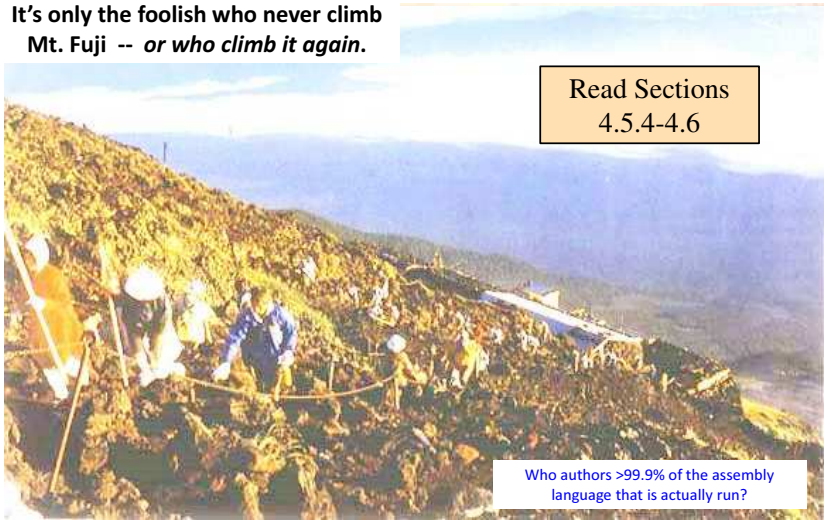
Why Assembly Language ?

It's only the foolish who never climb Mt. Fuji -- *or who climb it again*.



Read Sections 4.5.4-4.6

Who authors >99.9% of the assembly language that is actually run?

functions vs. instructions

1-1 correspondence is very rare!

```
def main():
    x = input()
    y = fun(x)
    print y

def fun(x):
    y = x*(x-1)
    return y
```

```
0  read  r1
1  calln r14 4
2  write r13
3  halt
4  copy  r13 r1
5  addn  r1 -1
6  mul   r13 r13 r1
7  jumpr r14
```

input / function call / output / the function / "*return*"

Fun: Python

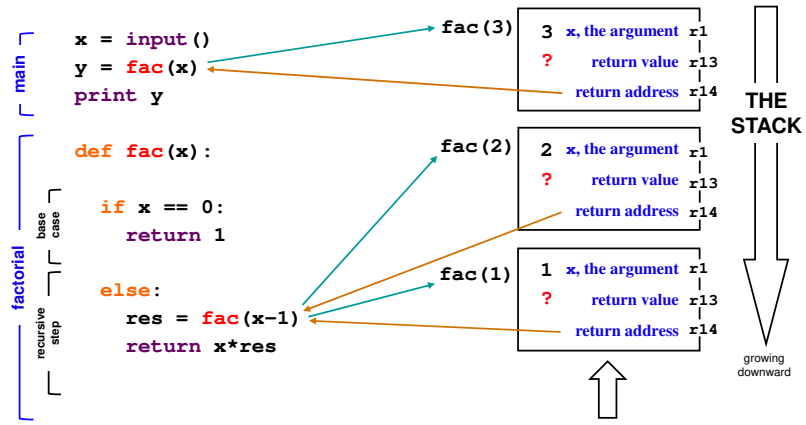
Instructions: Hmmm

From *functions* to *instructions*

Python | Assembly

```
main:
    x = input()
    y = fac(x)
    print y

factorial:
    def fac(x):
      base case:
        if x == 0:
          return 1
      recursive step:
        else:
          res = fac(x-1)
          return x*res
```



One stack frame per function call

Functions != instructions

```
def main():
    x = input()
    y = fun(x)
    print y+x

def fun(x):
    y = x*(x-1)
    return y
```

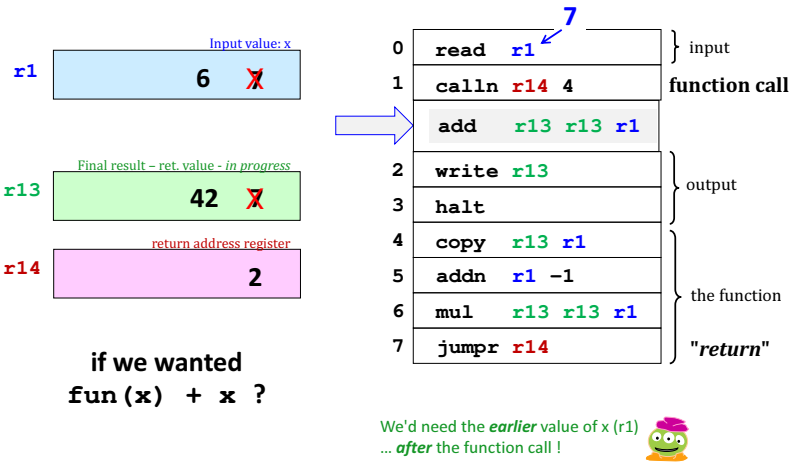
```
0  read  r1
1  calln r14 4
   add   r13 r13 r1
2  write r13
3  halt
4  copy  r13 r1
5  addn  r1 -1
6  mul   r13 r13 r1
7  jumpr r14
```

input / function call / output / the function / "*return*"

