CS 61A DISCUSSION 2

ENVIRONMENT DIAGRAMS AND RECURSION

Raymond Chan
Discussion 134
UC Berkeley Fall 16

AGENDA

- Announcements
- Environment diagram review
- Lambda
- Recursion

ANNOUNCEMENTS

- Project 1 Hog due tonight!!
- Lab 2 due Friday
- Homework 3 due Tuesday 9/13
- Guerrilla Section on Higher Order Functions & Recursion 9/10 noon - 3pm
- CSM small group tutoring sections sign ups

MIDTERM ANNOUNCEMENTS

- Midterm 1 next Thursday 9/15 8-10pm. Rooms TBD
- Topical Office Hours next week
- TA-led review session noon - 3pm Sunday 155 Dwindle
- HKN review session 2 - 5 pm Saturday 2050 VLSB

MORE ANNOUNCEMENTS

- Based on demand, I will hold at least one of the following (or both)
    - Review session to go over past exam problems Sunday 4-6pm Soda 320 (this may change)
    - Office Hour before midterm Thursday during discussion time (no discussion next week)

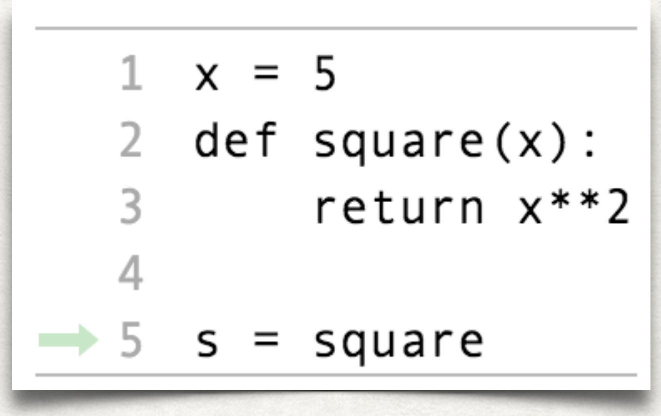
ENVIRONMENT DIAGRAMS

- Environment diagrams allow us to keep track of variables that have been defined and the values they are bound to.
- Assignment Statements
- Def Statements
- Function Calls
- Lambda Expressions

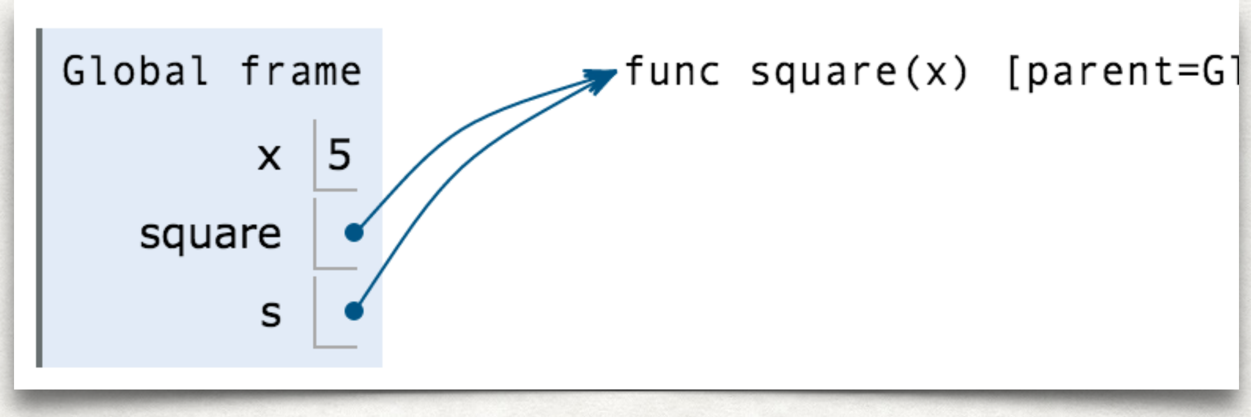
ASSIGNMENT STATEMENTS

REVIEW

- Evaluate right hand side.
    - Look up names in the current frame, and then parent frame.
    - Left hand side variable created in local frame if it does not exist.



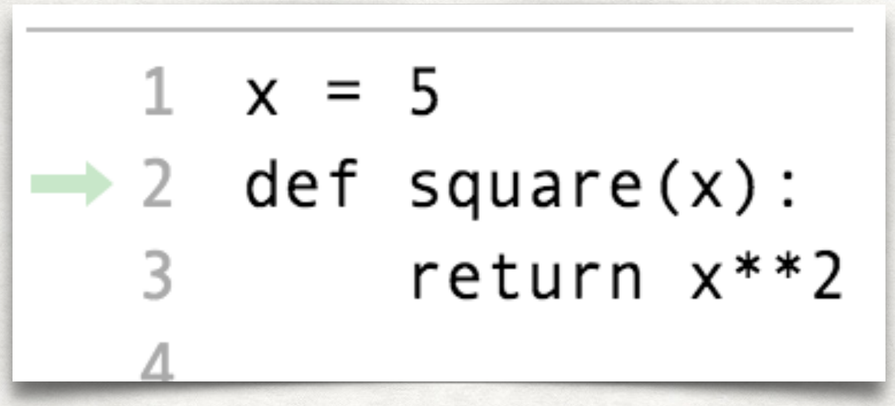
```
x = 5
def square(x):
    return x**2

s = square
```



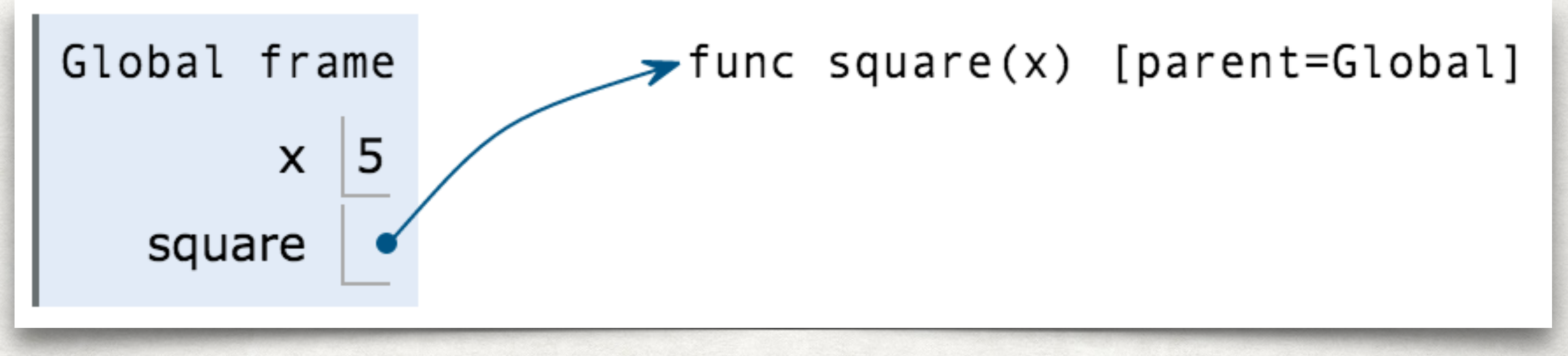
DEF STATEMENTS

REVIEW

- Function object has function signature (intrinsic name and formal parameters) and parent frame.
- The parent frame is the frame in which the frame is defined.
- Do not evaluate body.



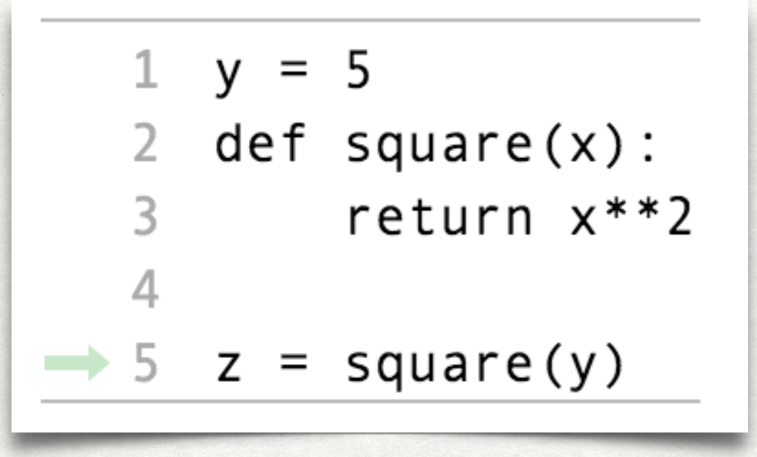
```
x = 5
def square(x):
    return x**2
```



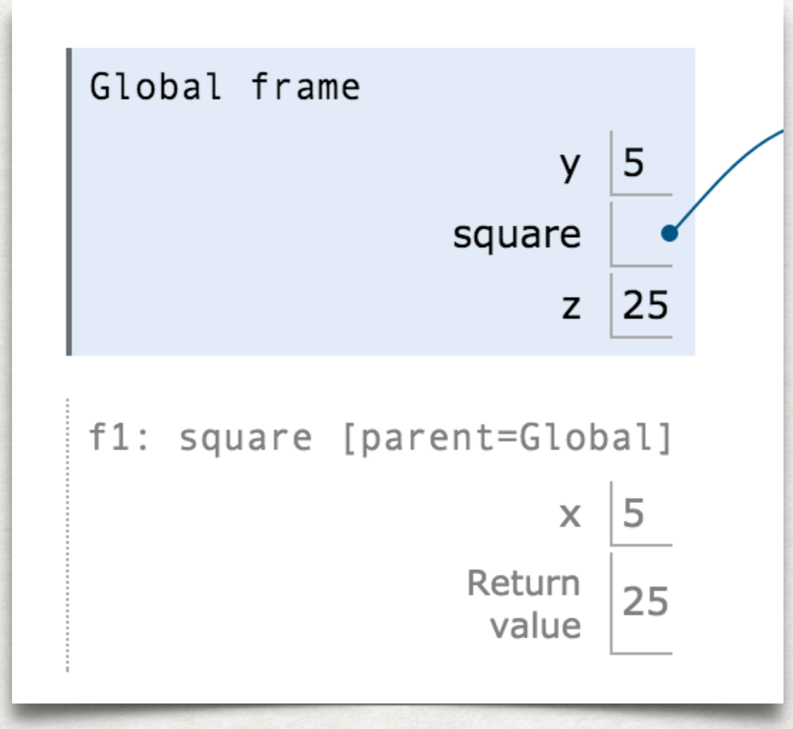
CALL EXPRESSIONS

REVIEW

- Evaluate the operator, then operands from left to right.
- Apply evaluated operands to operator and create new frame with intrinsic name.
- Bind arguments to formal parameters.



```
y = 5
def square(x):
    return x**2

z = square(y)
```



