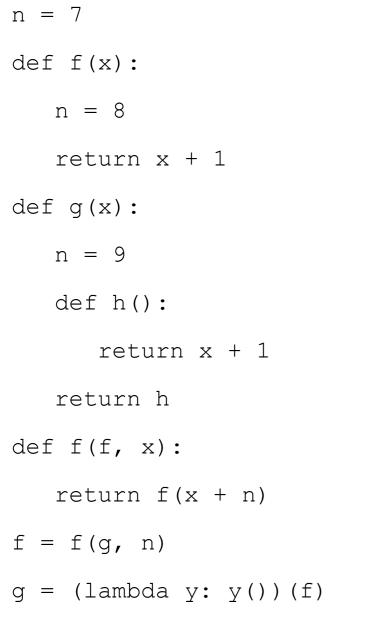
Discussion 02

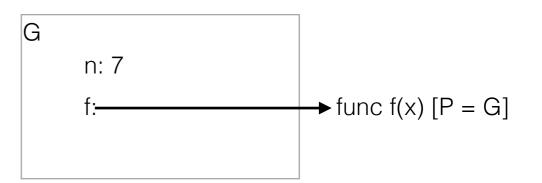
Environment Diagrams and Recursion

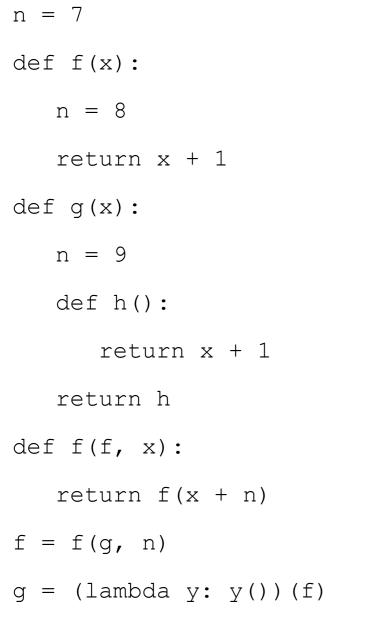
```
n = 7
def f(x):
   n = 8
   return x + 1
def g(x):
   n = 9
   def h():
      return x + 1
   return h
def f(f, x):
   return f(x + n)
f = f(g, n)
g = (lambda y: y())(f)
```

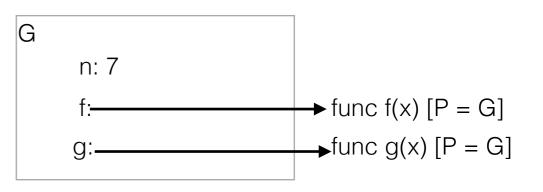
n = 7
def f(x):
n = 8
return x + 1
def g(x):
n = 9
def h():
return x + 1
return h
<pre>def f(f, x):</pre>
return f(x + n)
f = f(g, n)
g = (lambda y: y())(f)

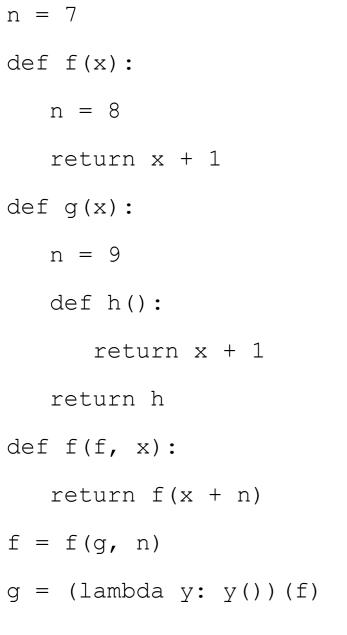


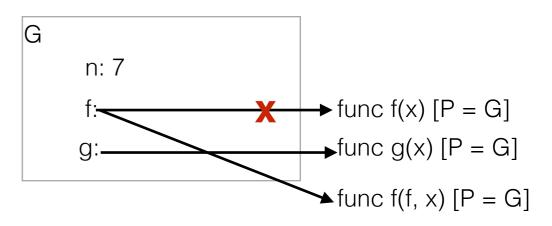


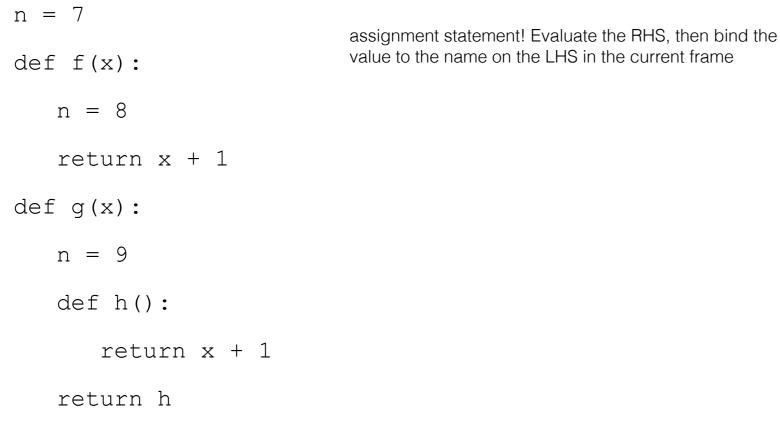


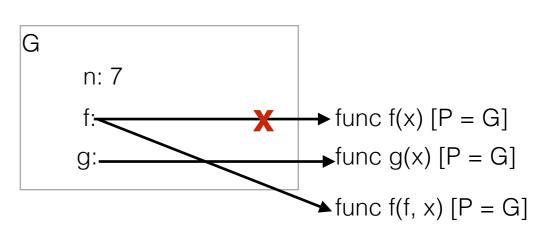












def f(f, x):

return f(x + n)

 $\blacktriangleright f = f(g, n)$

n = 7			
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!	G	
n = 8		n: 7 f: X	→ func f(x) [P = G]
return x + 1		g:	→func $g(x) [P = G]$
def g(x):			func f(f, x) [P = G]
n = 9			
def h():			
return x + 1			
return h			
def f(f, x):			
return f(x + n)			
f = f(g, n)			

n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame
n = 9	evaluate n look for what the name n is bound to in the current frame
def h():	
return x + 1	
return h	
def f(f, x):	
return f(x + n)	
f = f(g, n)	

G

n: 7

f:**~**

g:__

 $\rightarrow \text{func } f(x) [P = G]$

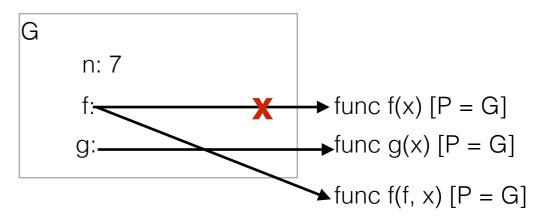
→func g(x) [P = G]

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

X

n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame
n = 9	evaluate n look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	
return h	
def f(f, x):	
return f(x + n)	

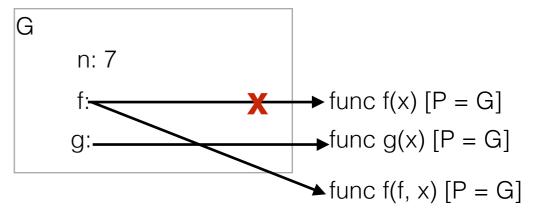
f = f(g, n)



n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	
def f(f, x):	
return f(x + n)	

f = f(g, n)

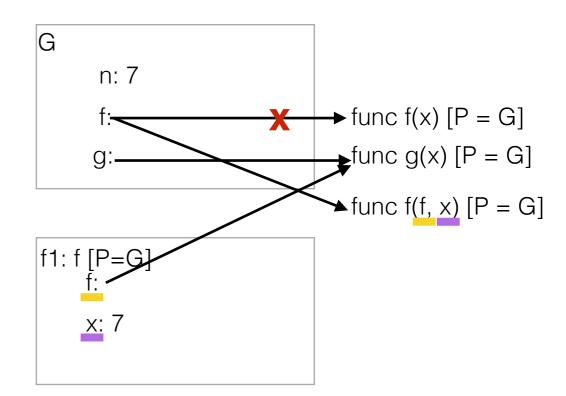
(f1)