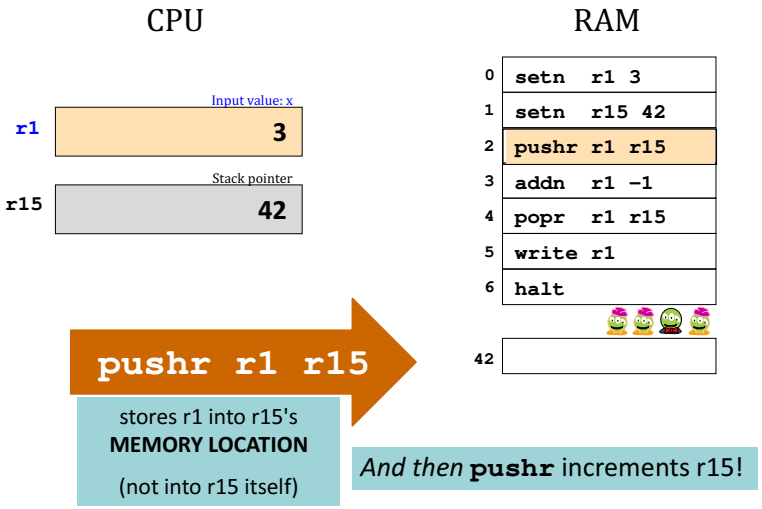
But what if we wanted fun(x) + x ?

We'd need the ***earlier*** value of x (r1) ... ***after*** the function call !

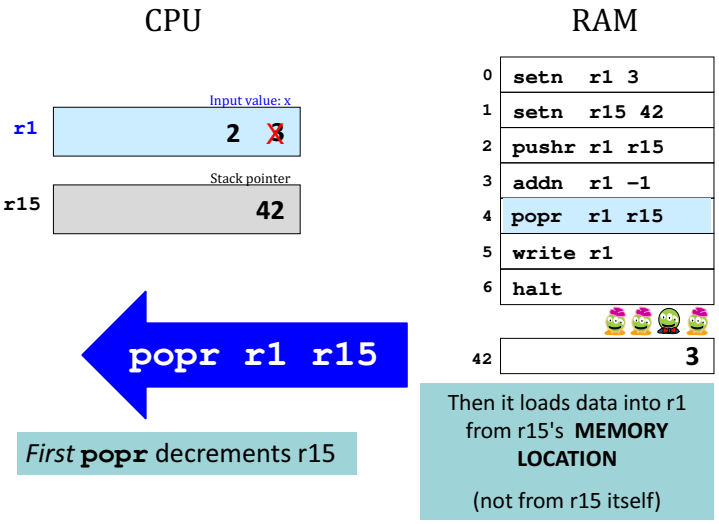
Functions != instructions



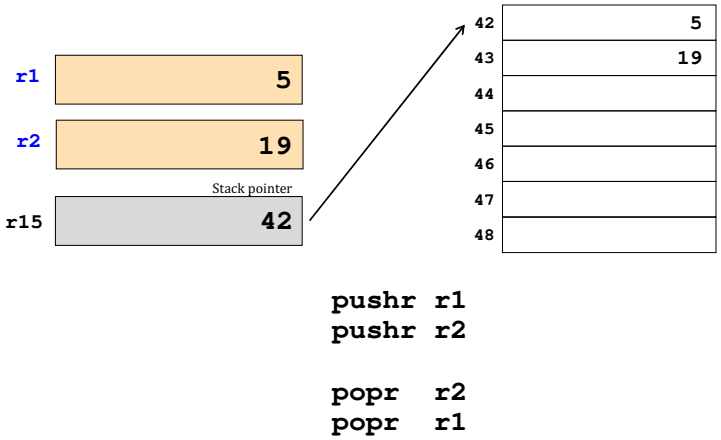
pushr stores TO memory



popr loads FROM memory



What's the Point?