Global frame

y 5

square

z 25

f1: square [parent=Global]

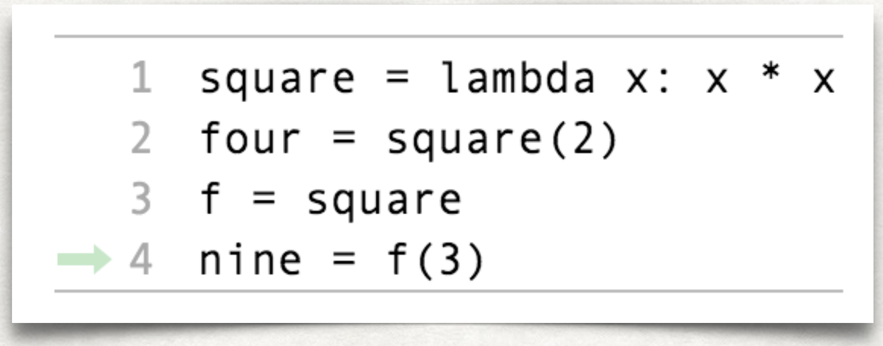
x 5

Return value 25

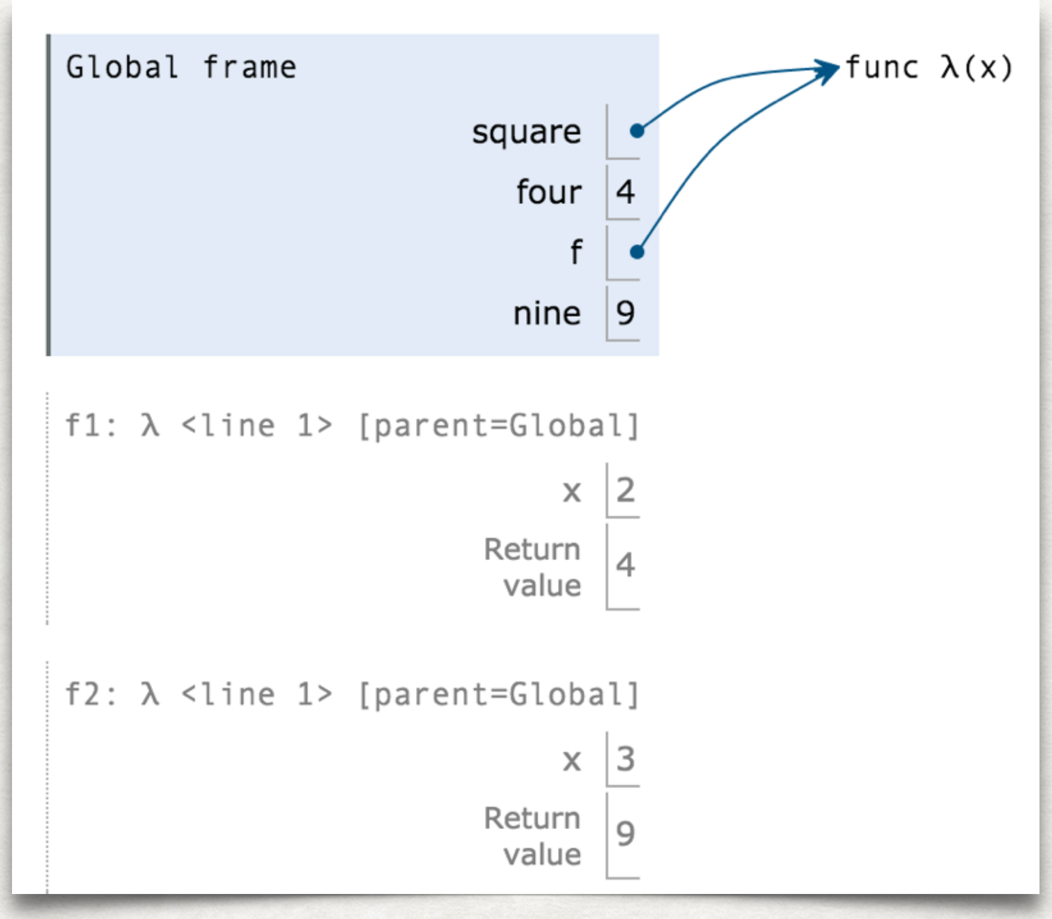
FUNCTION CALL VS. FUNCTION OBJECTS

REVIEW

- Function calls have parenthesis after variable that is bound to function object.



```
square = lambda x: x * x
four = square(2)
f = square
nine = f(3)
```



Global frame

square	●
four	4
f	●
nine	9

f1: λ <line 1> [parent=Global]

x	2
Return value	4

f2: λ <line 1> [parent=Global]

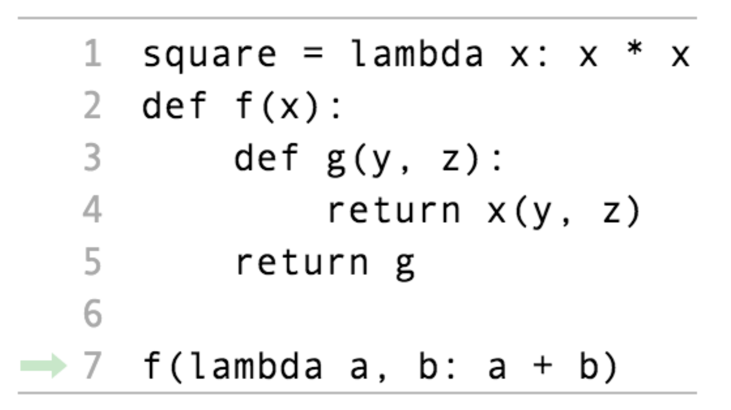
x	3
Return value	9

LAMBDA FUNCTIONS

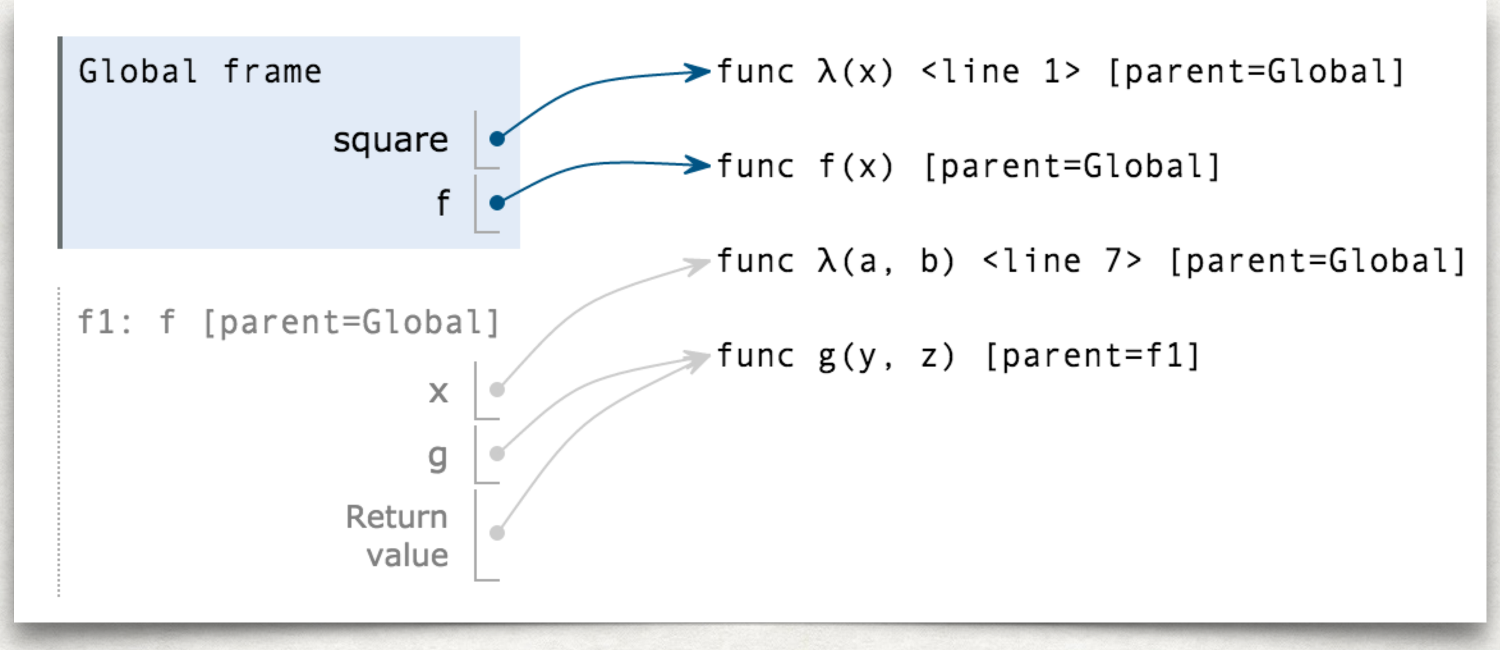
- lambda <parameters>: <body>
- There can be multiple parameters delimited by commas.
  - lambda x, y, z: <body>
- Lambda functions create function objects with the function name as **λ**.
- Create the function object in the environment diagram even if it is not assigned to a variable.

