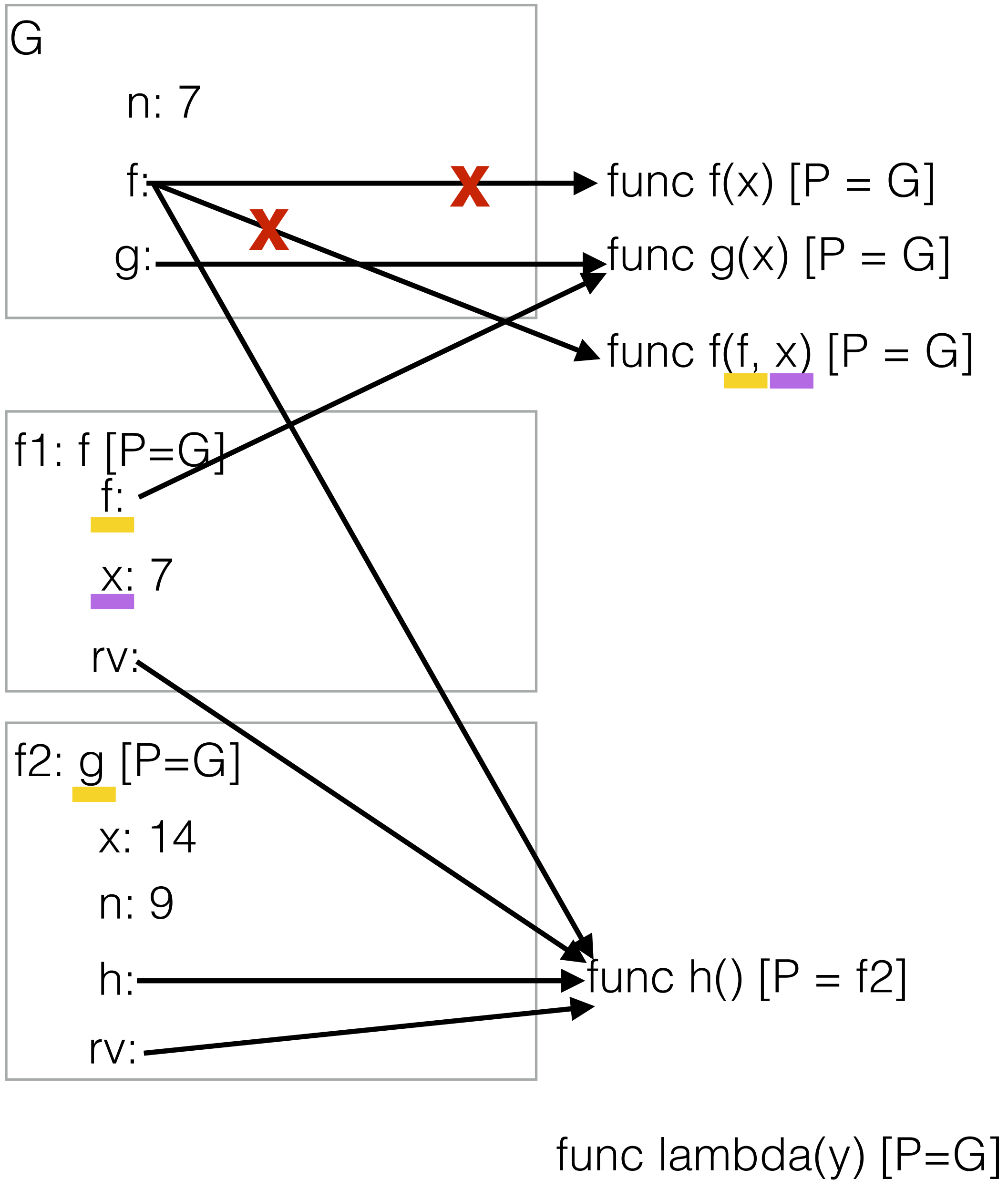
what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?

we just assigned f to point to the function h, so we pass in the function h as y

Checkpoint: why is lambda's parent G?

we have a function call, leave a mark where to return to once you complete f3



```
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) f2

f = f(g, n)  f1

g = (lambda y: y())(f)
                   f3
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

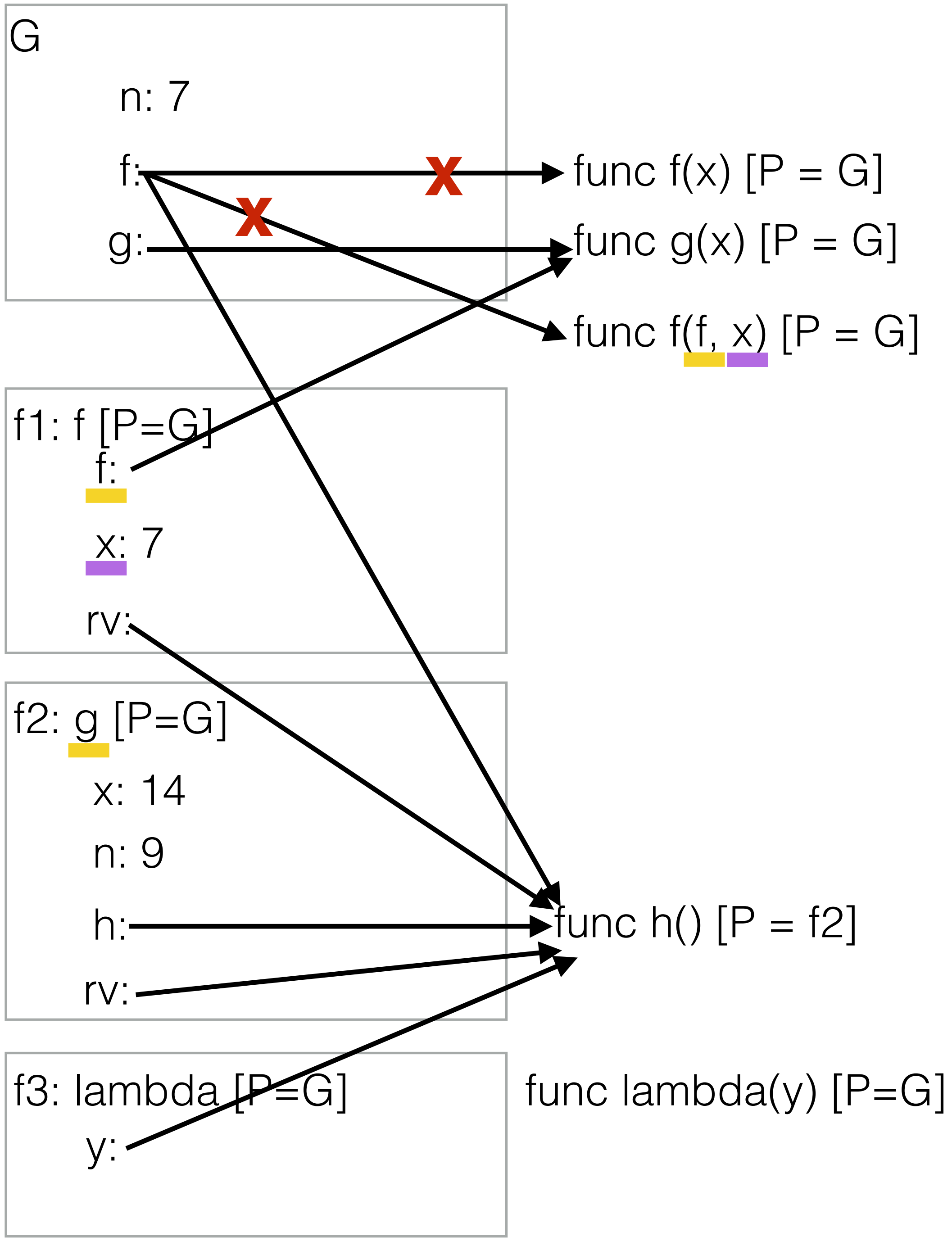
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?
we just assigned f to point to the function h, so we pass in the function h as y
Checkpoint: why is lambda's parent G?
we have a function call, leave a mark where to return to once you complete f3