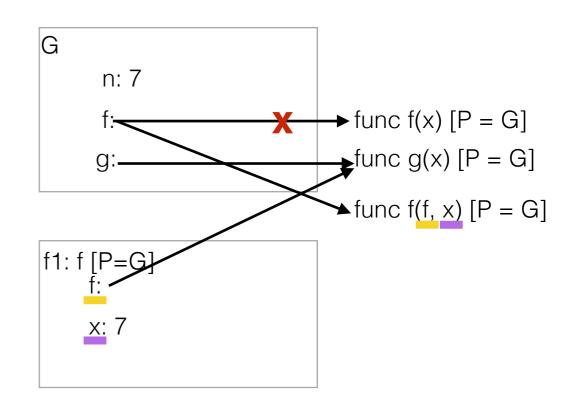
n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated
return f(x + n)	

f = f(g, n)

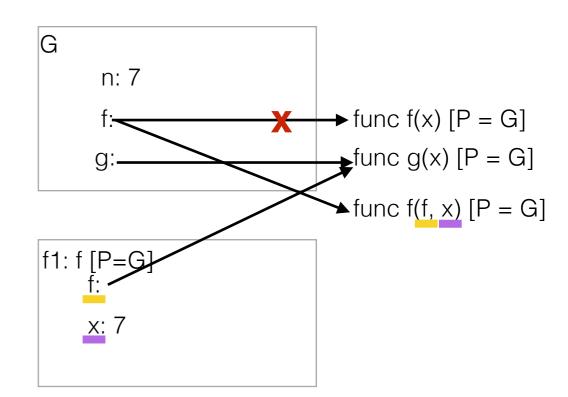
(f1)



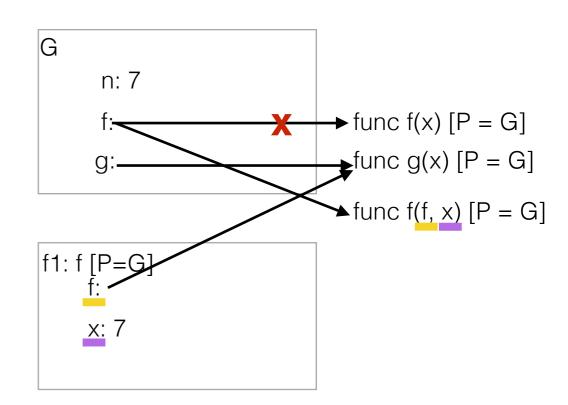
n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame
n = 9	evaluate n look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g, n)	
g = (lambda y: y())(f)	



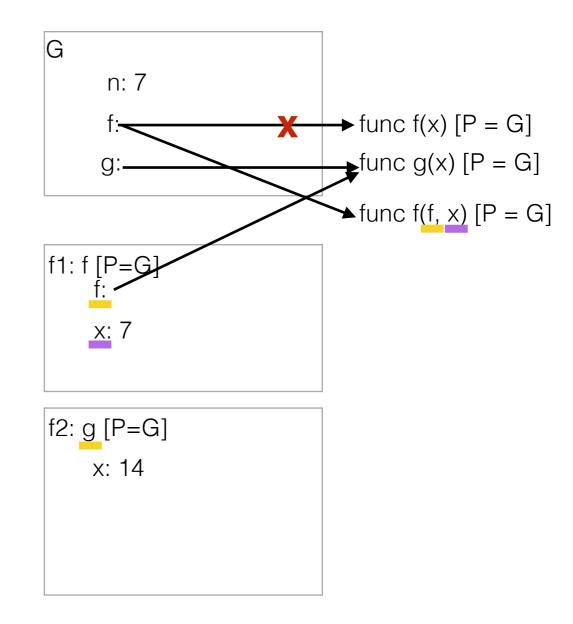
n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame
n = 9	evaluate n look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in f1_leave ap indication that this is where you will return
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g) (n) (f)	evaluate operator and operands
g = (lambda y: y())(f)	



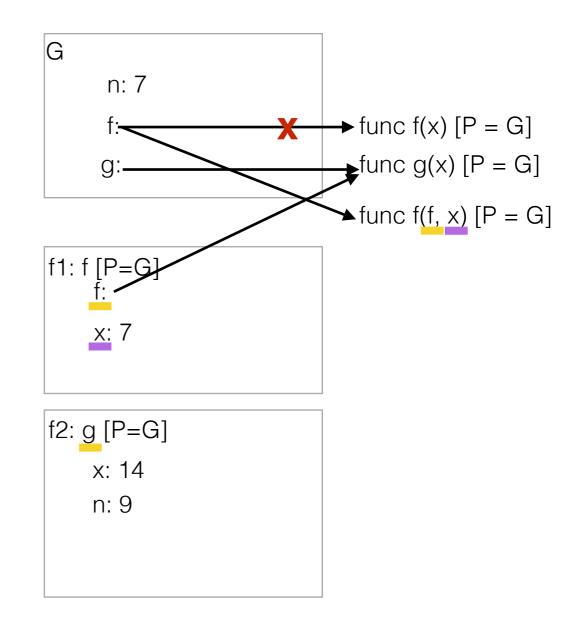
n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2 evaluate operator and operands
f = f(g, n) (f)	note that the operator is f, which is the name of the parameter we just passed in
g = (lambda y: y())(f)	note that x is also just passed in and we must look up n in G



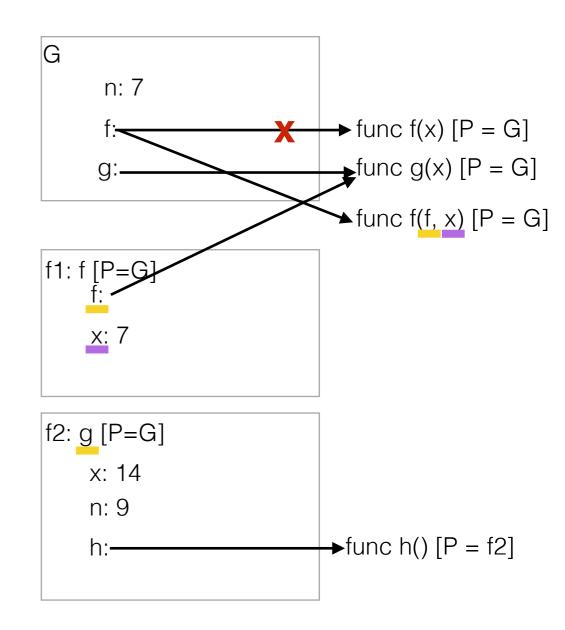
n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g; n) (f1)	evaluate operator and operands note that the operator is f, which is the name of the parameter we just passed in
g = (lambda y: y())(f)	parameter we just passed in note that x is also just passed in and we must look up n in G



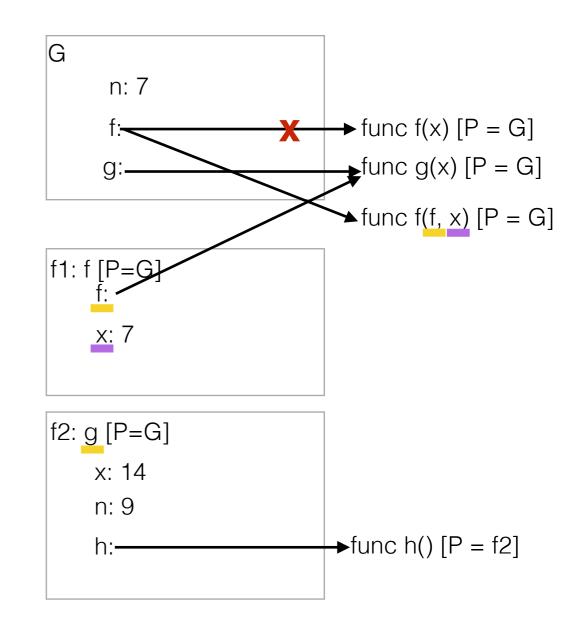
n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in
return f(x + n) f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g) (n) (f1)	evaluate operator and operands note that the operator is f, which is the name of the
	parameter we just passed in
g = (lambda y: y())(f)	note that x is also just passed in and we must look up n in G