LAMBDA FUNCTIONS

- Lambda functions cannot be accessed if it is not assigned to variables either by
  - using an explicit assignment statement or
  - passing the lambda function into another function's argument.



```
square = lambda x: x * x
def f(x):
    def g(y, z):
        return x(y, z)
    return g

f(lambda a, b: a + b)
```



Global frame

square

f

func λ(x) <line 1> [parent=Global]

func f(x) [parent=Global]

func λ(a, b) <line 7> [parent=Global]

f1: f [parent=Global]

x

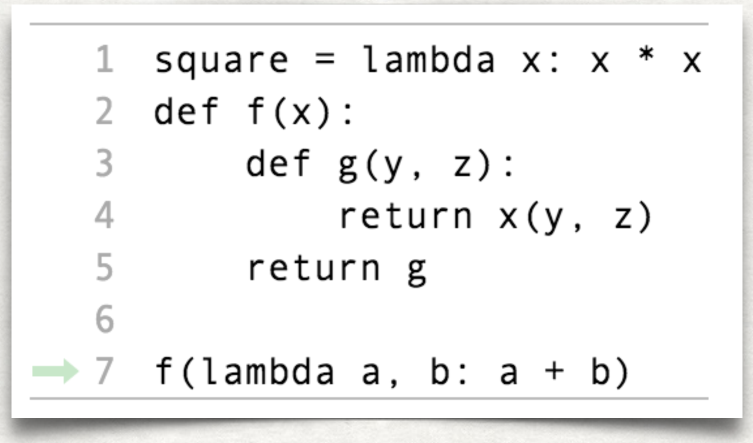
g

Return value

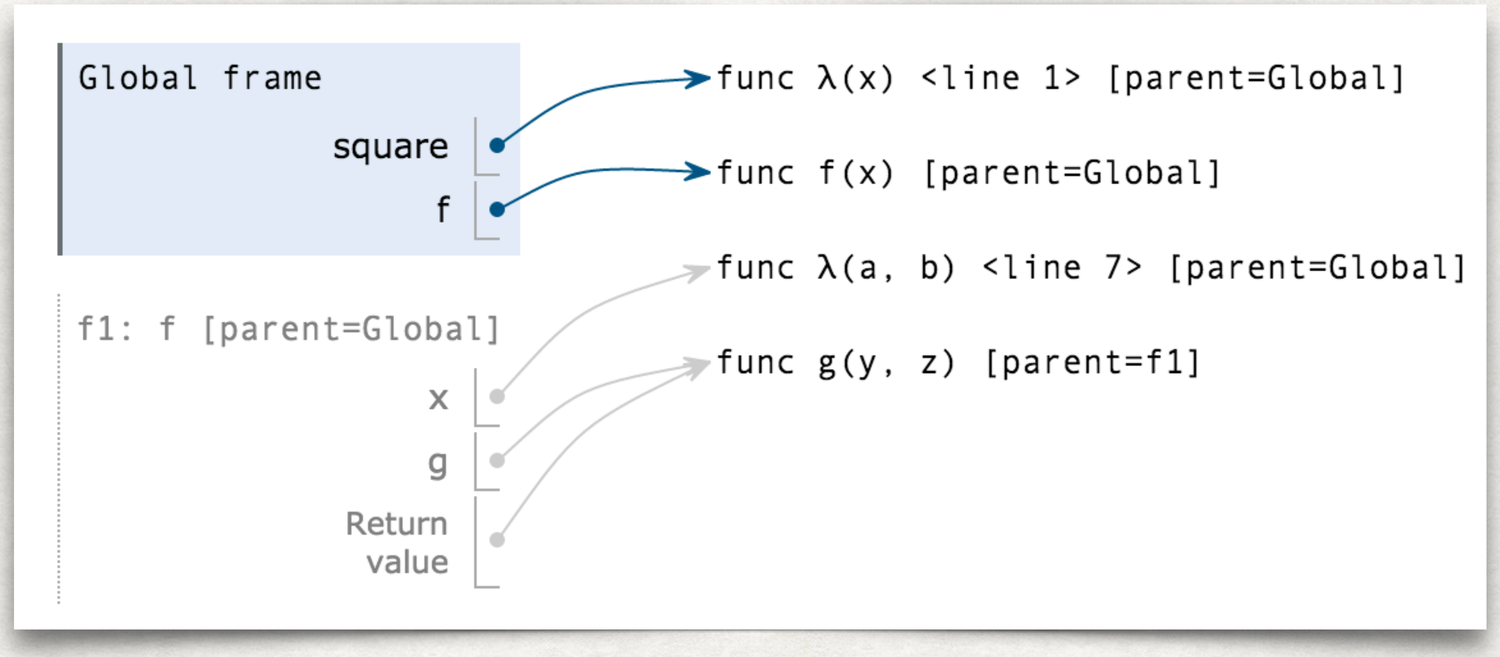
func g(y, z) [parent=f1]

LAMBDA FUNCTIONS

- Remember what frame you are in when creating lambda functions.
- Vital to the lambda’s parent frame.



```
square = lambda x: x * x
def f(x):
    def g(y, z):
        return x(y, z)
    return g

f(lambda a, b: a + b)
```



Global frame

square

f

func λ(x) <line 1> [parent=Global]

func f(x) [parent=Global]

func λ(a, b) <line 7> [parent=Global]

f1: f [parent=Global]

x

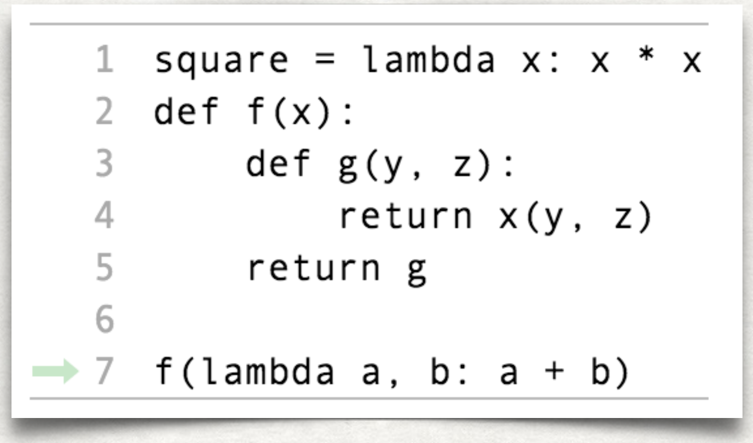
g

Return value

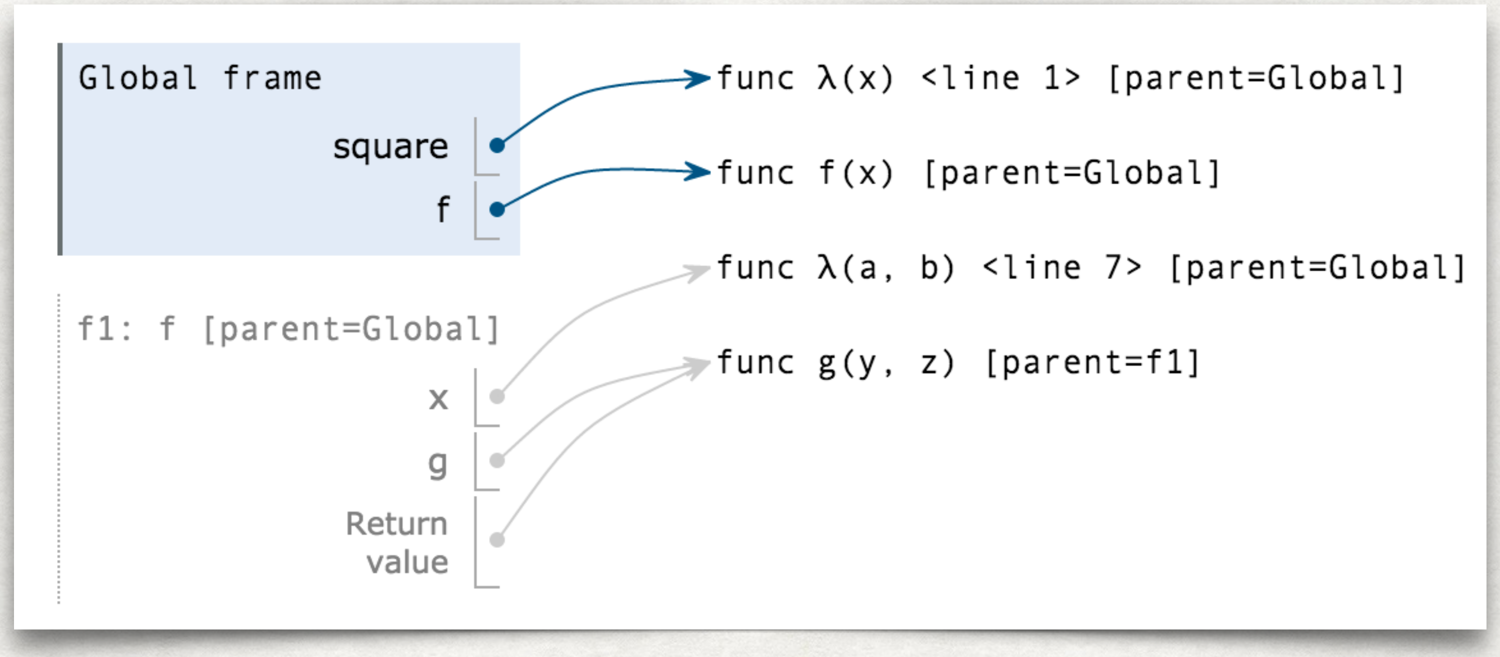
func g(y, z) [parent=f1]

LAMBDA FUNCTIONS

- Passing in newly defined lambda functions in a function call always creates the lambda object in the frame where the call expression is.



```
1  square = lambda x: x * x
2  def f(x):
3      def g(y, z):
4          return x(y, z)
5      return g
6
7  f(lambda a, b: a + b)
```



Global frame

square

f

f1: f [parent=Global]

x

g

Return value

func λ(x) <line 1> [parent=Global]

func f(x) [parent=Global]

func λ(a, b) <line 7> [parent=Global]

func g(y, z) [parent=f1]

RECURSION

- A recursive function is a function that calls itself.
- Three common steps
  - Figure our your base case(s)
  - Make the problem smaller and make a recursive call with that simpler argument
  - Use your recursive call to solve the full problem

RECURSION

- Base cases are there to stop the recursion.
- No base case —> continue making recursive calls forever

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

RECURSION

- Find a smaller problem for the recursive call.
- Make sure the problem is getting smaller **toward** the base case.
- Call the recursive function with this smaller argument.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

RECURSION

- Take the ***leap of faith*** and trust that your recursive function is correct on the smaller argument.
- Knowing that the recursive call returns what you want, how can you solve the bigger problem?