```
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)  f2

f = f(g, n)  f1

g = (lambda y: y())(f)
                    f3
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

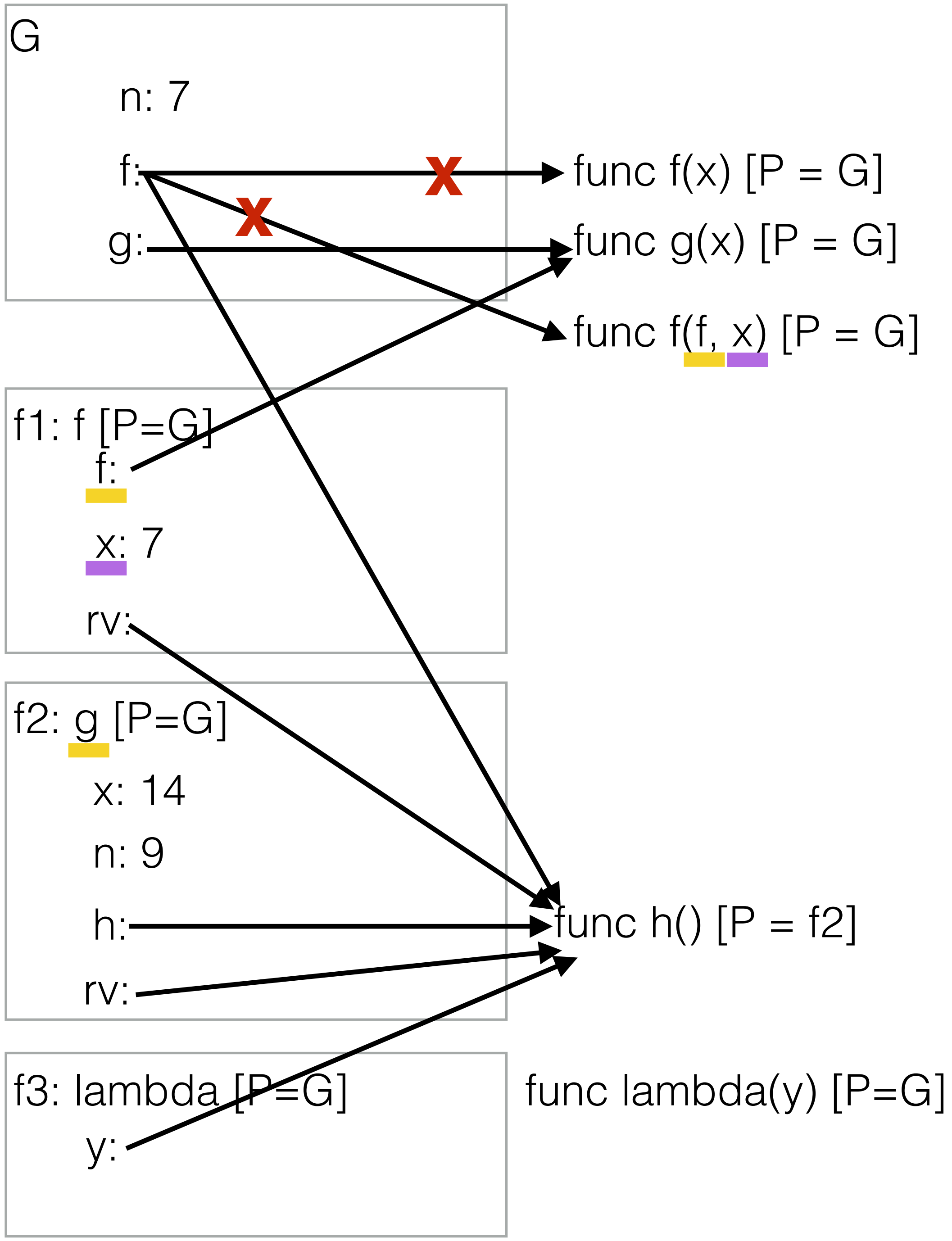
After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?
we just assigned f to point to the function h, so we pass in the function h as y
Checkpoint: why is lambda's parent G?
we have a function call, leave a mark where to return to once you complete f3
what does this lambda do? whats the body of the lambda? try converting the lambda to a normal def statement