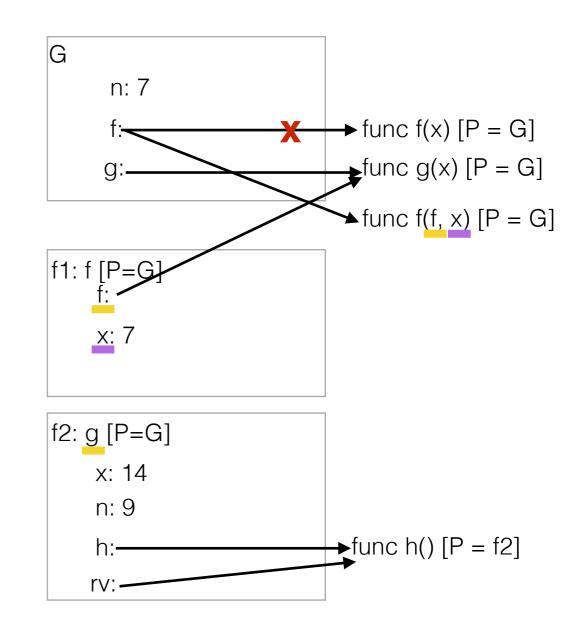
n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g; n) (f1	evaluate operator and operands note that the operator is f, which is the name of the
g = (lambda y: y())(f)	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):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame
n = 9	evaluate n look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in f1_leave an indication that this is where you will return
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g; n)	evaluate operator and operands note that the operator is f, which is the name of the
	parameter we just passed in
g = (lambda y: y())(f)	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):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame
n = 9	evaluate n look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in f1_leave an indication that this is where you will return
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g; n)	evaluate operator and operands note that the operator is f, which is the name of the
	parameter we just passed in
g = (lambda y: y())(f)	note that x is also just passed in and we must look up n in G
	Checkpoint: why is the parent of h, f2?

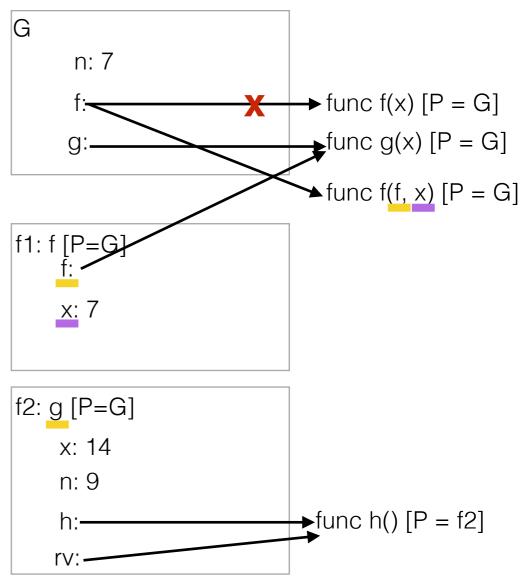


n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f
return x + 1	look for what the name g is bound to in the current frame evaluate g
def g(x):	look for what the name g is bound to in the current frame
n = 9	evaluate n look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in
return f(x + n) f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g; n) (f) g = (lambda y: y())(f)	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?

ame g is bound to in the current ame g is bound to in the current ame n is bound to in the current IEW FRAME UNTIL YOU EVALUATE ND OPERANDS that this is where in the code you fter completing the function call ew frame, bind the parameter names ature (f, x) to the values we just ion evaluates to is what we return in ion that this is where you will return f2 and operands tor is f, which is the name of the bassed in ist passed in and we must look up n

Checkpoint: why is the parent of h, f2?

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



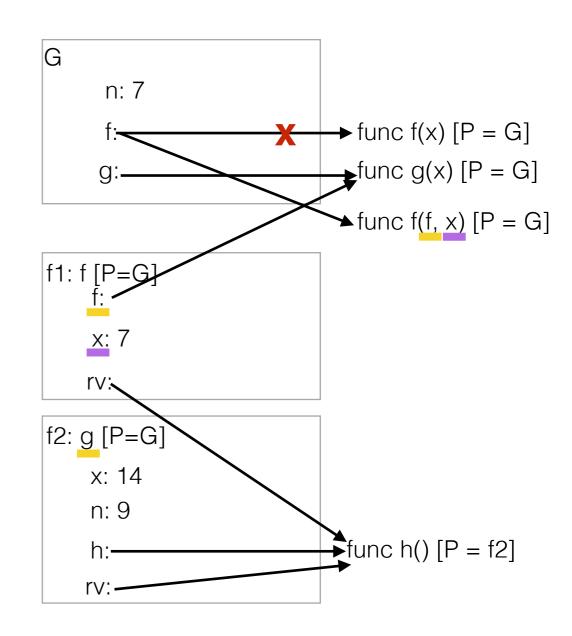
n = 7	
def f(x):	assignment state value to the nam function call!
n = 8	evaluate f look for what the
return x + 1	frame evaluate g
def g(x):	look for what the frame evaluate n
n = 9	look for what the frame
def h():	DO NOT OPEN
return x + 1	Leave an indicat will come back t
<pre>return h def f(f, x):</pre>	After opening th in the function si evaluated
return $f(x + n)$	whatever this fur f1. leave an indic to after completi
f = f(g, n) (1)	evaluate operato note that the ope parameter we ju
g = (lambda y: y())(f)	note that x is als in G

ement! Evaluate the RHS, then bind the ne on the LHS in the current frame name g is bound to in the current name g is bound to in the current name n is bound to in the current A NEW FRAME UNTIL YOU EVALUATE R AND OPERANDS tion that this is where in the code you to after completing the function call e new frame, bind the parameter names ignature (f, x) to the values we just nction evaluates to is what we return in cation that this is where you will return ing f2 or and operands erator is f, which is the name of the ist passed in

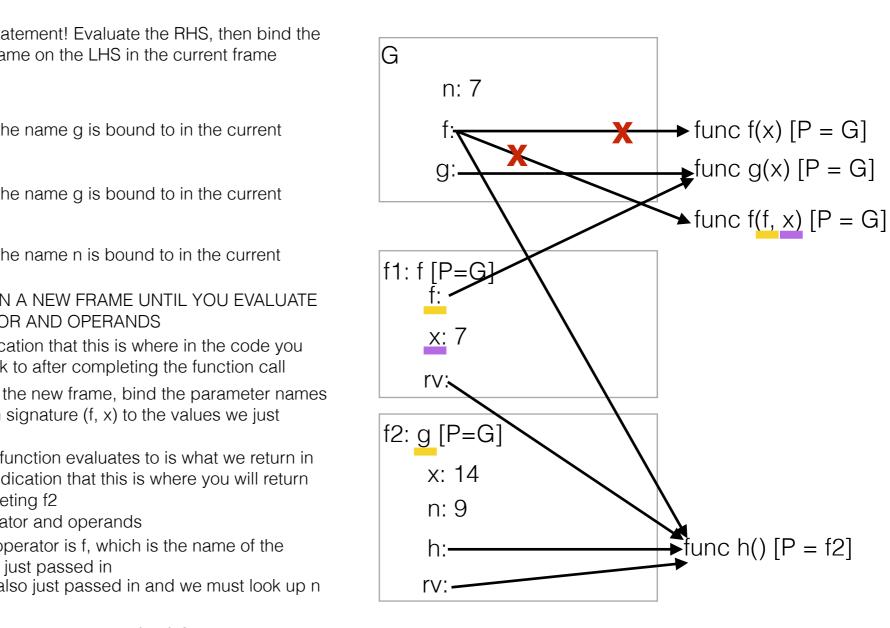
note that \boldsymbol{x} is also just passed in and we must look up \boldsymbol{n} in \boldsymbol{G}

Checkpoint: why is the parent of h, f2?