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

RECURSION

factorial(5)

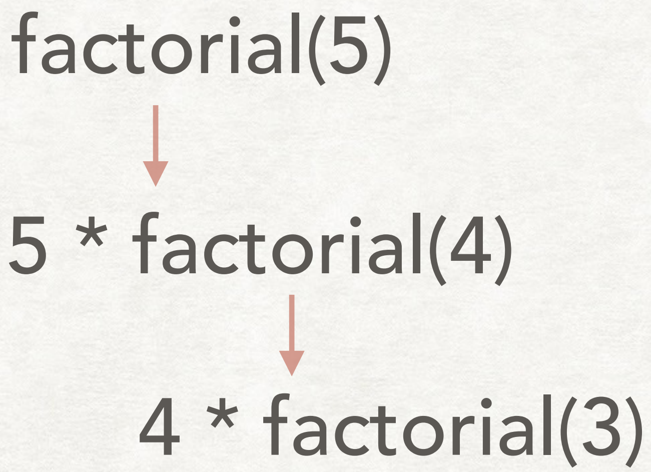
RECURSION

factorial(5)

↓

5 * factorial(4)

RECURSION

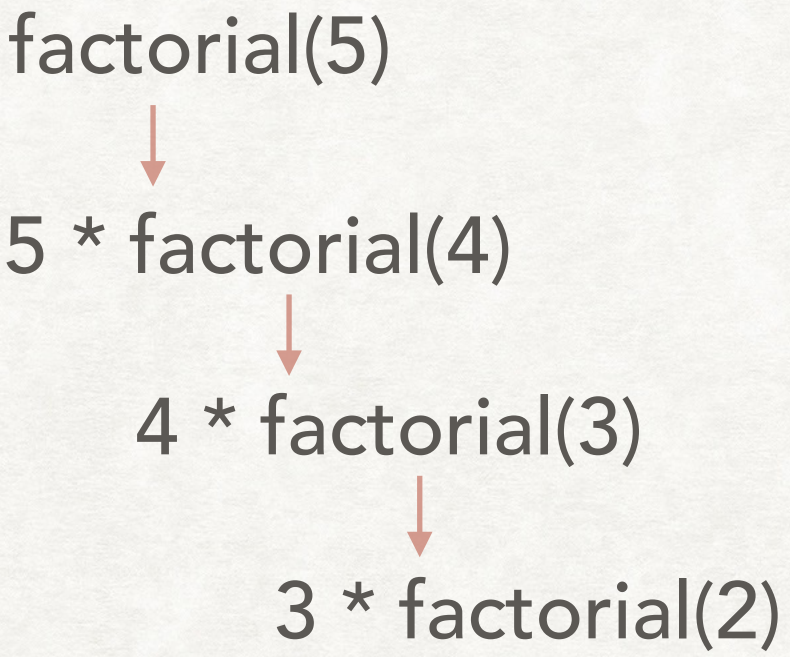


factorial(5)

5 * factorial(4)

4 * factorial(3)

RECURSION



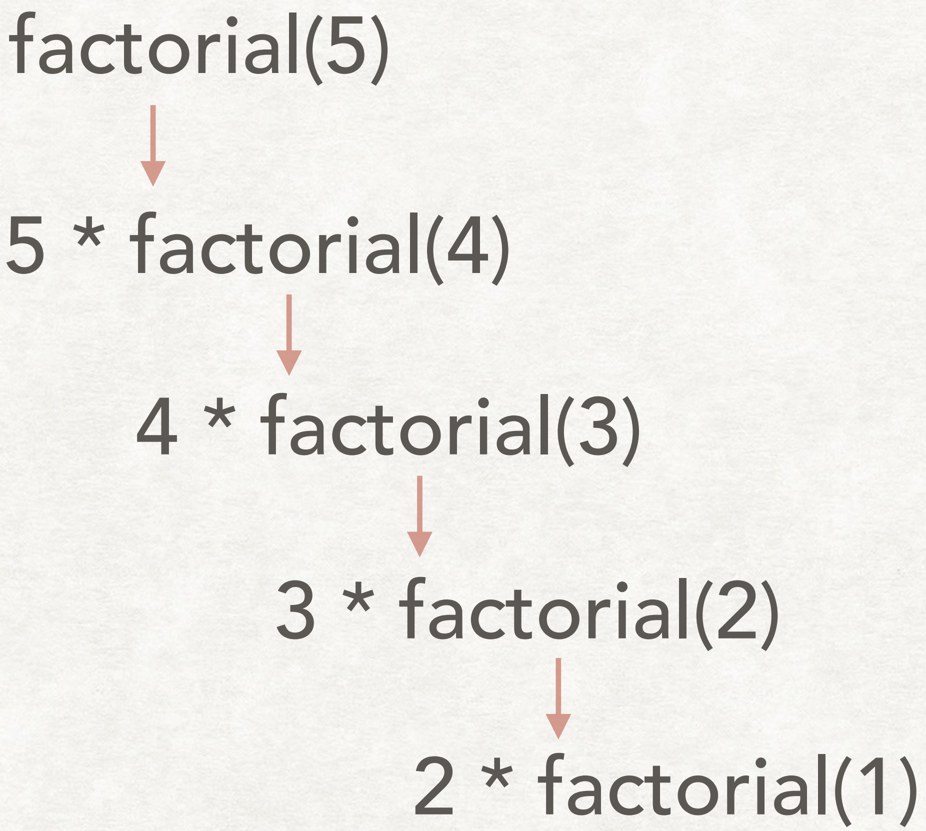
factorial(5)

5 * factorial(4)

4 * factorial(3)

3 * factorial(2)

RECURSION



factorial(5)

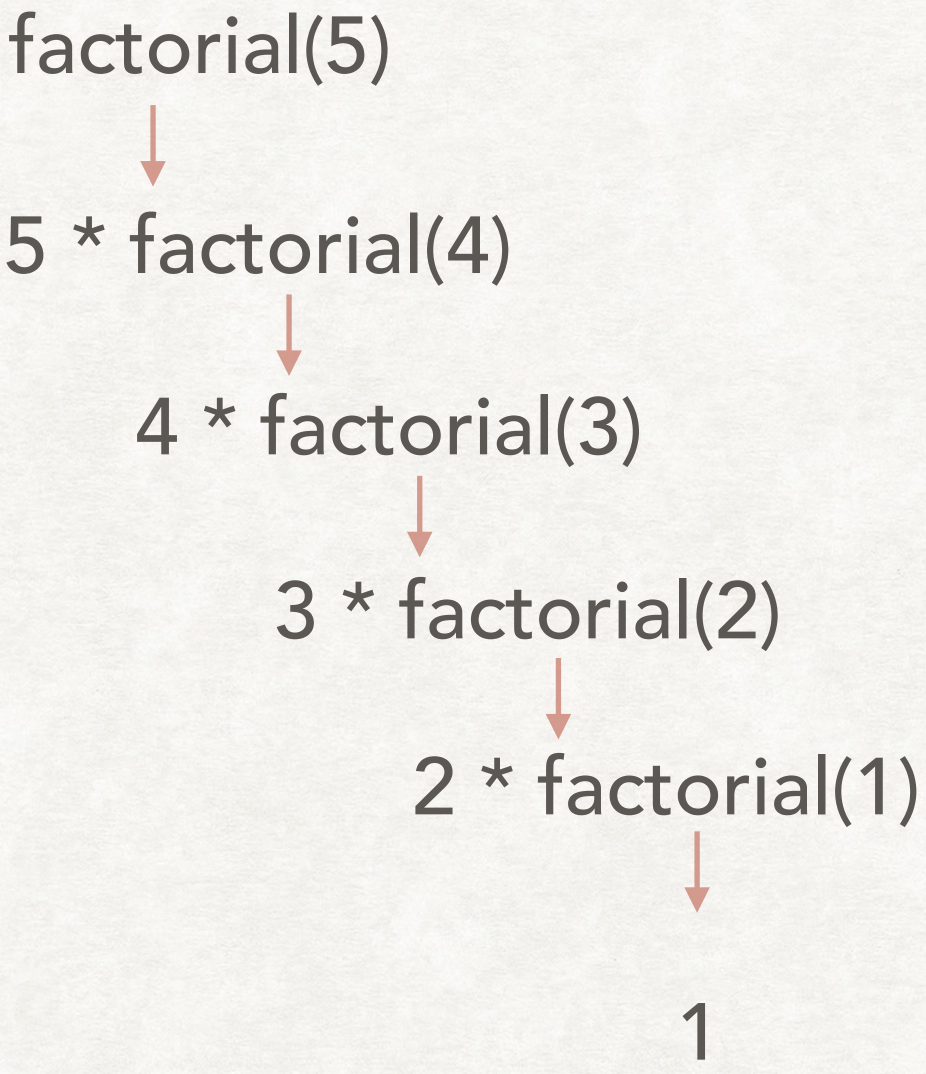
5 * factorial(4)

4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

RECURSION



factorial(5)

5 * factorial(4)