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

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

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

f3: lambda [P=G]
y:
func lambda(y) [P=G]

```
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) f2

f = f(g, n)  f1

g = (lambda y: y())(f)
                f4  f3
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f

look for what the name g is bound to in the current frame

evaluate g

look for what the name g is bound to in the current frame

evaluate n

look for what the name n is bound to in the current frame

DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated

whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2

evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in

note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?

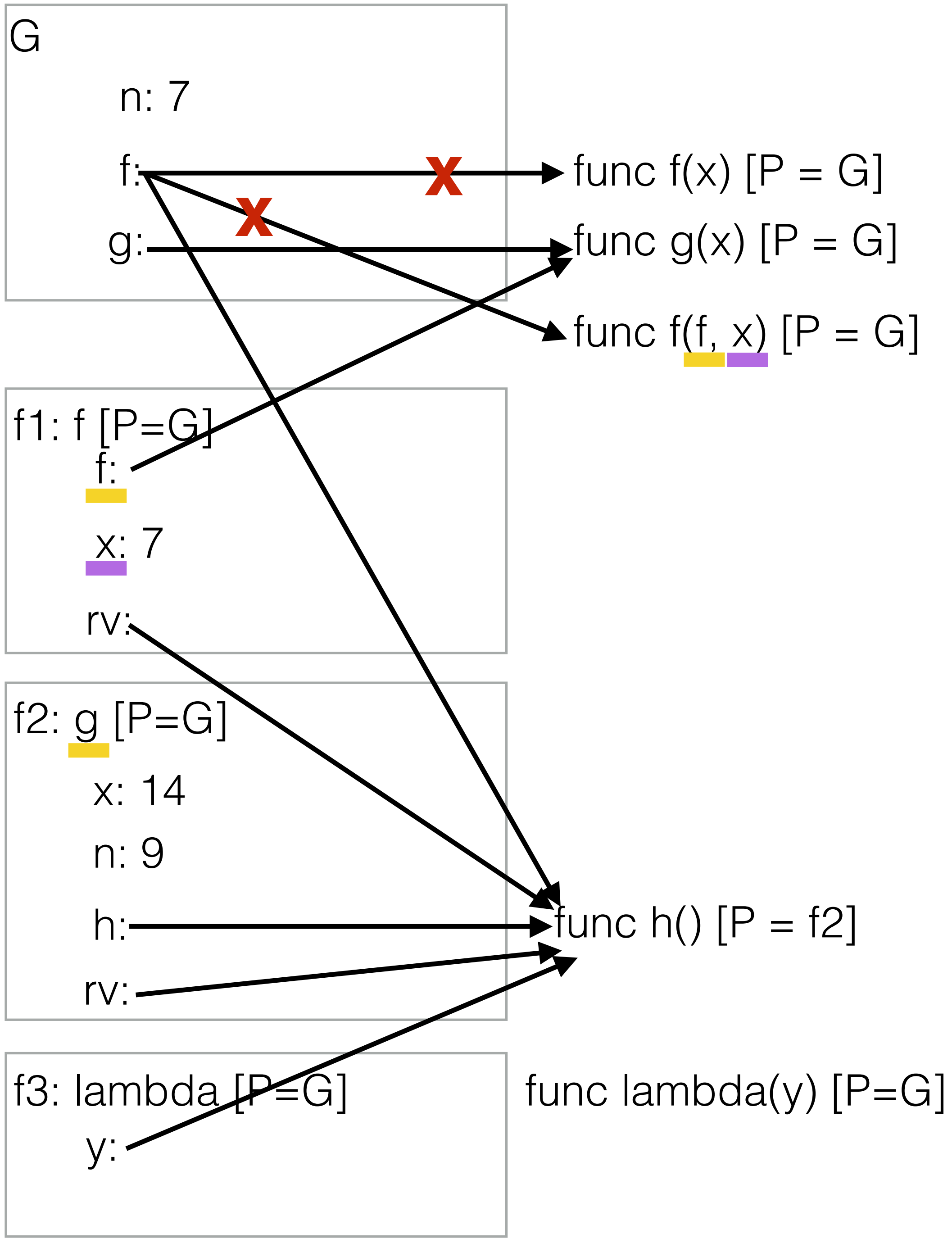
we just assigned f to point to the function h, so we pass in the function h as y

Checkpoint: why is lambda's parent G?

we have a function call, leave a mark where to return to once you complete f3

what does this lambda do? whats the body of the lambda? try converting the lambda to a normal def statement

we have another function call, leave a mark where to return to once you complete f4



```
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) f2

f = f(g, n)  f1

g = (lambda y: y())(f)
                 f4   f3
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

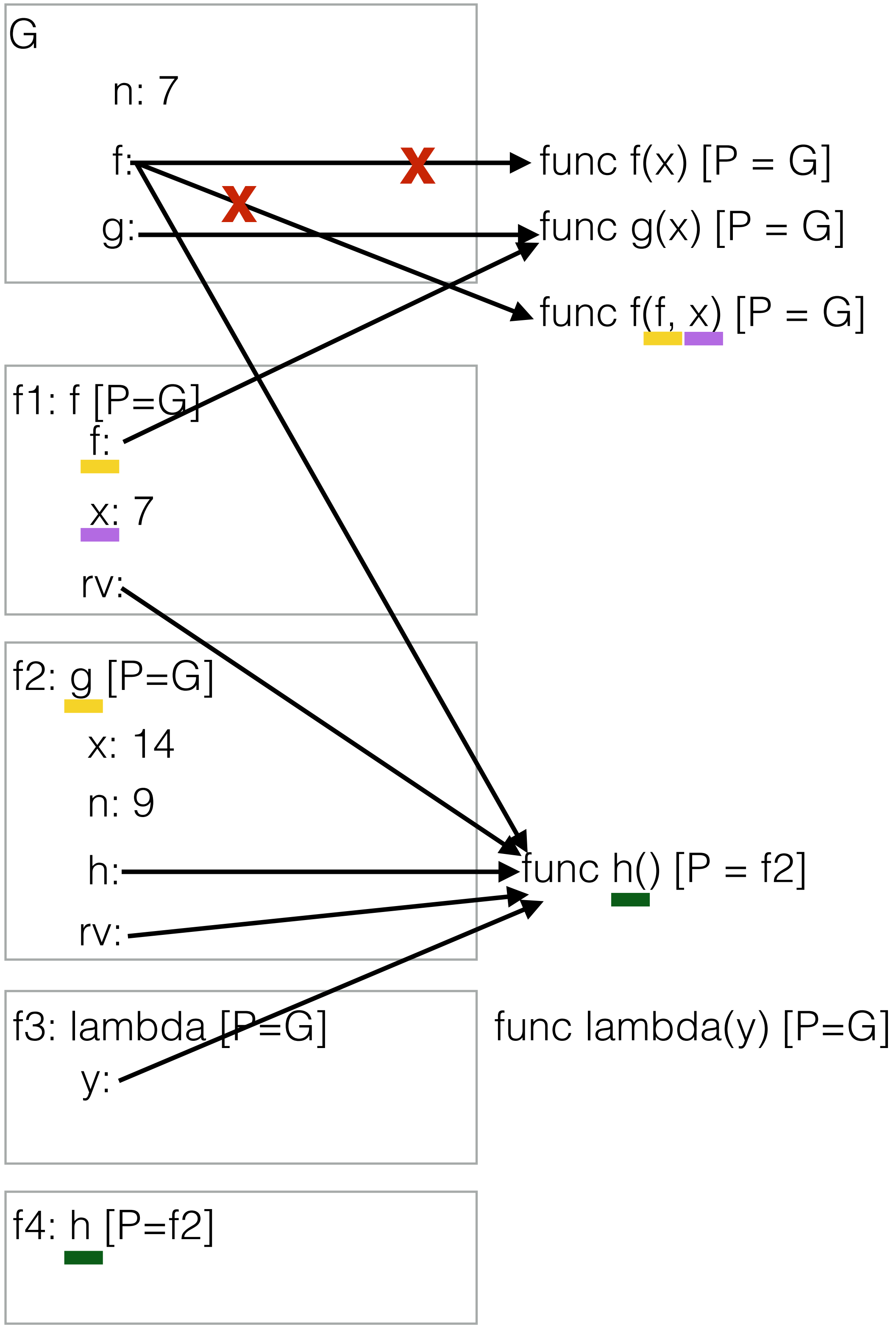
note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)
what does it mean to evaluate a lambda?
we just assigned f to point to the function h, so we pass in the function h as y
Checkpoint: why is lambda's parent G?
we have a function call, leave a mark where to return to once you complete f3
what does this lambda do? whats the body of the lambda? try converting the lambda to a normal def statement