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



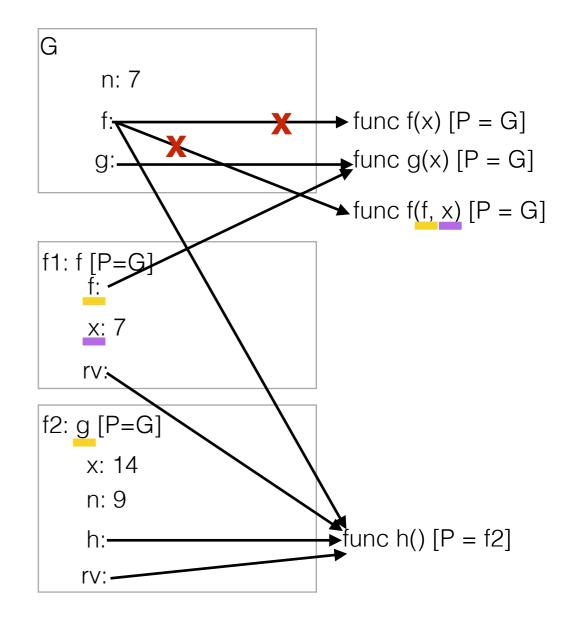
n = 7	
def f(x):	assignment statement! Evaluate the RH value to the name on the LHS in the cu function call!
n = 8	evaluate f look for what the name g is bound to ir
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in frame evaluate n
n = 9	look for what the name n is bound to in frame
def h():	DO NOT OPEN A NEW FRAME UNTIL THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in will come back to after completing the
return h	After opening the new frame, bind the in the function signature (f, x) to the val
<pre>def f(f, x):</pre>	evaluated whatever this function evaluates to is w
return $f(x + n)$ f2	f1. leave an indication that this is where to after completing f2
f = f(g, n) (f)	evaluate operator and operands note that the operator is f, which is the
g = (lambda y: y())(f)	parameter we just passed in note that x is also just passed in and w 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):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame
n = 9	evaluate n look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
<pre>def f(f, x):</pre>	evaluated whatever this function evaluates to is what we return in f1_leave ap indication that this is where you will return
return f(x + n) f2	f1. leave an indication that this is where you will return to after completing f2
f = f(q; n)	evaluate operator and operands
f = f(g; n)	note that the operator is f, which is the name of the parameter we just passed in
g = (lambda y: y())(f)	note that x is also just passed in and we must look up n in G
	Checkpoint: why is the parent of h, f2?

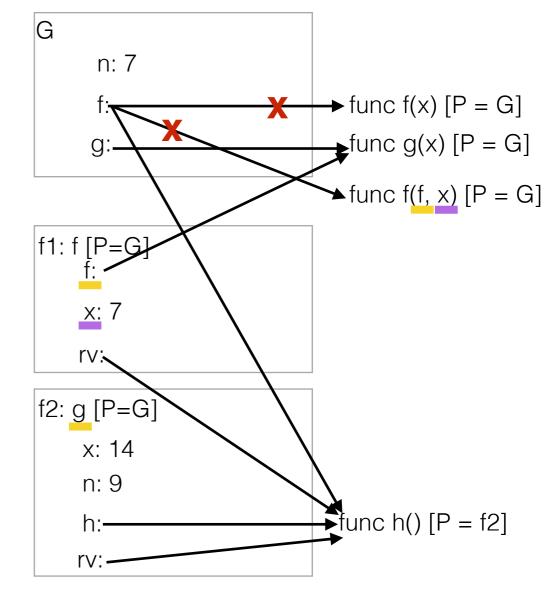
what type of thing do we return? (hint: what the difference between h and h()?)



n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame
n = 9	evaluate n look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
def f(f, x):	evaluated whatever this function evaluates to is what we return in
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g, n) (1)	evaluate operator and operands note that the operator is f, which is the name of the
g = (lambda y: y())(f)	parameter we just passed in note that x is also just passed in and we must look up n in G
	Chackpoint: why is the parent of h f22

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):	assignm value to
	function
n = 8	evaluate look for
return x + 1	frame evaluate
def g(x):	look for frame evaluate
n = 9	look for frame
def h():	DO NOT THE OP
return x + 1	Leave a will com
return h	After op in the fu
<pre>def f(f, x):</pre>	evaluate whateve f1. leave
return $f(x + n)$ f2	to after of evaluate
f = f(g, n) (f1)	note that paramet
g = (lambda y: y())(f)	note that in G
	Chookny

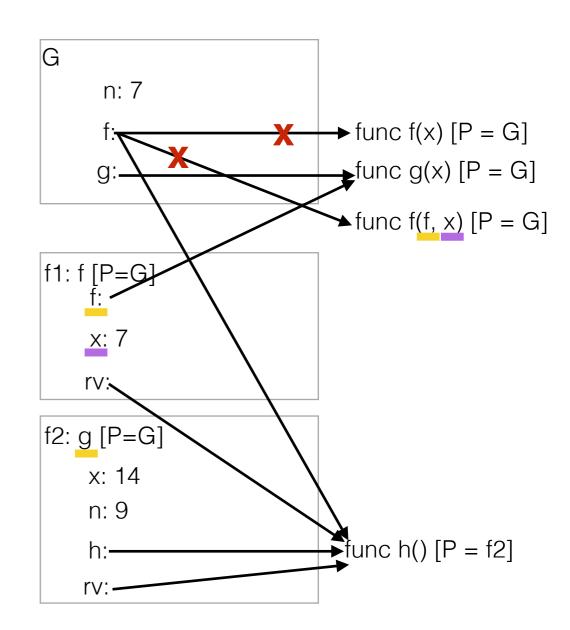
ent statement! Evaluate the RHS, then bind the the name on the LHS in the current frame call! f what the name g is bound to in the current g what the name g is bound to in the current n what the name n is bound to in the current OPEN A NEW FRAME UNTIL YOU EVALUATE ERATOR AND OPERANDS n indication that this is where in the code you e back to after completing the function call ening the new frame, bind the parameter names nction signature (f, x) to the values we just d er this function evaluates to is what we return in an indication that this is where you will return completing f2 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):	assign value t
n = 8	functio evalua
return x + 1	look fo frame evalua
def g(x):	look fo frame
n = 9	evalua look fo
def h():	frame DO NC THE O
return x + 1	Leave will cor
return h	After o in the f
<pre>def f(f, x):</pre>	evalua whatev
return $f(x + n)$ f2	f1. leav to after
f = f(g; n) f1 $g = (lambda y: y())(f)$	evalua note th param note th in G

ment statement! Evaluate the RHS, then bind the to the name on the LHS in the current frame on call! ate f or what the name g is bound to in the current ite g or what the name g is bound to in the current ite n or what the name n is bound to in the current OT OPEN A NEW FRAME UNTIL YOU EVALUATE PERATOR AND OPERANDS an indication that this is where in the code you me back to after completing the function call pening the new frame, bind the parameter names function signature (f, x) to the values we just ted ver this function evaluates to is what we return in ve an indication that this is where you will return r completing f2 ate operator and operands hat the operator is f, which is the name of the

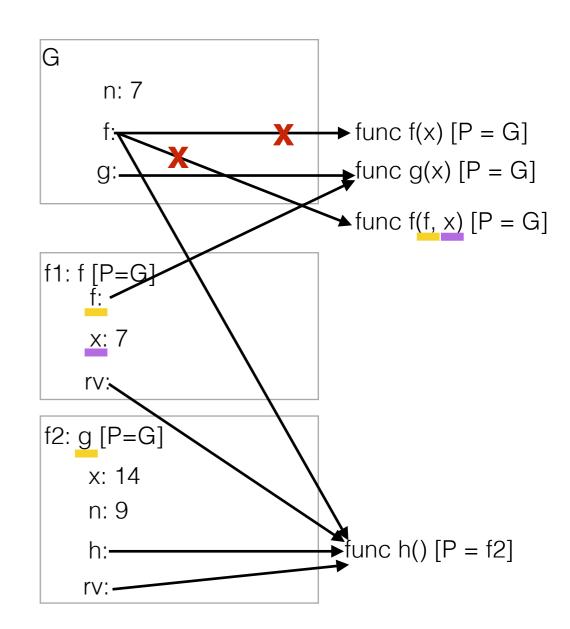
parameter we just passed in

note that \boldsymbol{x} is also just passed in and we must look up \boldsymbol{n} in \boldsymbol{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):	assigr value functio
n = 8	evalua look fo
return x + 1	frame evalua
def g(x):	look fo frame evalua
n = 9	look fo frame
def h(): return x + 1	DO N THE C Leave
return h	will co After o
<pre>def f(f, x):</pre>	in the evalua whate f1. lea
return $f(x + n)$ f2 f = f(g, n) f1	to afte evalua
g = (lambda y: y()) (f)	note ti paran note ti in G