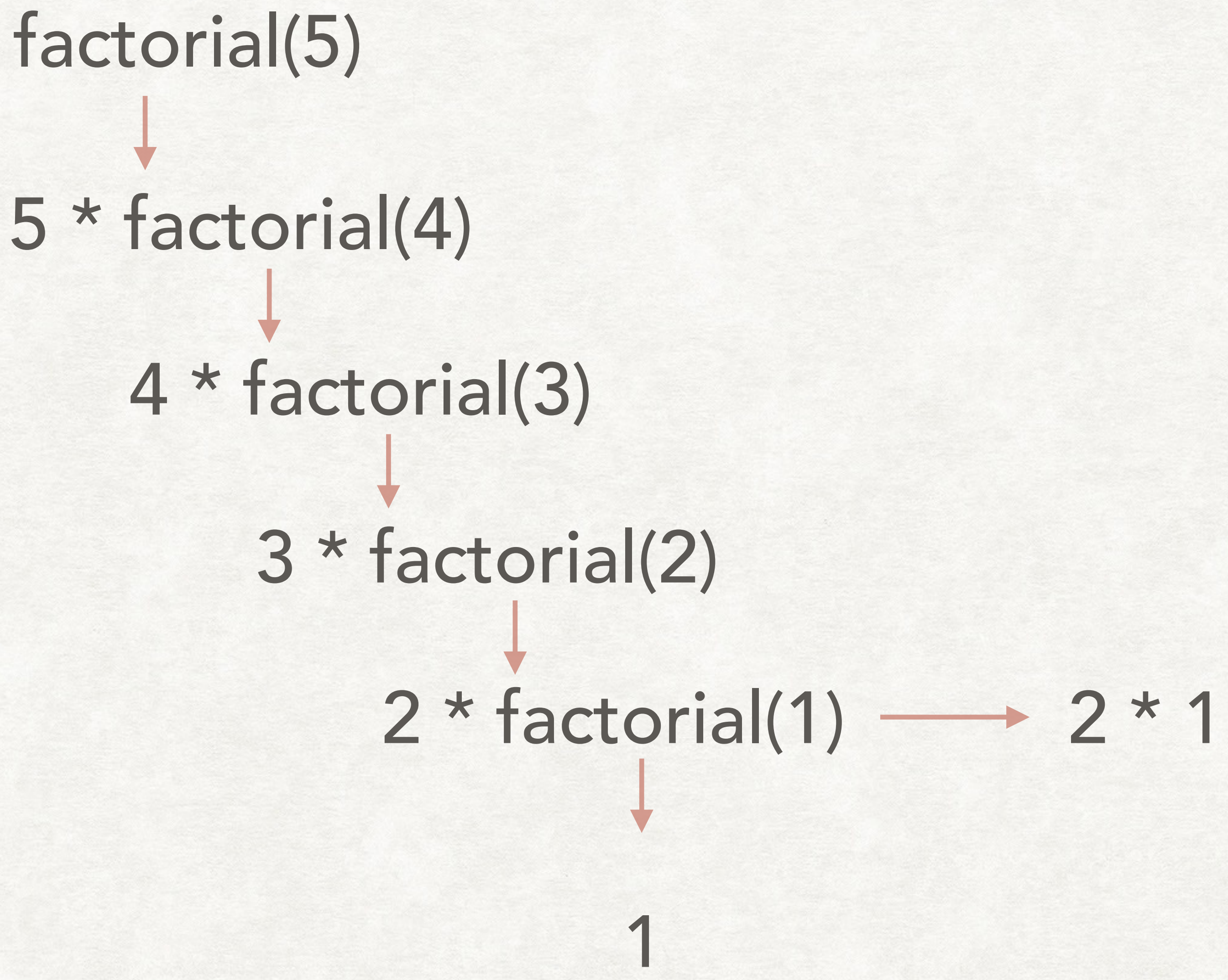
4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

1

RECURSION



factorial(5)

5 * factorial(4)

4 * factorial(3)

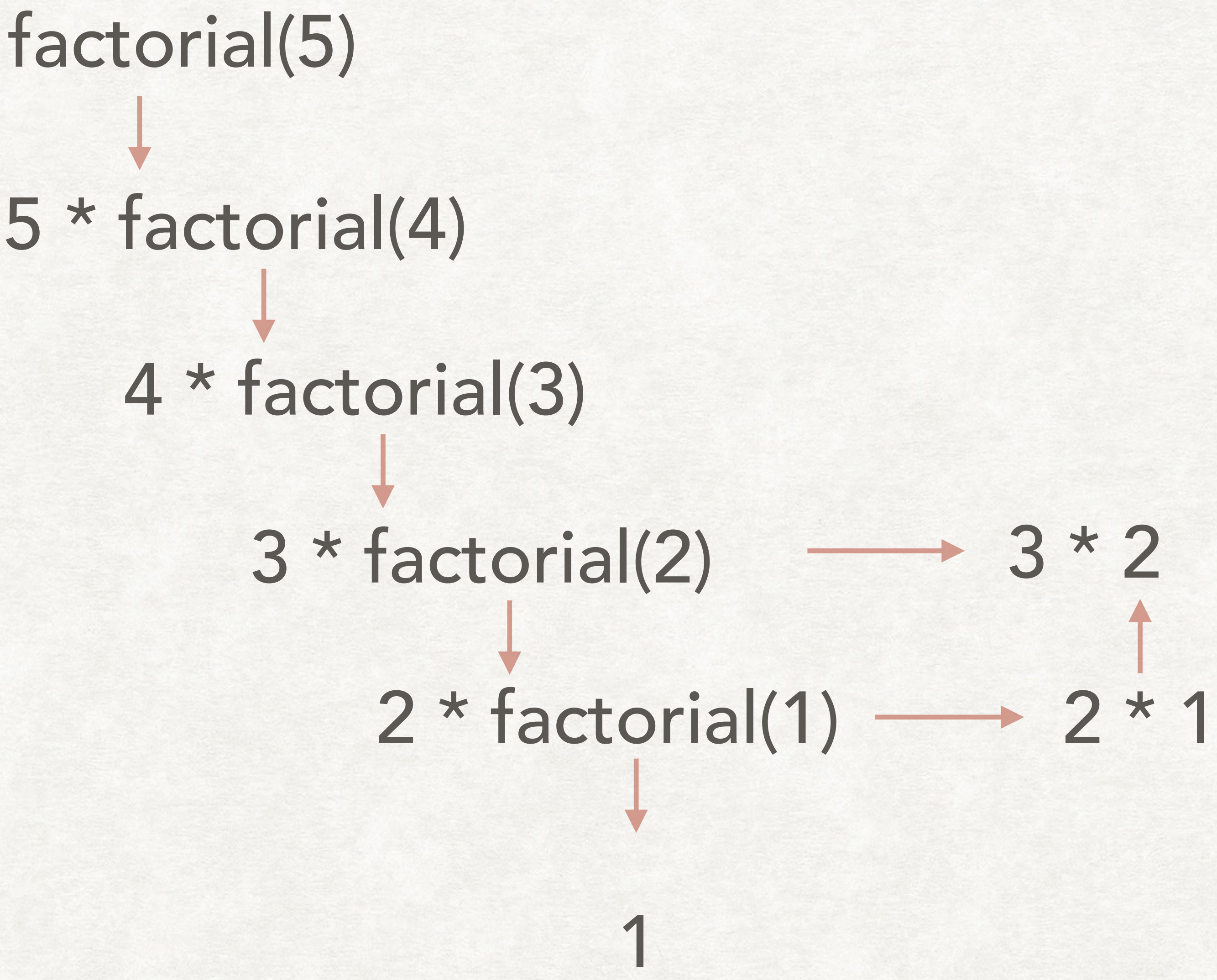
3 * factorial(2)

2 * factorial(1)

2 * 1

1

RECURSION



factorial(5)

5 * factorial(4)

4 * factorial(3)

3 * factorial(2)