we have another function call, leave a mark where to return to once you complete f4

y is really h, and h just adds x and 1 and returns the sum. we can find x in f2 since that is the parent of h



```
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) f2

f = f(g, n)  f1

g = (lambda y: y())(f)
                 f4   f3
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

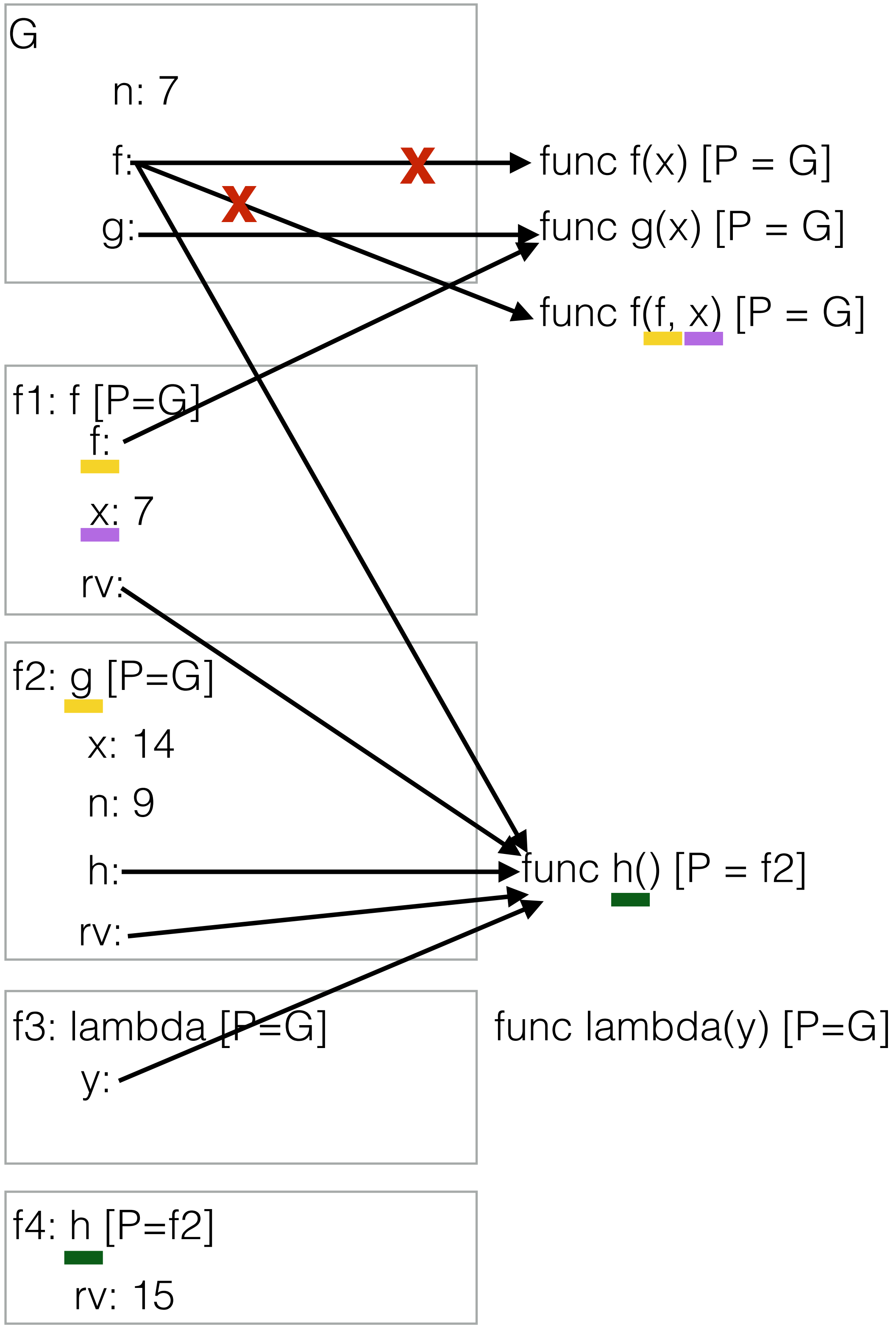
evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands
note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)
what does it mean to evaluate a lambda?
we just assigned f to point to the function h, so we pass in the function h as y
Checkpoint: why is lambda's parent G?
we have a function call, leave a mark where to return to once you complete f3
what does this lambda do? whats the body of the lambda? try converting the lambda to a normal def statement
we have another function call, leave a mark where to return to once you complete f4
y is really h, and h just adds x and 1 and returns the sum. we can find x in f2 since that is the parent of h



```
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) f2
f = f(g, n)  f1
g = (lambda y: y())(f)
               f4   f3
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f

look for what the name g is bound to in the current frame

evaluate g

look for what the name g is bound to in the current frame

evaluate n

look for what the name n is bound to in the current frame

DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated

whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2

evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in

note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?

we just assigned f to point to the function h, so we pass in the function h as y

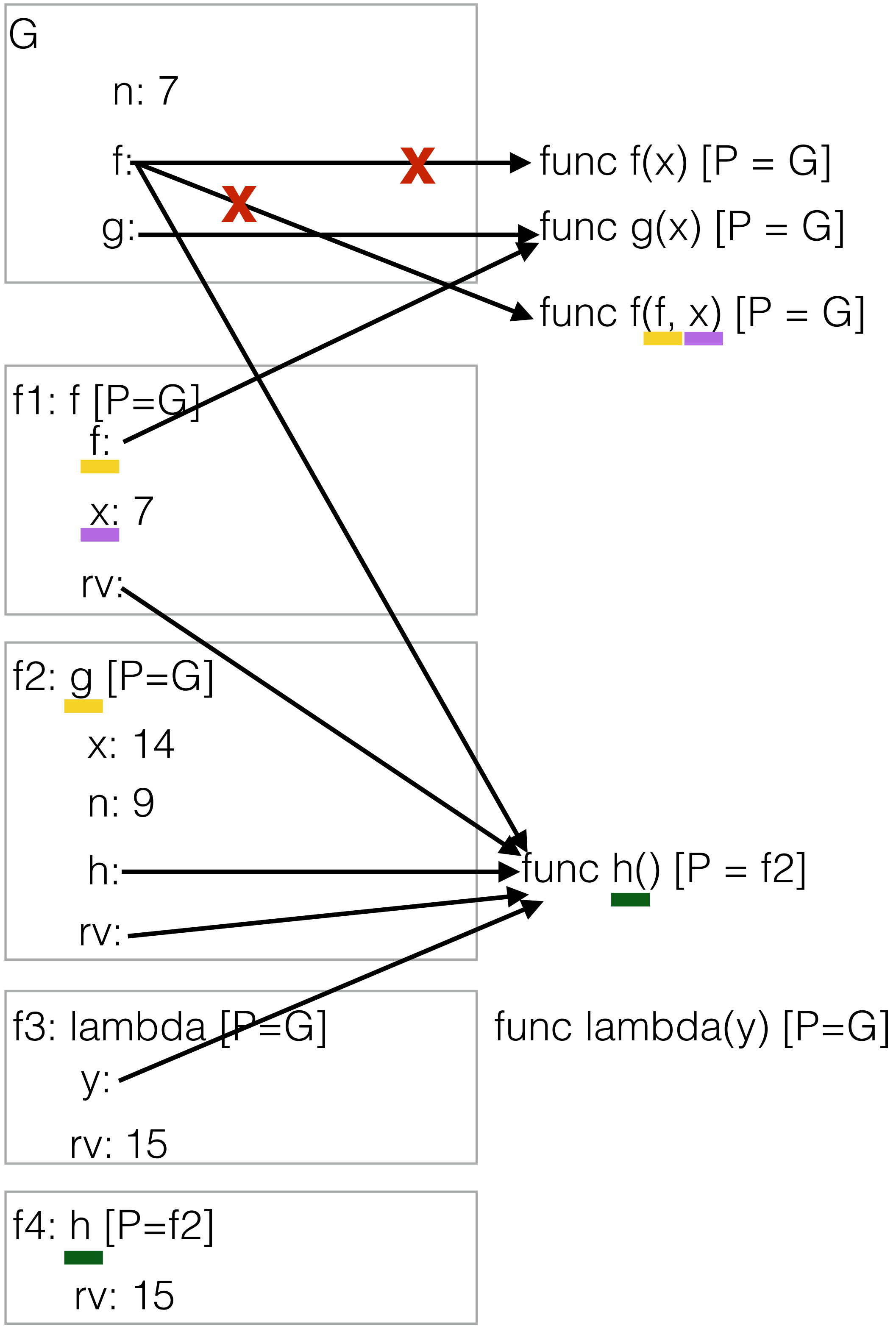
Checkpoint: why is lambda's parent G?