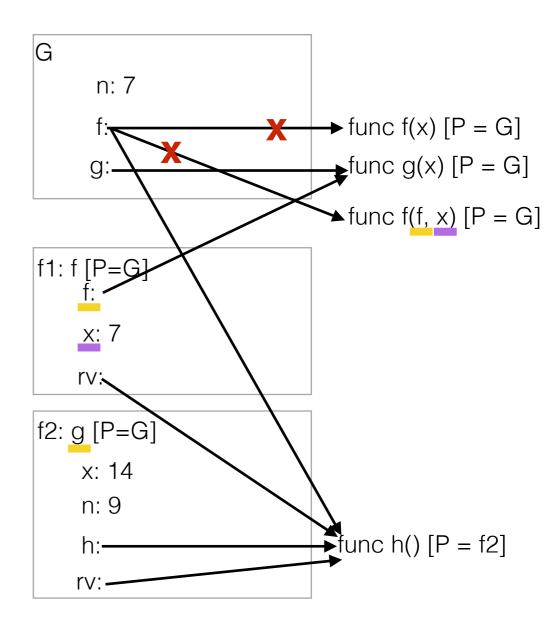
nment statement! Evaluate the RHS, then bind the to the name on the LHS in the current frame ion call! ate f or what the name g is bound to in the current ate g or what the name g is bound to in the current ate n or what the name n is bound to in the current IOT OPEN A NEW FRAME UNTIL YOU EVALUATE OPERATOR AND OPERANDS e an indication that this is where in the code you ome back to after completing the function call opening the new frame, bind the parameter names function signature (f, x) to the values we just ated ever this function evaluates to is what we return in ave an indication that this is where you will return er completing f2 ate 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

note that \boldsymbol{x} is also just passed in and we must look up \boldsymbol{n} in \boldsymbol{G}

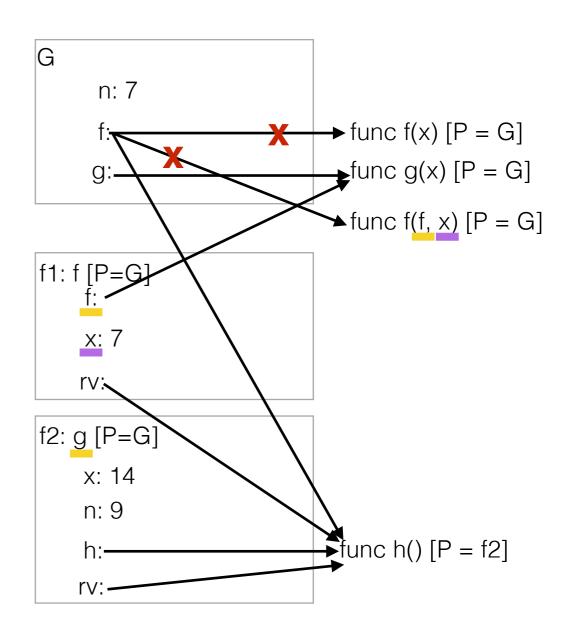
Checkpoint: why is the parent of h, f2?

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 yCheckpoint: why is lambda's parent G?



n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
<pre>def f(f, x):</pre>	evaluated whatever this function evaluates to is what we return in f1 loove on indication that this is where you will return
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g; n) (f1)	evaluate operator and operands note that the operator is f, which is the name of the
g = (lambda y: y())(f)	parameter we just passed in note that x is also just passed in and we must look up n in G
f 3	Checkpoint: why is the parent of h, f2?
	what type of thing do we return? (hint: what the

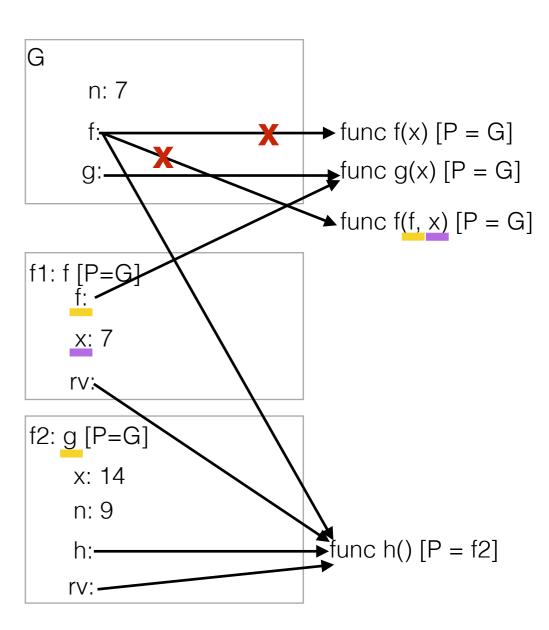
type of thing do we return? (hint: what the difference between h and h()?) what does it mean to evaluate a lambda? we just assigned f to point to the function h, so we pass in the function h as y Checkpoint: why is lambda's parent G?



n = 7	
def f(x):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
<pre>def f(f, x):</pre>	evaluated whatever this function evaluates to is what we return in
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g, n) (f1)	evaluate operator and operands note that the operator is f, which is the name of the parameter we just passed in
g = (lambda y: y())(f)	note that x is also just passed in and we must look up n in G
f3	Checkpoint: why is the parent of h, f2?
	what type of thing do we return? (hint: what the

NOT OPEN A NEW FRAME UNTIL YOU EVALUATE **OPERATOR AND OPERANDS** ve an indication that this is where in the code you come back to after completing the function call opening the new frame, bind the parameter names e function signature (f, x) to the values we just uated tever this function evaluates to is what we return in eave an indication that this is where you will return fter completing f2 uate operator and operands that the operator is f, which is the name of the ameter we just passed in that x is also just passed in and we must look up n ckpoint: why is the parent of h, f2? t 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):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
<pre>def f(f, x):</pre>	evaluated whatever this function evaluates to is what we return in
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2
f = f(g, n) (f1)	evaluate operator and operands note that the operator is f, which is the name of the parameter we just passed in
g = (lambda y: y())(f)	note that x is also just passed in and we must look up n in G
(f3)	Checkpoint: why is the parent of h, f2?
	what type of thing do we return? (hint: what the

to the name on the LHS in the current frame ion call! late f for what the name g is bound to in the current late g for what the name g is bound to in the current late n or what the name n is bound to in the current IOT OPEN A NEW FRAME UNTIL YOU EVALUATE OPERATOR AND OPERANDS e an indication that this is where in the code you come back to after completing the function call opening the new frame, bind the parameter names function signature (f, x) to the values we just ated ever this function evaluates to is what we return in ave an indication that this is where you will return er completing f2 ate operator and operands that the operator is f, which is the name of the meter we just passed in that x is also just passed in and we must look up n kpoint: why is the parent of h, f2?

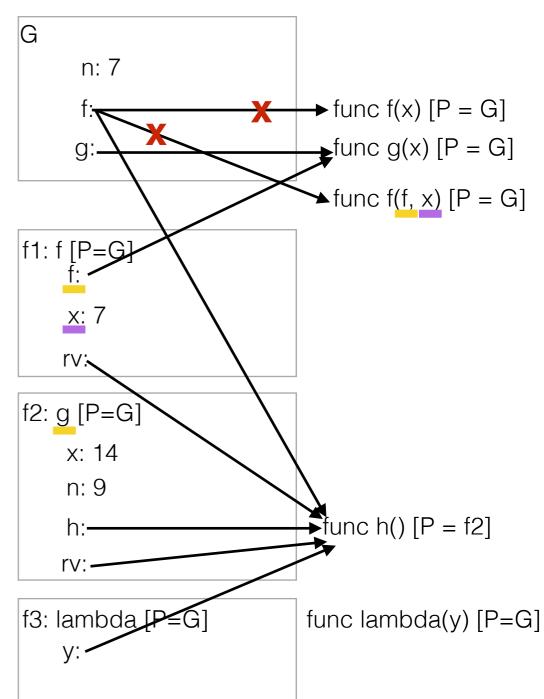
type of thing do we return? (hint: what the difference between h and h()?)