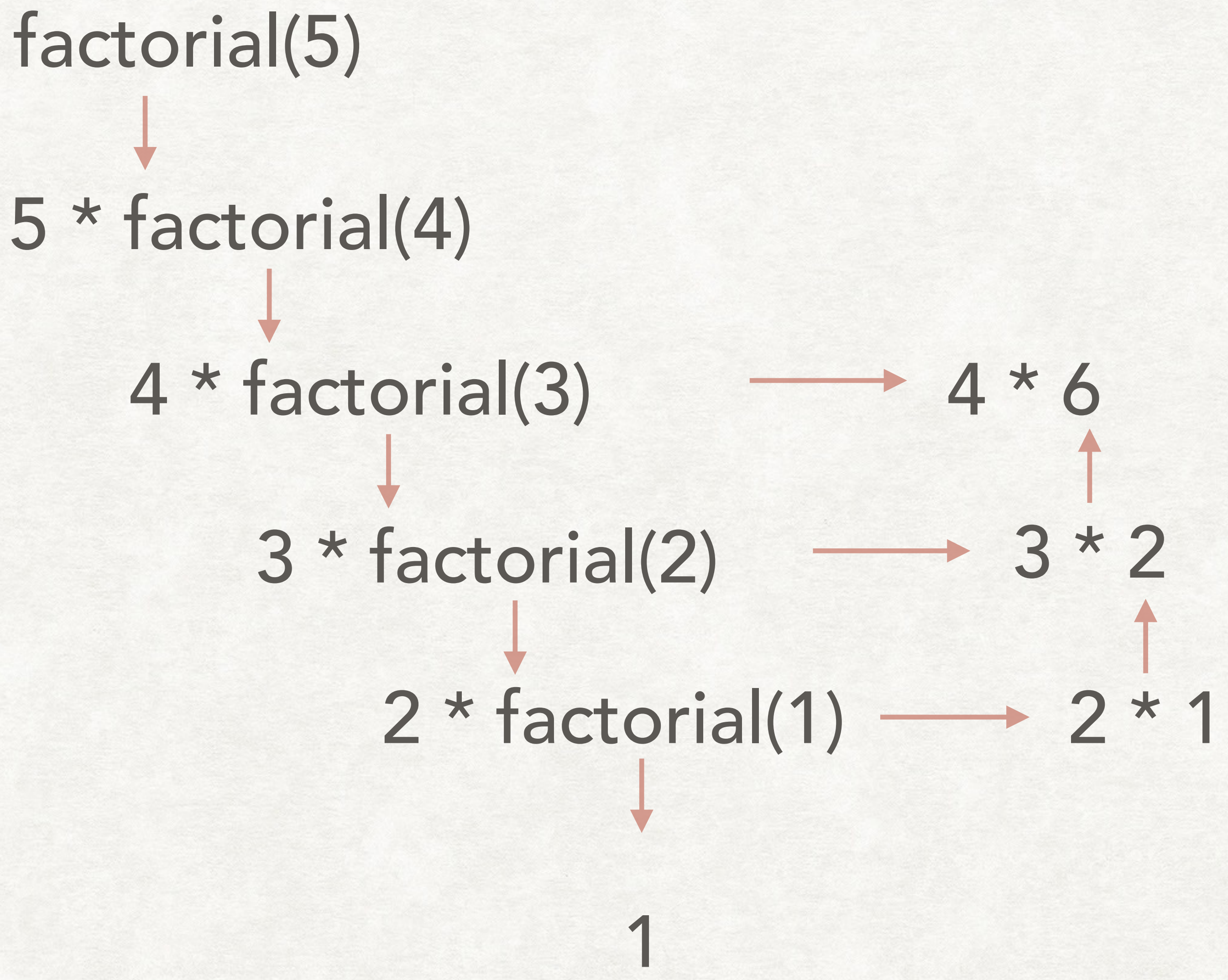
3 * 2

2 * factorial(1)

2 * 1

1

RECURSION



factorial(5)

5 * factorial(4)

4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

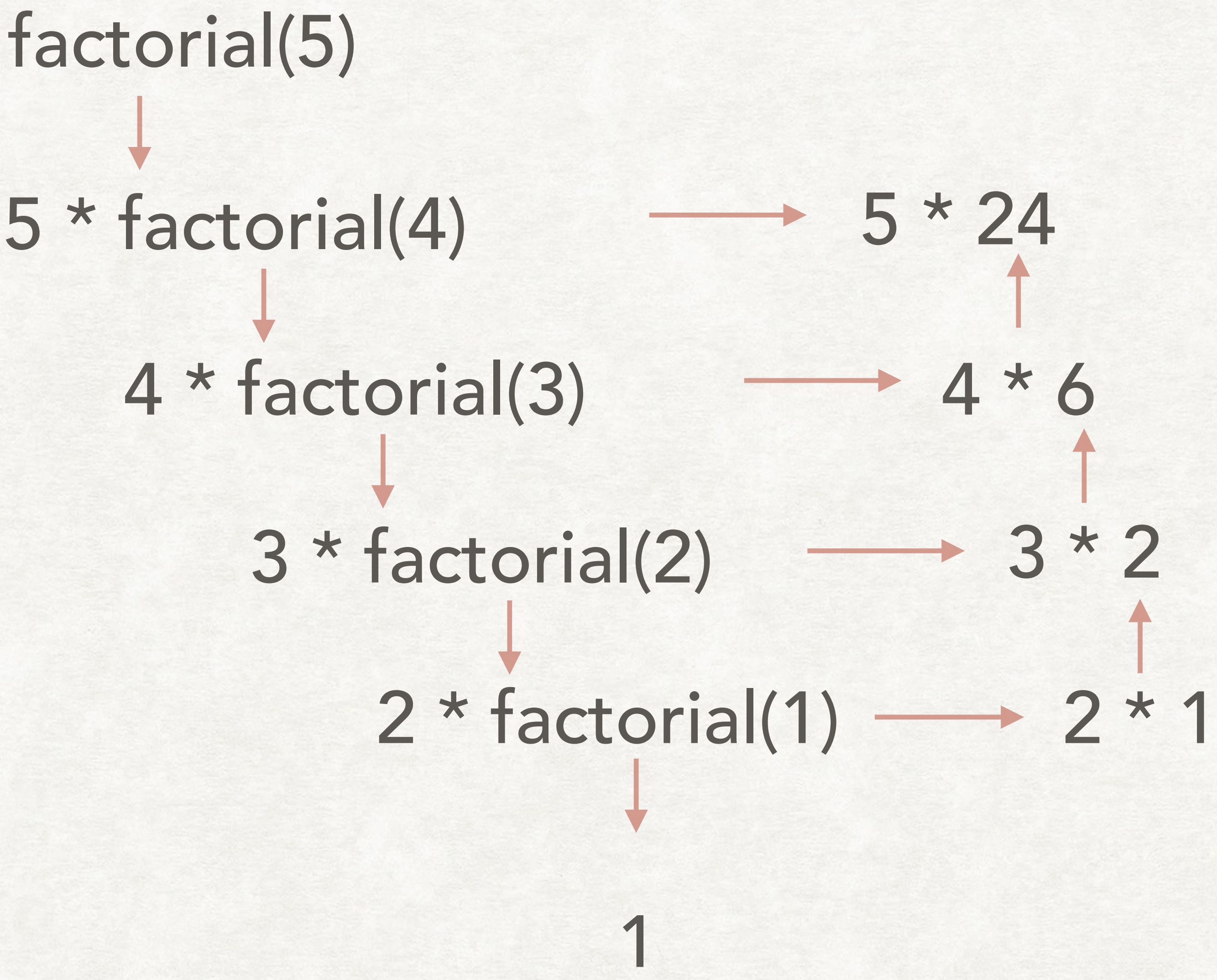
1

2 * 1

3 * 2

4 * 6

RECURSION



factorial(5)

5 * factorial(4) → 5 * 24

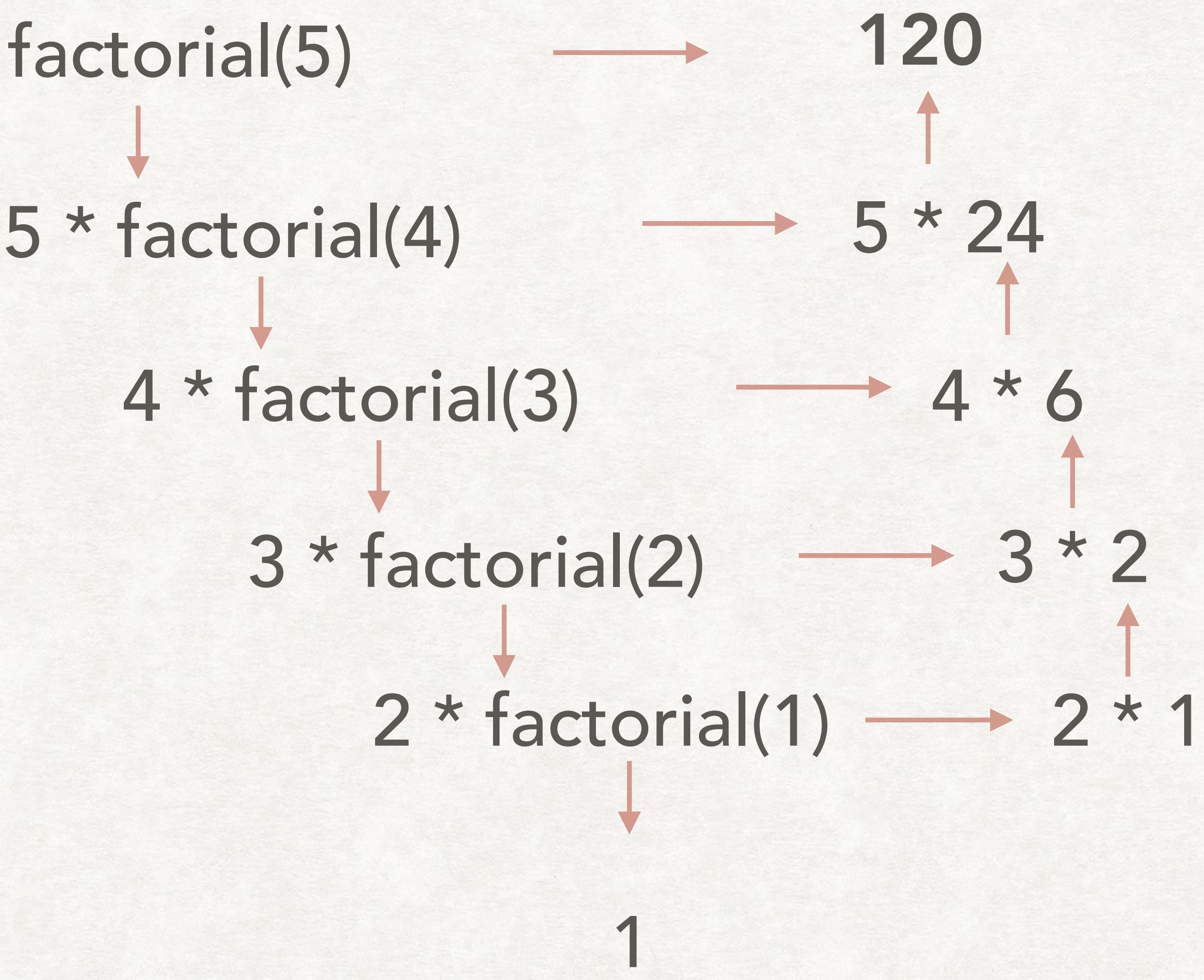
4 * factorial(3) → 4 * 6

3 * factorial(2) → 3 * 2

2 * factorial(1) → 2 * 1

1

RECURSION



factorial(5)

120

5 * factorial(4)

5 * 24

4 * factorial(3)

4 * 6

3 * factorial(2)

3 * 2

2 * factorial(1)

2 * 1

1

TREE RECURSION

- Recursive functions that make more than one recursive call in its recursive case.
- Example: fibonacci sequence