we have a function call, leave a mark where to return to once you complete f3

what does this lambda do? whats the body of the lambda? try converting the lambda to a normal def statement

we have another function call, leave a mark where to return to once you complete f4

y is really h, and h just adds x and 1 and returns the sum. we can find x in f2 since that is the parent of h



G
n: 7
f:
g:

func f(x) [P = G]
func g(x) [P = G]
func f(f, x) [P = G]

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

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

func h() [P = f2]

f3: lambda [P=G]
y:
rv: 15

func lambda(y) [P=G]

f4: h [P=f2]
rv: 15

```
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)
```

f2

f1

f4 f3

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f

look for what the name g is bound to in the current frame

evaluate g

look for what the name g is bound to in the current frame

evaluate n

look for what the name n is bound to in the current frame

DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated

whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2

evaluate operator and operands

note that the operator is f, which is the name of the parameter we just passed in

note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)

what does it mean to evaluate a lambda?

we just assigned f to point to the function h, so we pass in the function h as y

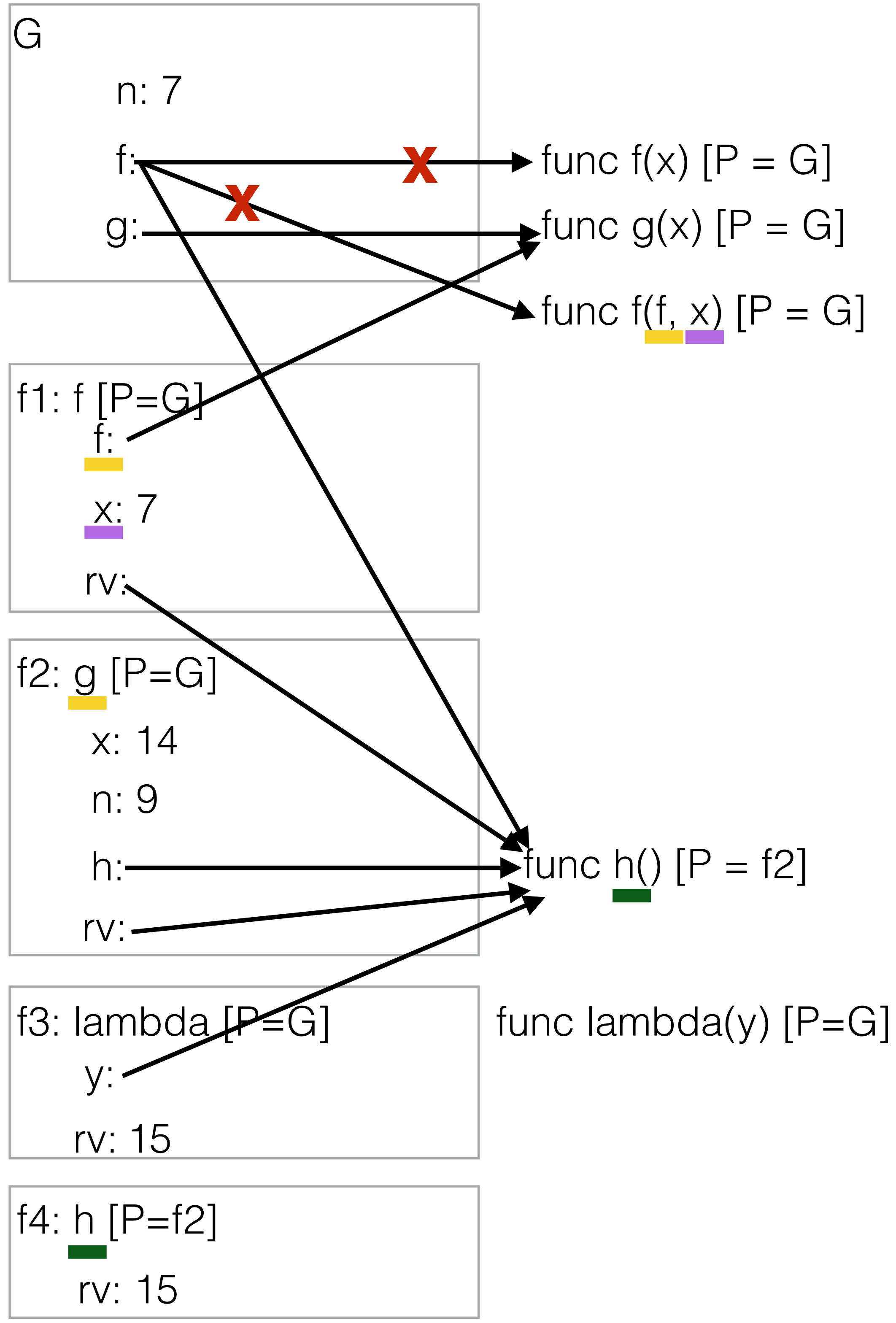
Checkpoint: why is lambda's parent G?

we have a function call, leave a mark where to return to once you complete f3

what does this lambda do? whats the body of the lambda? try converting the lambda to a normal def statement

we have another function call, leave a mark where to return to once you complete f4

y is really h, and h just adds x and 1 and returns the sum. we can find x in f2 since that is the parent of h



remember to return back to where we left marks. the last thing to do is assign the value of the function call to the name g in G

```
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) f2

f = f(g, n)  f1

g = (lambda y: y())(f)
                f4    f3
```

assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame

function call!

