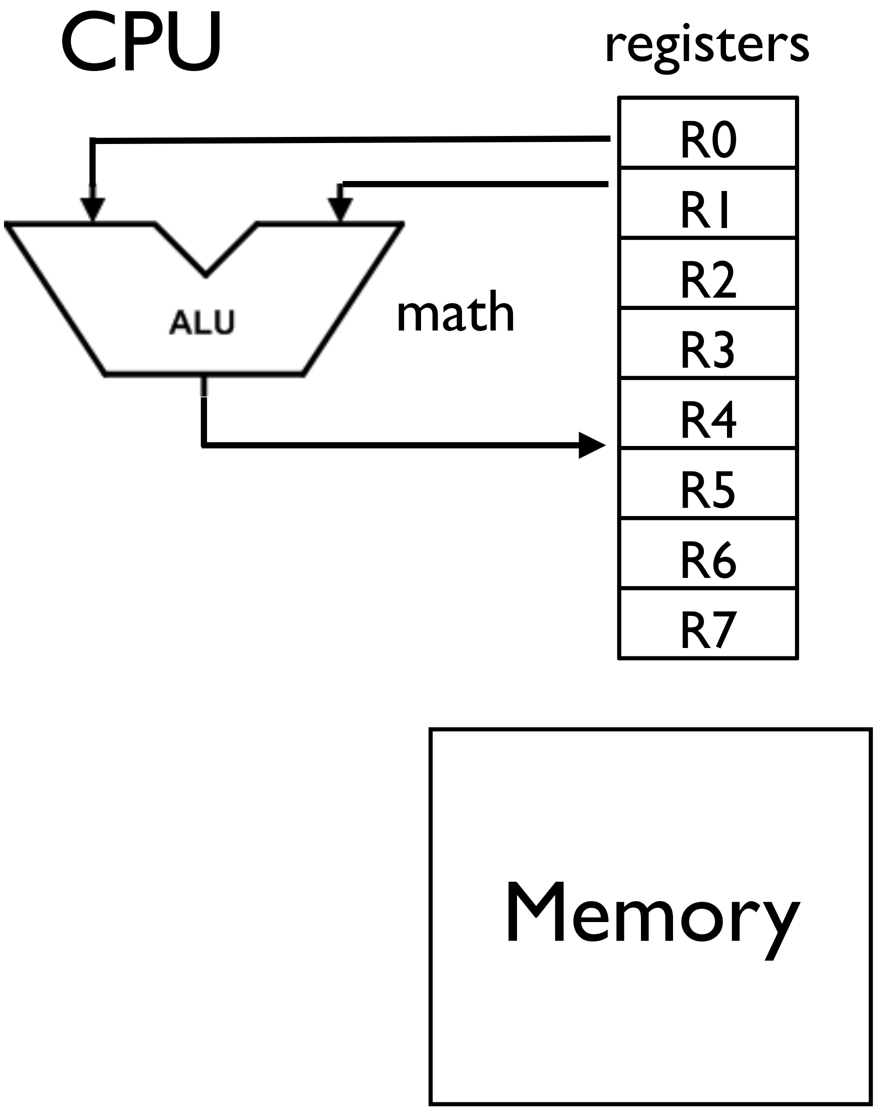
Part 8

The Runtime

Runtime Environment

- Programs execute in the real world
    - On a real CPU
    - On a real computer
    - On a real operating system
- There are various details...