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):	assignment statement! Evaluate the RHS, then bind the value to the name on the LHS in the current frame function call!
n = 8	evaluate f look for what the name g is bound to in the current
return x + 1	frame evaluate g
def g(x):	look for what the name g is bound to in the current frame evaluate n
n = 9	look for what the name n is bound to in the current frame
def h():	DO NOT OPEN A NEW FRAME UNTIL YOU EVALUATE THE OPERATOR AND OPERANDS
return x + 1	Leave an indication that this is where in the code you will come back to after completing the function call
return h	After opening the new frame, bind the parameter names in the function signature (f, x) to the values we just
<pre>def f(f, x):</pre>	evaluated whatever this function evaluates to is what we return in
return $f(x + n)$ f2	f1. leave an indication that this is where you will return to after completing f2 evaluate operator and operands
f = f(g; n) (1)	note that the operator is f, which is the name of the parameter we just passed in
g = (lambda y: y())(f)	note that x is also just passed in and we must look up n in G
f3	Checkpoint: why is the parent of h, f2?
	what type of thing do we return? (hint: what the

e to the name on the LHS in the current frame tion call! uate f for what the name g is bound to in the current ne uate g for what the name g is bound to in the current ne uate n for what the name n is bound to in the current lе NOT OPEN A NEW FRAME UNTIL YOU EVALUATE **OPERATOR AND OPERANDS** ve an indication that this is where in the code you come back to after completing the function call opening the new frame, bind the parameter names e function signature (f, x) to the values we just uated tever this function evaluates to is what we return in eave an indication that this is where you will return fter completing f2 uate operator and operands that the operator is f, which is the name of the ameter we just passed in that x is also just passed in and we must look up n ckpoint: why is the parent of h, f2? what type of thing do we return? (hint: what the difference between h and h()?)

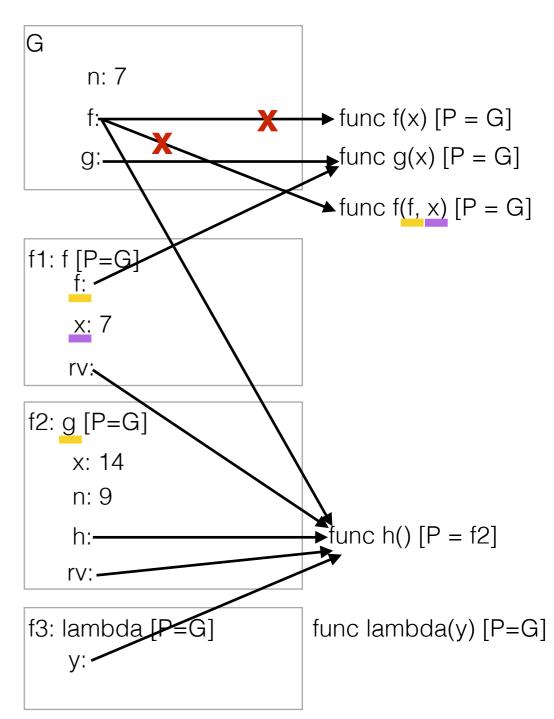
what does it mean to evaluate a lambda?

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



n = 7	
def f(x):	ass valu
	func
n = 8	eva look
return x + 1	fran
	eva
def g(x):	look fran
	eva
n = 9	look
	fran
def h():	DO
	THE
return x + 1	Lea will
return h	Afte
	in th eva
def f(f, x):	wha
	f1.
return $f(x + n)$ f2	to a
f = f(g; n) (f1)	eva note
	para
g = (lambda y: y())(f)	note
	in G
$\begin{pmatrix} f4 & f3 \end{pmatrix}$	Che

ignment statement! Evaluate the RHS, then bind the ue to the name on the LHS in the current frame ction call! luate f k for what the name g is bound to in the current ne luate g k for what the name g is bound to in the current ne luate n k for what the name n is bound to in the current ne NOT OPEN A NEW FRAME UNTIL YOU EVALUATE E OPERATOR AND OPERANDS ave an indication that this is where in the code you come back to after completing the function call er opening the new frame, bind the parameter names he function signature (f, x) to the values we just luated atever this function evaluates to is what we return in leave an indication that this is where you will return after completing f2 luate operator and operands e that the operator is f, which is the name of the ameter 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?

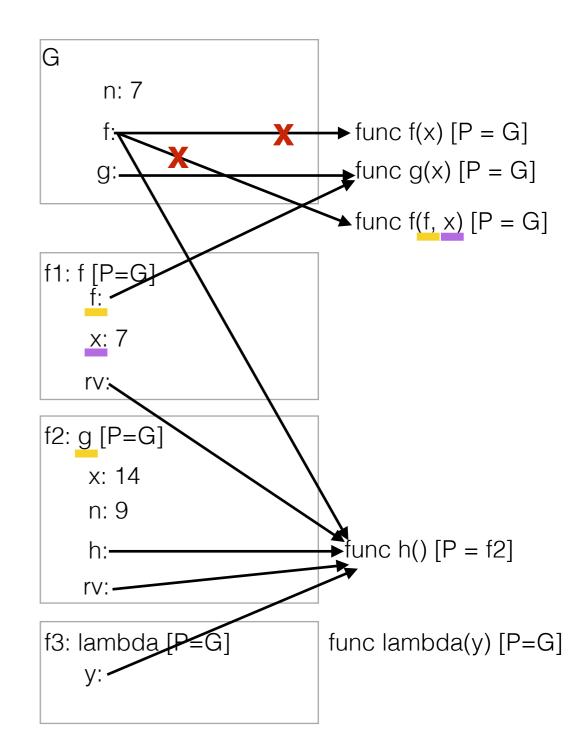
we just assigned f to point to the function h, so we pass in the function h as $\ensuremath{\mathsf{y}}$

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):	assių value
	funct
n = 8	evalı look
return x + 1	fram evalı
def g(x):	look fram
n = 9	evalı look
def h():	fram DO N
	THE
return x + 1	Leav will c
return h	After in the
<pre>def f(f, x):</pre>	evalı what
return f(x + n) f2	f1. le to af
f = f(g; n) (f)	evalı note
g = (lambda y: y())(f)	para note
f4 f3	in G Cheo
	CHE

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

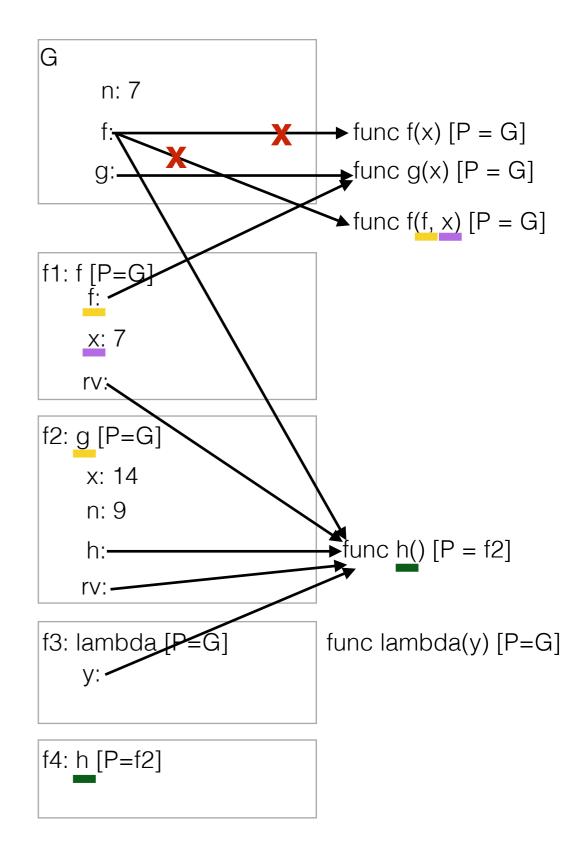
Checkpoint: why is lambda's parent G?