evaluate f
look for what the name g is bound to in the current frame
evaluate g
look for what the name g is bound to in the current frame
evaluate n
look for what the name n is bound to in the current frame
DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS

Leave an indication that this is where in the code you will come back to after completing the function call

After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just evaluated
whatever this function evaluates to is what we return in f1. leave an indication that this is where you will return to after completing f2
evaluate operator and operands

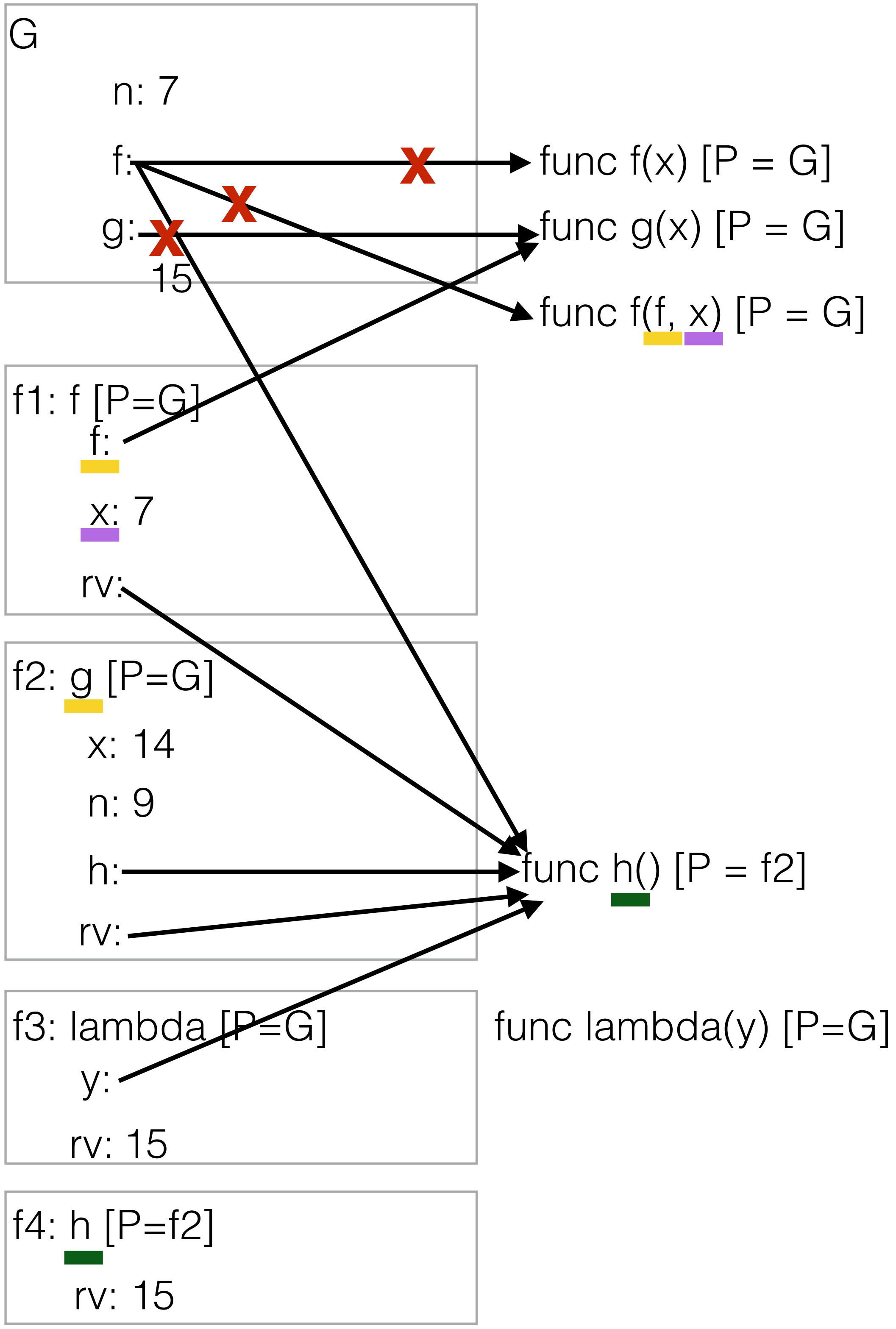
note that the operator is f, which is the name of the parameter we just passed in
note that x is also just passed in and we must look up n in G

Checkpoint: why is the parent of h, f2?

what type of thing do we return? (hint: what the difference between h and h()?)
what does it mean to evaluate a lambda?
we just assigned f to point to the function h, so we pass in the function h as y
Checkpoint: why is lambda's parent G?
we have a function call, leave a mark where to return to once you complete f3
what does this lambda do? whats the body of the lambda? try converting the lambda to a normal def statement

we have another function call, leave a mark where to return to once you complete f4

y is really h, and h just adds x and 1 and returns the sum. we can find x in f2 since that is the parent of h



G
n: 7
f:
g:
15
func f(x) [P = G]
func g(x) [P = G]
func f(f, x) [P = G]

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

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

f3: lambda [P=G]
y:
rv: 15
func lambda(y) [P=G]

f4: h [P=f2]
rv: 15

remember to return back to where we left marks. the last thing to do is assign the value of the function call to the name g in G

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that’s a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):

    print n

    if n == 1:

        return 1

    num_odds = count_down(n-1)

    return num_odds + n % 2
```

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that’s a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):

   print n

   if n == 1:

      return 1

   num_odds = count_down(n-1)

   return num_odds + n % 2
```

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):

    print n

    if n == 1:

        return 1

    num_odds = count_down(n-1)

    return num_odds + n % 2
```

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
   print n

   if n == 1:

      return 1

   num_odds = count_down(n-1)

   return num_odds + n % 2
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
    print n
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

```
    if n == 1:

        return 1
```

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

```
    num_odds = count_down(n-1)
```

How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

```
    return num_odds + n % 2
```

Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
    print n
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

```
    if n == 1:

        return 1
```

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

```
    num_odds = count_down(n-1)
```

How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

```
    return num_odds + n % 2
```

Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
    print n
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

```
    if n == 1:

        return 1
```

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

```
    num_odds = count_down(n-1)
```

How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

```
    return num_odds + n % 2
```

Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.

```
n = 3
print 3

num_odds = count_down(2)
```

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
    print n
    if n == 1:

        return 1
    num_odds = count_down(n-1)
    return num_odds + n % 2
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

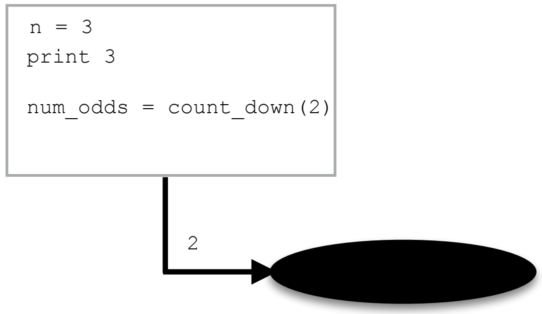
For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.