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

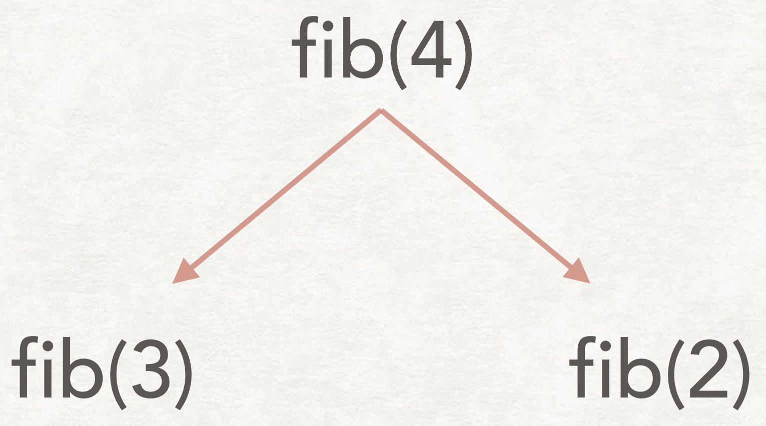
TREE RECURSION

- Recursive functions that make more than one recursive call in its recursive case

fib(4)

TREE RECURSION

- Recursive functions that make more than one recursive call in its recursive case

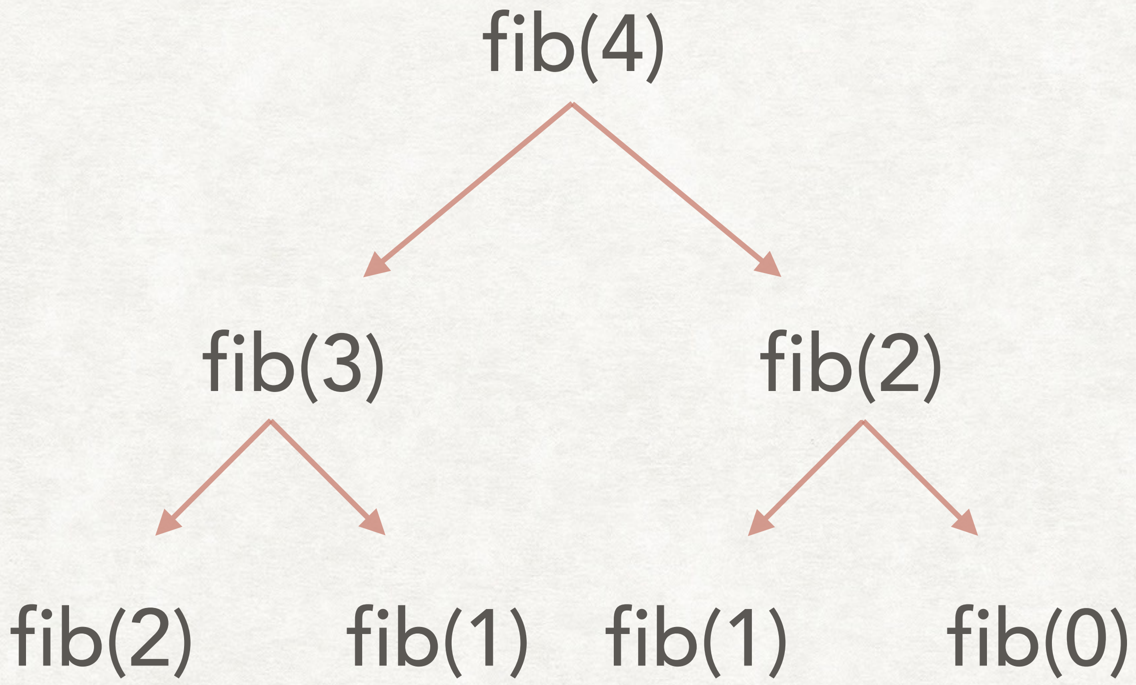

fib(4)

fib(3)

fib(2)

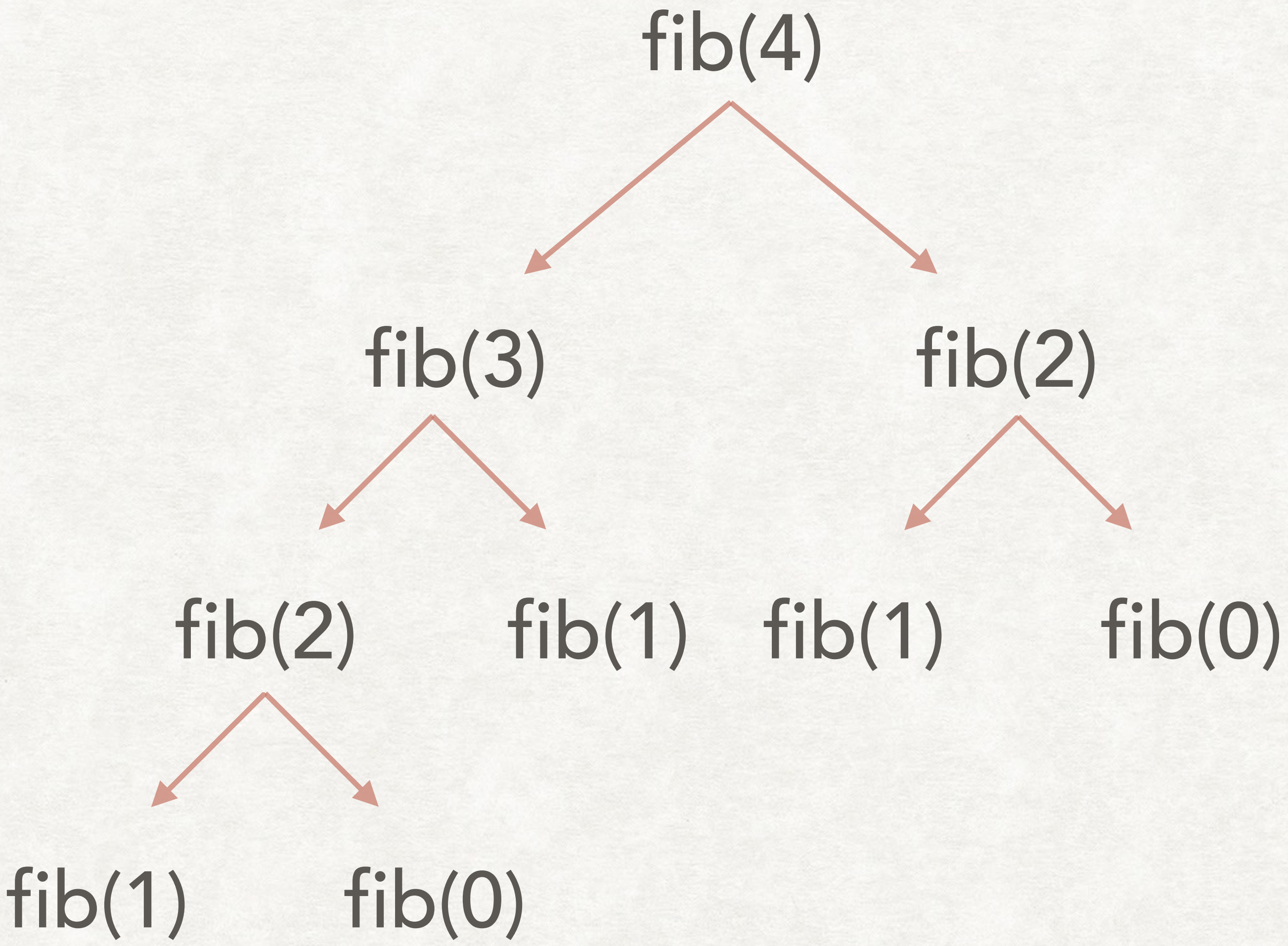
TREE RECURSION

- Recursive functions that make more than one recursive call in its recursive case



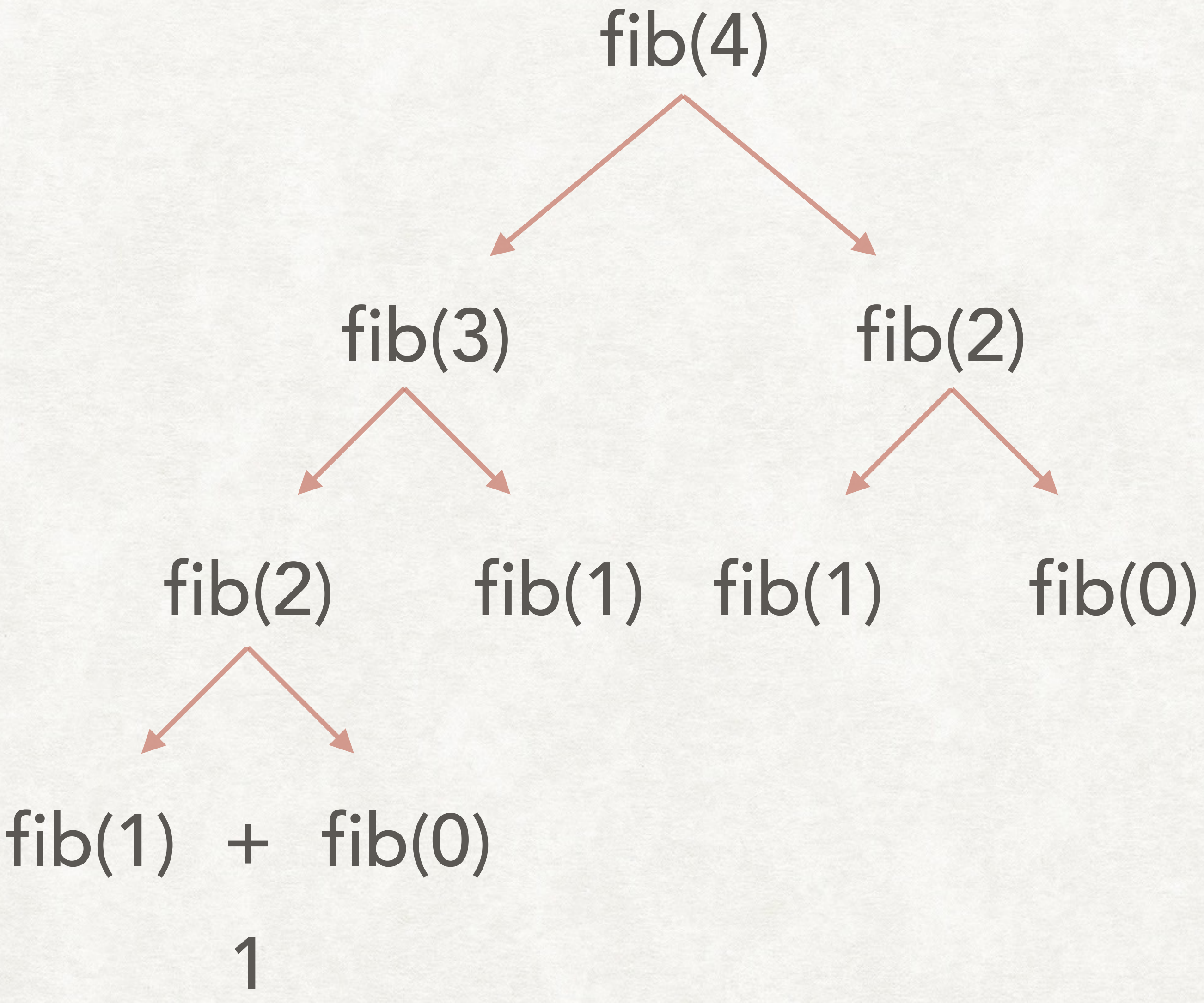
TREE RECURSION

- Recursive functions that make more than one recursive call in its recursive case