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):	assig value
	funct
n = 8	evalı look
return x + 1	fram evalu
def g(x):	look fram
n = 9	evalı look from
def h():	fram DO N THE
return x + 1	Leav will c
return h	After in the
def f(f, x):	evalı what
return $f(x + n)$ f2	f1. le to aff
f = f(g; n) (f1)	evalu note
g = (lambda y: y())(f)	para note in G
f4 f3	Cheo

gnment statement! Evaluate the RHS, then bind the e to the name on the LHS in the current frame tion call! uate f for what the name g is bound to in the current е uate g for what the name g is bound to in the current е uate n for what the name n is bound to in the current е NOT OPEN A NEW FRAME UNTIL YOU EVALUATE **OPERATOR AND OPERANDS** e an indication that this is where in the code you come back to after completing the function call opening the new frame, bind the parameter names e function signature (f, x) to the values we just uated ever this function evaluates to is what we return in eave an indication that this is where you will return ter completing f2 late operator and operands that the operator is f, which is the name of the meter we just passed in that x is also just passed in and we must look up n ckpoint: 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

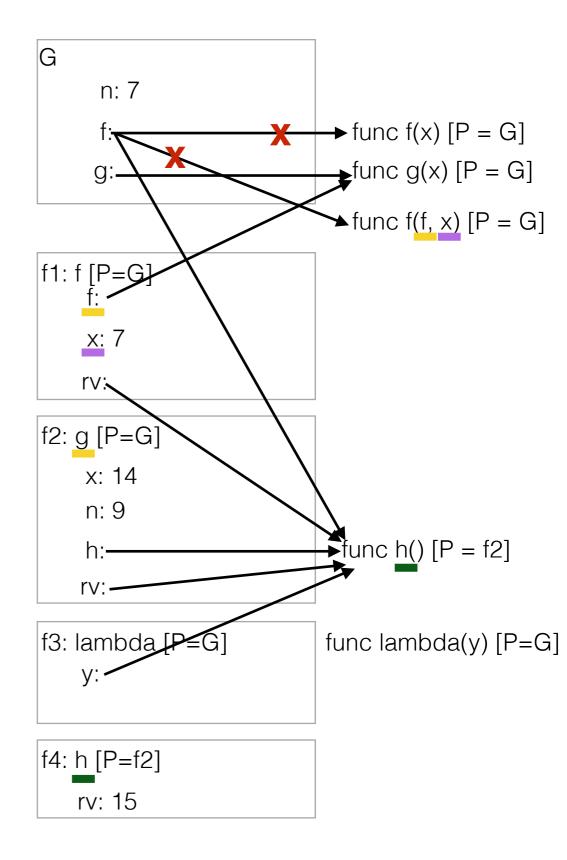
Checkpoint: why is lambda's parent G?

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):	assig value
	funct
n = 8	evalı look
return x + 1	fram evalu
def g(x):	look fram
n = 9	evalı look from
def h():	fram DO N THE
return x + 1	Leav will c
return h	After in the
def f(f, x):	evalı what
return $f(x + n)$ f2	f1. le to aff
f = f(g; n) (f1)	evalu note
g = (lambda y: y())(f)	para note in G
f4 f3	Cheo

gnment statement! Evaluate the RHS, then bind the e to the name on the LHS in the current frame tion call! uate f for what the name g is bound to in the current le uate g for what the name g is bound to in the current ne uate n for what the name n is bound to in the current lе NOT OPEN A NEW FRAME UNTIL YOU EVALUATE **OPERATOR AND OPERANDS** ve an indication that this is where in the code you come back to after completing the function call opening the new frame, bind the parameter names e function signature (f, x) to the values we just uated tever this function evaluates to is what we return in eave an indication that this is where you will return ter completing f2 uate operator and operands that the operator is f, which is the name of the meter we just passed in that x is also just passed in and we must look up n ckpoint: 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

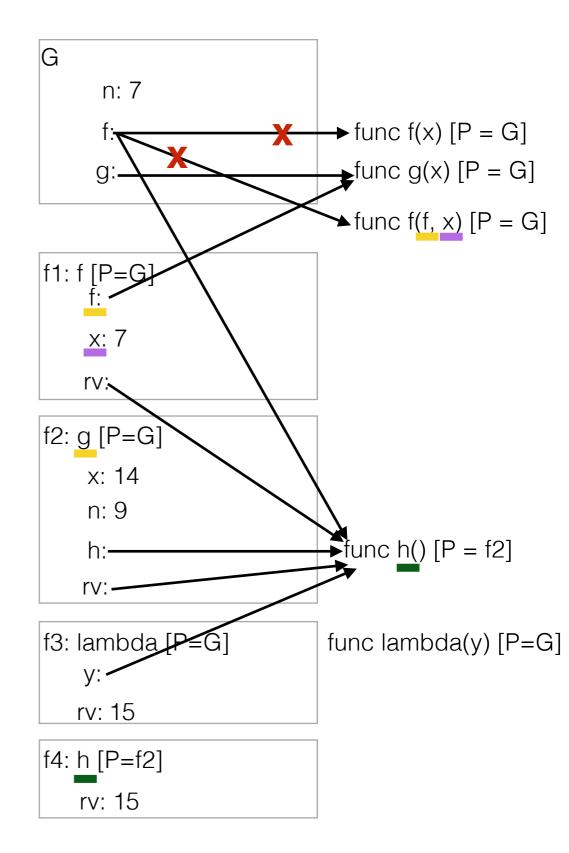
Checkpoint: why is lambda's parent G?

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):	assig value
	functi
n = 8	evalu look f
return x + 1	frame evalu
def g(x):	look f frame
n = 9	evalu look f frame
def h():	DO N THE (
return x + 1	Leave will co
return h	After in the
<pre>def f(f, x):</pre>	evalu whate
return f(x + n) (f2)	f1. lea to afte
f = f(g; n) (f)	evalu note t
g = (lambda y: y())(f)	parar note t in G
f4 f3	Chec

nment statement! Evaluate the RHS, then bind the to the name on the LHS in the current frame ion call! late f for what the name g is bound to in the current late g for what the name g is bound to in the current late n or what the name n is bound to in the current IOT OPEN A NEW FRAME UNTIL YOU EVALUATE OPERATOR AND OPERANDS e an indication that this is where in the code you ome back to after completing the function call opening the new frame, bind the parameter names function signature (f, x) to the values we just ated ever this function evaluates to is what we return in ave an indication that this is where you will return er completing f2 ate operator and operands that the operator is f, which is the name of the meter we just passed in that x is also just passed in and we must look up n kpoint: 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

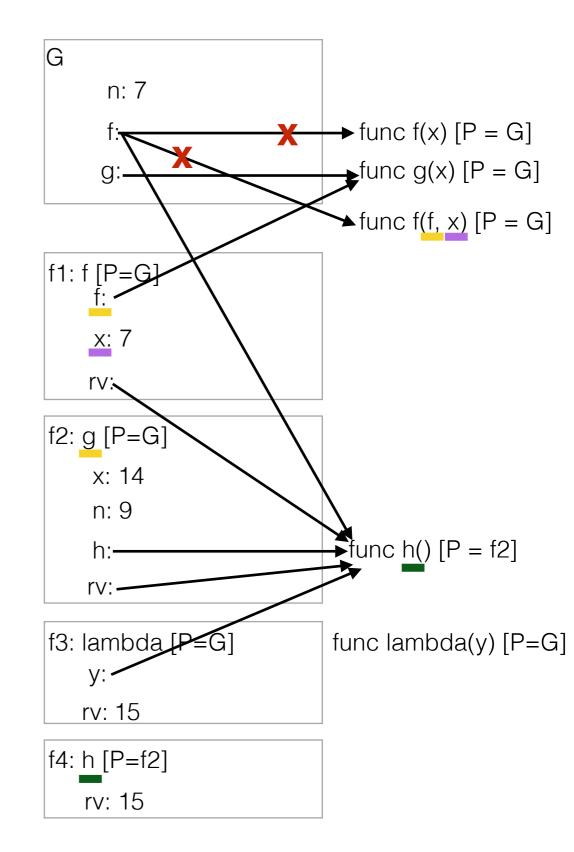
Checkpoint: why is lambda's parent G?