```
n = 3
print 3

num_odds = count_down(2)
```

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
    print n
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

```
    if n == 1:

        return 1
```

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

```
    num_odds = count_down(n-1)
```

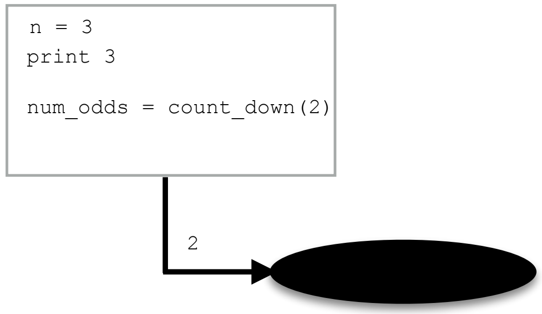
How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

```
    return num_odds + n % 2
```

Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.



```
n = 3
print 3

num_odds = count_down(2)
```

Again, I execute the lines of code that come before the recursive call.

```
n = 2
print 2

num_odds = count_down(1)
```

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
  print n
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

```
  if n == 1:

     return 1
```

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

```
  num_odds = count_down(n-1)
```

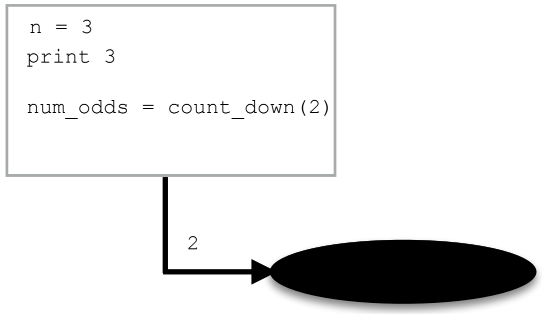
How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

```
  return num_odds + n % 2
```

Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

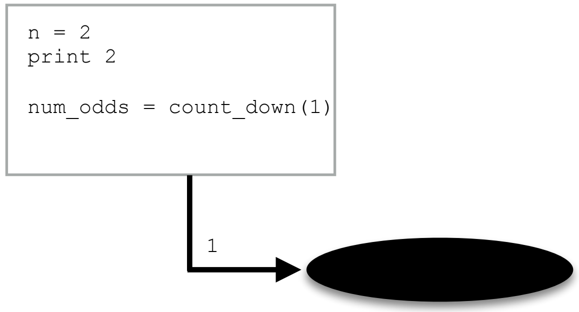
Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.



```
n = 3
print 3

num_odds = count_down(2)
```

Again, I execute the lines of code that come before the recursive call.



```
n = 2
print 2

num_odds = count_down(1)
```

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
  print n
  if n == 1:

     return 1
  num_odds = count_down(n-1)
  return num_odds + n % 2
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

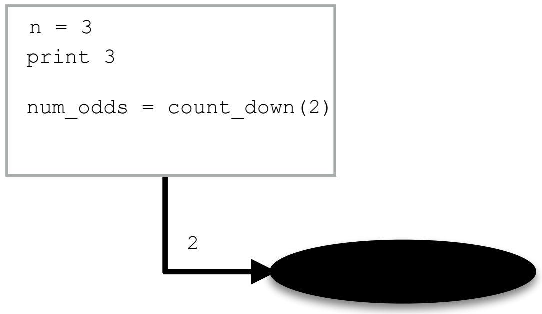
For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

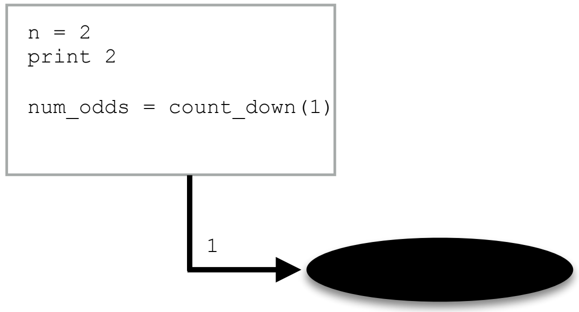
Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.



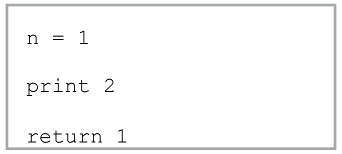
```
n = 3
print 3

num_odds = count_down(2)
```

Again, I execute the lines of code that come before the recursive call.



```
n = 2
print 2

num_odds = count_down(1)
```

Since n is 1, i go into the if statement and return 1
This is how i hop back! Now the only thing i take with me is the 1 that was returned.



```
n = 1

print 2

return 1
```

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
    print n
    if n == 1:

       return 1
    num_odds = count_down(n-1)
    return num_odds + n % 2
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

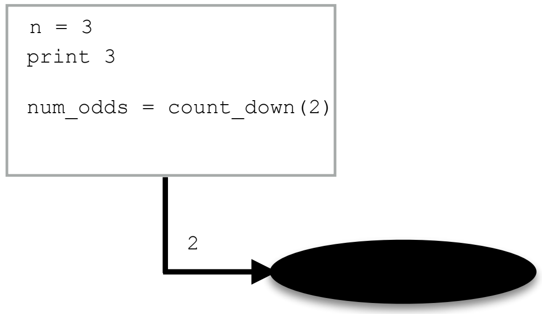
For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

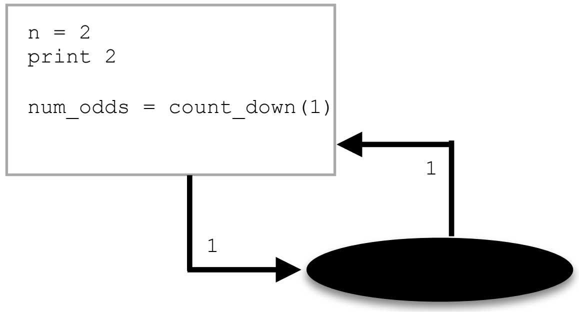
Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.


```
n = 3
print 3

num_odds = count_down(2)
```

Again, I execute the lines of code that come before the recursive call.


```
n = 2
print 2

num_odds = count_down(1)
```

Since n is 1, i go into the if statement and return 1
This is how i hop back! Now the only thing i take with me is the 1 that was returned.