TREE RECURSION

- Recursive functions that make more than one recursive call in its recursive case



fib(4)

fib(3) fib(2)

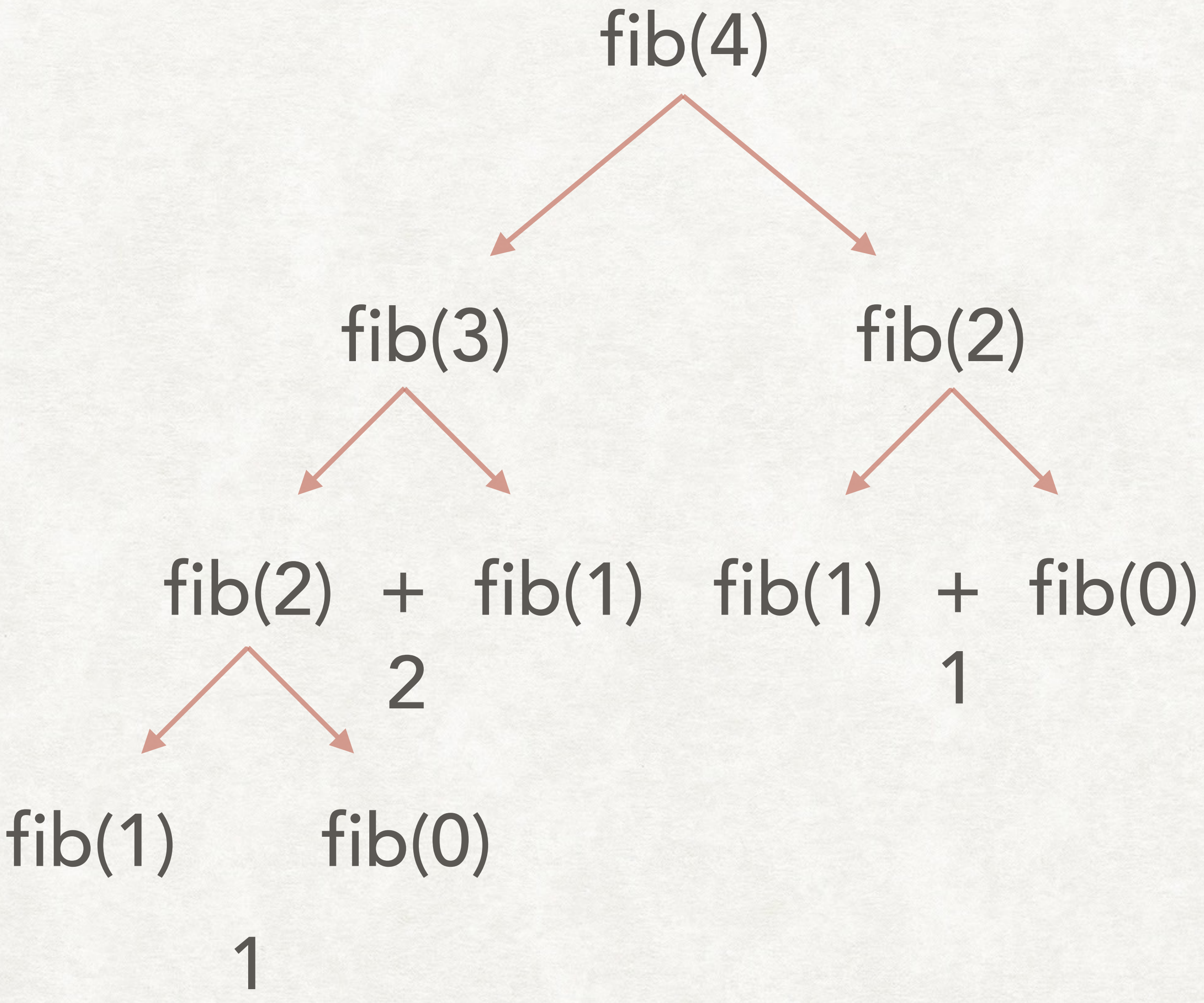
fib(2) fib(1) fib(1) fib(0)

fib(1) + fib(0)

1

TREE RECURSION

- Recursive functions that make more than one recursive call in its recursive case



fib(4)

fib(3) fib(2)

fib(2) + fib(1) fib(1) + fib(0)

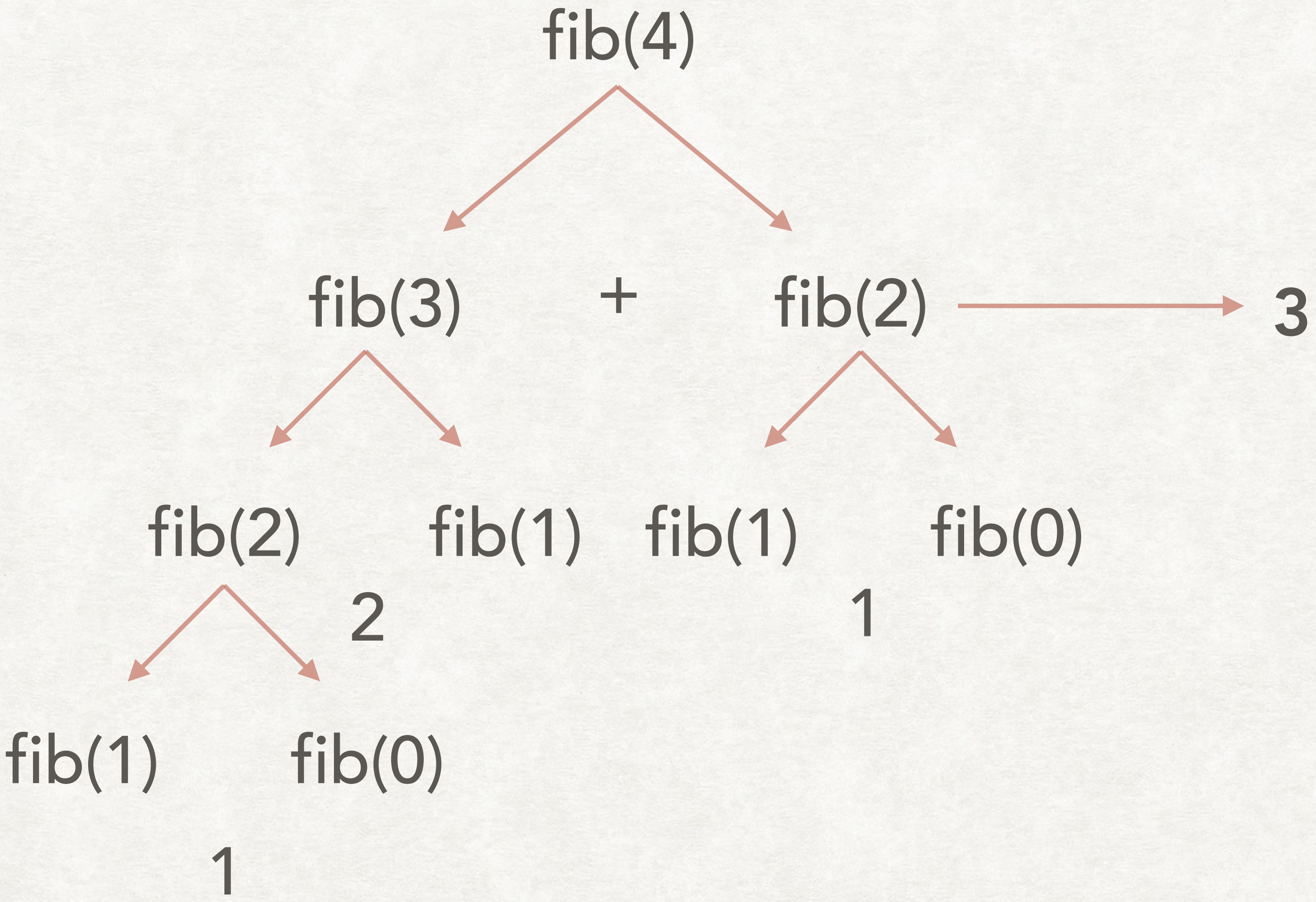
2 1

fib(1) fib(0)

1

TREE RECURSION

- Recursive functions that make more than one recursive call in its recursive case



fib(4)

fib(3) + fib(2) → 3

fib(2) fib(1) fib(1) fib(0)

2 1

fib(1) fib(0)

1

RECAP

- Environment diagrams allow us to keep track of a variables and their values.

- Recursion functions call themselves.

- Tree recursive functions call themselves multiple times from one frame.