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):	assig value
n = 8	functi evalu
return x + 1	look f frame evalu
def g(x):	look f frame
n = 9	evalu look f frame
def h():	DO N THE (
return x + 1	Leave will co
return h	After in the
<pre>def f(f, x):</pre>	evalu whate
return $f(x + n)$ f2	f1. lea to afte evalue
f = f(g; n) (1)	note t
g = (lambda y: y())(f)	note t in G
	Chec

nment statement! Evaluate the RHS, then bind the to the name on the LHS in the current frame ion call! ate f or what the name g is bound to in the current ate g or what the name g is bound to in the current ate n or what the name n is bound to in the current IOT OPEN A NEW FRAME UNTIL YOU EVALUATE OPERATOR AND OPERANDS e an indication that this is where in the code you ome back to after completing the function call opening the new frame, bind the parameter names function signature (f, x) to the values we just ated ever this function evaluates to is what we return in ave an indication that this is where you will return er completing f2 ate operator and operands that the operator is f, which is the name of the neter we just passed in that x is also just passed in and we must look up n kpoint: 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

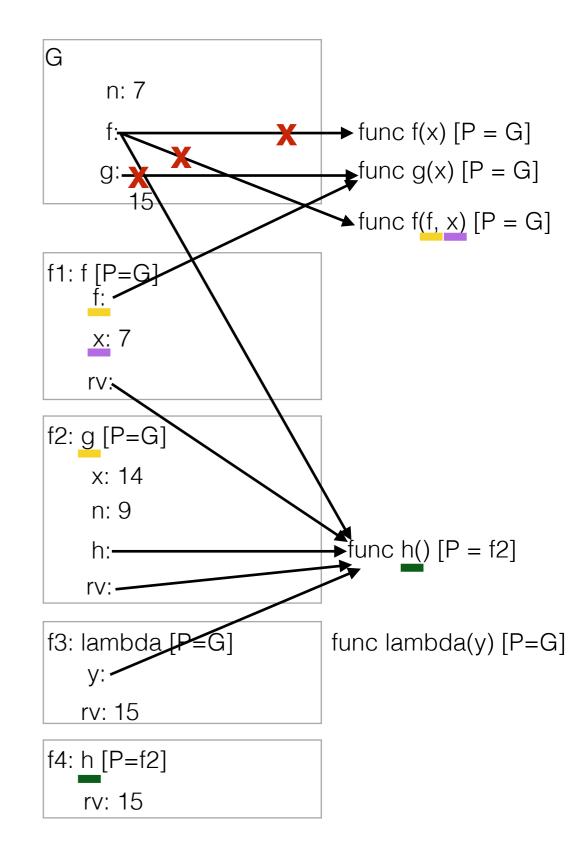
Checkpoint: why is lambda's parent G?

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

Here are the 3 main parts of recursion:

- 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?

- 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?

- 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.

- 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)

- 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

return 1

if n == 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)

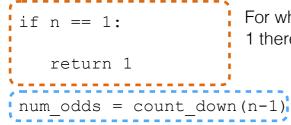
- Here are the 3 main parts of recursion: 1. Base case: what is the simplest problem you could be given? Ask yourself, for what input, do I immediately know what the answer is?
 - 2. Recursive call: how can you simplify the problem? Make sure you are calling the function on a smaller problem
 - 3. Putting 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



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 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
if n == 1: return 1	For w 1 ther
num_odds = count_dow	n(n-1)

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

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 ere 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 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	v come back here. Why do we put the print statement here? Notice that we print before doing our recursive call. why? 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 ere is just one odd number, 1! So in this case, we will return 1. We just wrote our base case .	
if n == 1:		
return 1 num_odds = count_dow	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	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 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:	rhich 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 re is just one odd number, 1! So in this case, we will return 1. We just wrote our base case .	
return 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	
num_odds = count_dow	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	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

- 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:	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 .	
	return 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	
	num_odds = count_down	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 ⁹	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)

- 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?

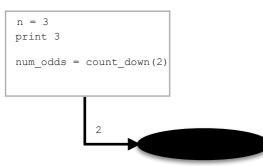
Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

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? 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 .	
if n == 1:		
return 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	
num_odds = count_down	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 ^s	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.



- 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?

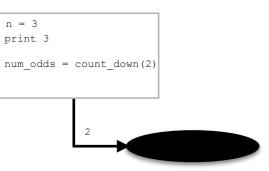
Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

def count down(n):

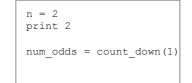
	print n	Now come back here. Why do we put the print statement here? Notice that we print before doing our recursive call. why? For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our base case .	
	if n == 1:		
	return 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	
	num_odds = count_dow	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	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.



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



- 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?

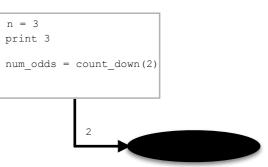
Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

def count down(n):

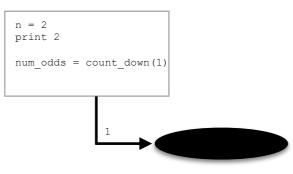
print n	Now come back here. Why do we put the print statement here? Notice that we print before doing our recursive call. why?	
if n == 1:	For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our base case .	
return 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	
num_odds = count_dow	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	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.



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



- 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?

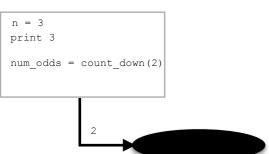
Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

def count down(n):

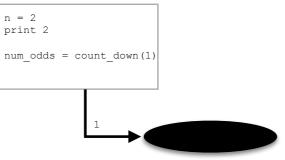
	print n	Now come back here. Why do we put the print statement here? Notice that we print before doing our recursive call. why? For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our base case .	
	if n == 1:		
	return 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	
	num_odds = count_down	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	Here we are combining the results of the recursive call in order to answer the original problem. I explain how we do this later	

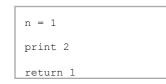
Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

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.



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





- 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?

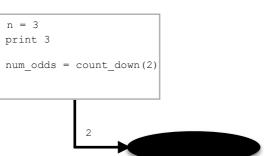
Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

def count down(n):

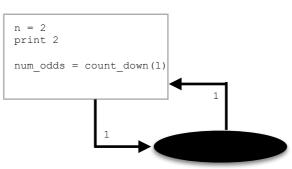
	print n	Now come back here. Why do we put the print statement here? Notice that we print before doing our recursive call. why? For which n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to 1 there is just one odd number, 1! So in this case, we will return 1. We just wrote our base case .	
	if n == 1:		
	return 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	
	num_odds = count_down	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	Here we are combining the results of the recursive call in order to answer the original problem. I explain how we do this later	

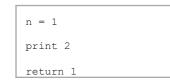
Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

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.



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





- 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):

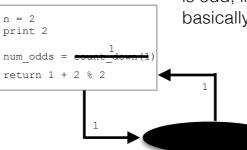
print n	Now c	ome back here. Why do we put the print statement here? Notice that we print before doing our recursive call. why?	
if n == 1:	-	h n do you know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to s just one odd number, 1! So in this case, we will return 1. We just wrote our base case .	
return 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	
num_odds = count_down	n(n-1)	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	82	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

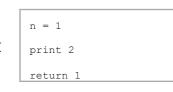
I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.



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



- 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
if n == 1:	For which n do you k 1 there is just one od
return 1	How can we
num_odds = count_dow	n(n-1) the number o there are from
return num_odds + n	8 2 Here we are a

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

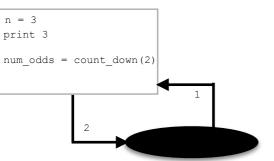
know immediately how many odd numbers there are from n to 1? If n is 1, then we know that from 1 to dd number, 1! So in this case, we will return 1. We just wrote our **base case**.

make the problem smaller? Well we know we can find out if the current n is even or odd. If we can find out 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 m n - 1 to 1 if we're even. (leap of faith)

combining the results of the recursive call in order to answer the original problem. I explain how we do this late

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

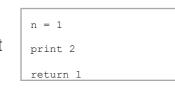
I picture each recursive call as jumping down into a hole. I can only take the parameters with me. So in this case, when I jump in the hole I take 2 with me.



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



- 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?

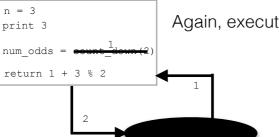
Lets do an example. Write a function that prints out all the numbers from n to 1 in decreasing order AND returns the number of odd numbers from n to 1.

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:	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 .
return 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
num_odds = count_dow	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	Here we are combining the results of the recursive call in order to answer the original problem. I explain how we do this later

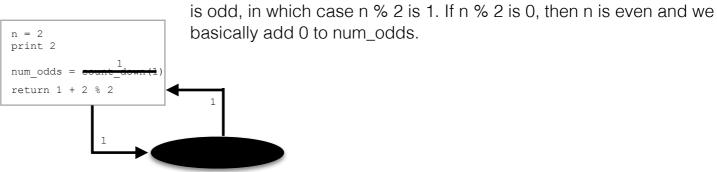
Here's what I picture happening whenever I have to do recursion. All the code that you see before the recursive call happens in the current frame. Say we try to do count_down(3). Let's walk through what happens

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.

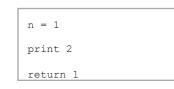


Again, execute the code after the recursive call

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



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.



After you hop back, execute the code after the recursive call.

Then we return 1 summed with 2 % 2 which is 0.

So we know that count_down(1) returns 1, so we assign num_odds to 1.

Note that we have a little trick here. We will only add 1 to num_odds if n