```
n = 1

print 2

return 1
```

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
  print n
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

```
  if n == 1:

     return 1
```

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

```
  num_odds = count_down(n-1)
```

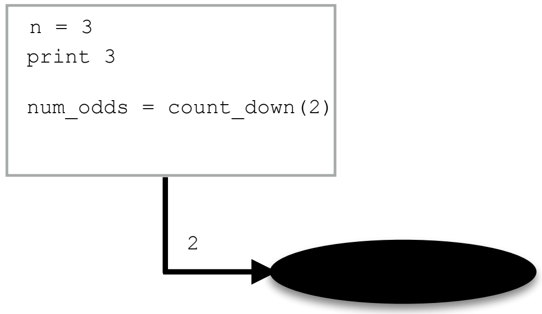
How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

```
  return num_odds + n % 2
```

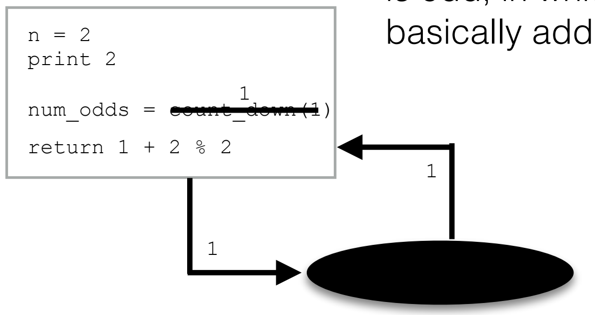
Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.


```
n = 3
print 3

num_odds = count_down(2)
```

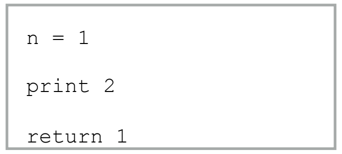
Again, I execute the lines of code that come before the recursive call.


n = 2
print 2

num_odds = ~~count_down(1)~~ 1

return 1 + 2 % 2

Since n is 1, i go into the if statement and return 1
This is how i hop back! Now the only thing i take with me is the 1 that was returned.


```
n = 1

print 2

return 1
```

After you hop back, execute the code after the recursive call.
So we know that count_down(1) returns 1, so we assign num_odds to 1.
Then we return 1 summed with 2 % 2 which is 0.
Note that we have a little trick here. We will only add 1 to num_odds if n is odd, in which case n % 2 is 1. If n % 2 is 0, then n is even and we basically add 0 to num_odds.

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
  print n
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

```
  if n == 1:

     return 1
```

For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

```
  num_odds = count_down(n-1)
```

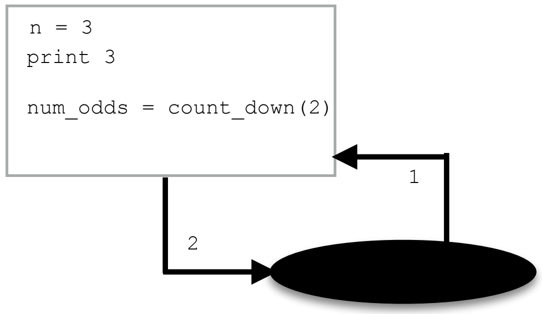
How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

```
  return num_odds + n % 2
```

Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.

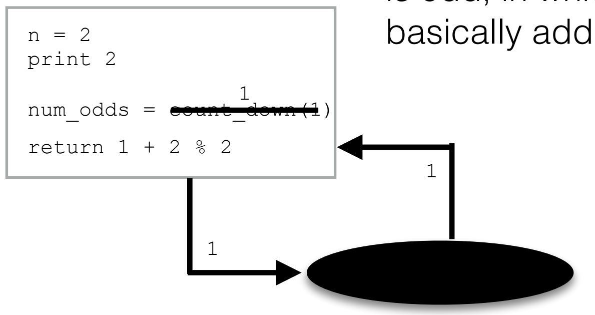


n = 3
print 3

num_odds = count_down(2)

After you hop back, execute the code after the recursive call.
So we know that count_down(1) returns 1, so we assign num_odds to 1.
Then we return 1 summed with 2 % 2 which is 0.
Note that we have a little trick here. We will only add 1 to num_odds if n is odd, in which case n % 2 is 1. If n % 2 is 0, then n is even and we basically add 0 to num_odds.

Again, I execute the lines of code that come before the recursive call.



n = 2
print 2

num_odds = ~~count_down(1)~~ 1

return 1 + 2 % 2

Since n is 1, i go into the if statement and return 1
This is how i hop back! Now the only thing i take with me is the 1 that was returned.

```
n = 1

print 2

return 1
```

We never got a chance to talk about recursion in depth in discussion, but here is how I like to think about it.

Here are the 3 main parts of recursion:

1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
3. Putting it all back together: Now is the hard part, the leap of faith! Assuming your function works correctly (which it obviously does since you wrote it) how can you use the results of the function call to answer the original question?

But that's a lot of vague hand waving talk. What does this actually look like? How can you use the above statements in a meaningful way?

Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

```
def count_down(n):
  print n
  if n == 1:

     return 1
  num_odds = count_down(n-1)
  return num_odds + n % 2
```

Now come back here. Why do we put the print statement here? Notice that we **print** before doing our recursive call. why?

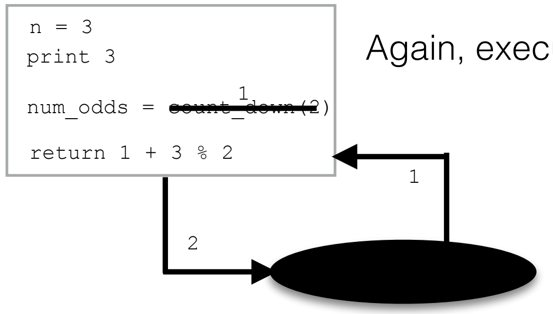
For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our **base case**.

How can we make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out the number of odd numbers from n - 1 to 1 we can just add 1 to that if we're odd, or return however many odd numbers there are from n - 1 to 1 if we're even. (**leap of faith**)

Here we are **combining the results** of the recursive call in order to answer the original problem. I explain how we do this later

Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.



n = 3
print 3

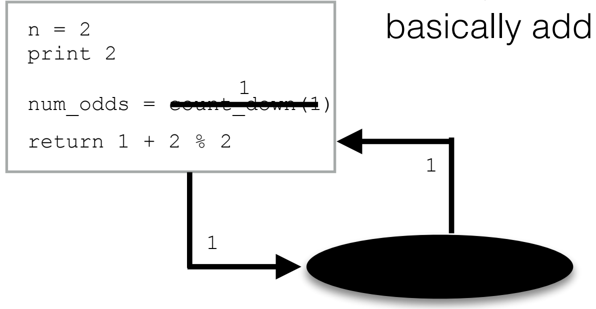
1
num_odds = ~~count_down(2~~)

return 1 + 3 % 2

Again, execute the code after the recursive call

After you hop back, execute the code after the recursive call.
So we know that count_down(1) returns 1, so we assign num_odds to 1.
Then we return 1 summed with 2 % 2 which is 0.
Note that we have a little trick here. We will only add 1 to num_odds if n is odd, in which case n % 2 is 1. If n % 2 is 0, then n is even and we basically add 0 to num_odds.

Again, I execute the lines of code that come before the recursive call.



n = 2
print 2

1
num_odds = ~~count_down(1~~)

return 1 + 2 % 2

Since n is 1, i go into the if statement and return 1
This is how i hop back! Now the only thing i take with me is the 1 that was returned.

```
n = 1

print 2

return 1
```