Python Hmmm

```
x = int(input())
y = fac(x)
print y

def fac(x):
  if x==0:
    return 1
  else:
    RES = fac(x-1)
    return x*RES
```

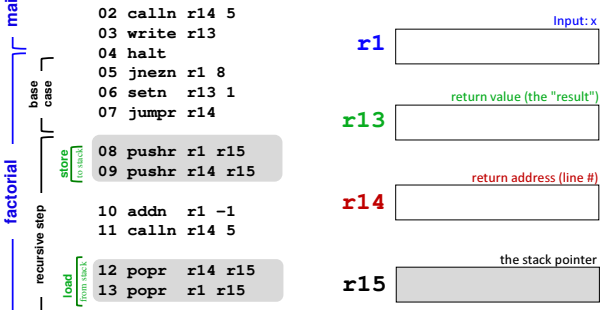
```
00 read  r1
01 setn  r15 42
02 calln r14 5
03 write r13
04 halt
05 jnezn r1 8
06 setn  r13 1
07 jumpr r14
08 pushr r1  r15
09 pushr r14 r15
10 addn  r1 -1
11 call  r14 5
12 popr  r14 r15
13 popr  r1 r15
14 mul   r13 r13 r1
15 jumpr r14
```

main; factorial function; base case; recursive step; input; function call; output; base case; recursive call; calculate final result; store; load

For an input of 0, trace what happens here ...

CPU + Registers

Main Memory "*the stack*"



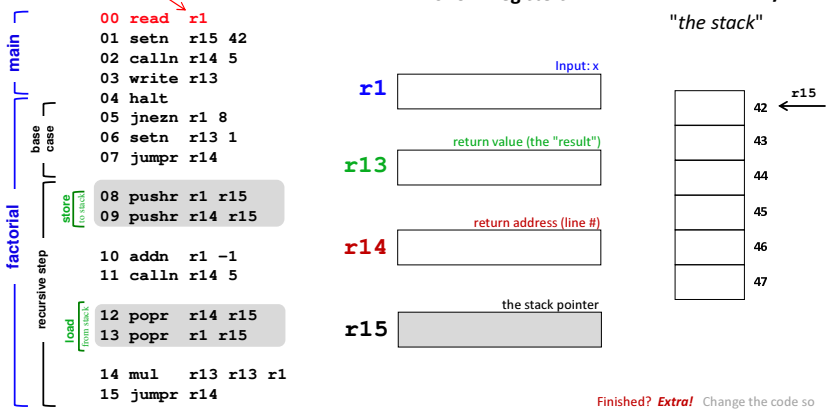
How many lines of assembly code get executed, in total, when the input is 0?

Yesterday I'd never heard of *Recursive Assembly*, but today I ❤ it.

For an input of 3, trace what happens here ...

CPU + Registers

Main Memory "*the stack*"



Finished? ***Extra!*** Change the code so that it instead raises r2 to the r1 power – you need to change ***very few lines !***

It's easy to get lost! Follow these line numbers below to stay on track...

0, 1, 2, 5, 8, 9, 10, 11, 5, 8, 9, 10, 11, 5, 8, 9, 10, 11, 5, 6, 7, 12, 13, 14, 15, 12, 13, 14, 15, 12, 13, 14, 15, 3, 4

Journey: instructions to functions...

Python

```
x = input()
y = fac(x)
print y

def fac(x):

  if x==0: return 1

  else:
    RES = fac(x-1)
    return x*RES
```

Assembly

Function call !

- **push** everything to memory
- make the function **`call`**
- ***destroy data as needed***
- **`jumpr`** to return
- **pop** all *back* from memory

Conceptually

Strategy: standardize registers

Simplify by having a **standard place** for **standard data**.

`r1`	argument(s) will be in r1, r2, ...
`r13`	the return *value* (result) will be in r13
`r14`	the return *address* will be in r14
`r15`	the "stack pointer" will be in r15

This is the NEXT location in memory for storing data

Organizationally

Quiz

Names(s) _____

```
0     fac(3)


1  def fac(x):
       """Factorial with printing"""
2      print("x is", x)

3      if x == 0:
4          print("Base: x:", x, "Res: 1")
5          return 1

6      else:
7          print("Next: fac(", x-1, ")")
8          smaller = fac(x-1)
9          result = x * smaller
10         print("x:", x, "Res:", result)
11         return result
```

What will this call to fac(3) *print*?

[and by ***which*** line?]

What's printed...	Circle the line #
`x is 3`	2 · 4 · 7 · 10
`Next: _____`	2 · 4 · 7 · 10
	2 · 4 · 7 · 10
	2 · 4 · 7 · 10
	2 · 4 · 7 · 10
	2 · 4 · 7 · 10
	2 · 4 · 7 · 10
	2 · 4 · 7 · 10
	2 · 4 · 7 · 10
	2 · 4 · 7 · 10
`x:3 Res:6`	2 · 4 · 7 · 10