Mapping to Hardware



CPU

registers

math

R0

R1

R2

R3

R4

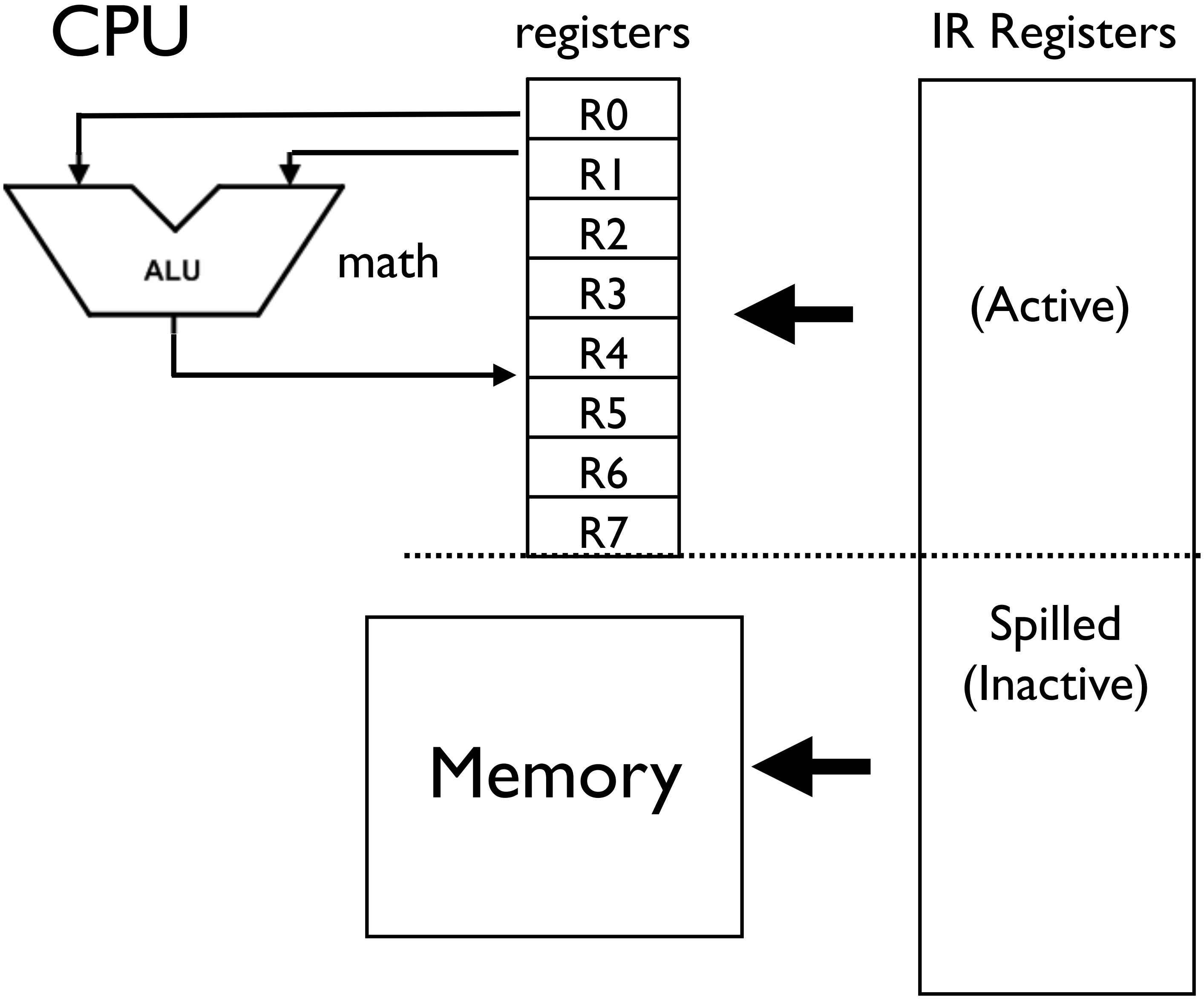
R5

R6

R7

Memory

Mapping to Hardware



Active registers in IR are mapped to CPU registers

If you run out of registers, inactive registers need to be spilled to memory (on the call stack)

Commentary

- The IR->hardware mapping is really interesting
- Many different CPUs and architectures
- GPUs, SIMD, FPUs, etc.
- Instruction scheduling
- Hardware bug workarounds
- Beyond scope of this course (LLVM handles it)

OS Interface

- Programs need to perform operations related to the host operating system
  - I/O (printing, read/write, files, etc.)
  - Memory management
  - Threads/processes/etc.
- This functionality usually provided in the form of a low-level system library (e.g., libc).

RTS Library

- The programming language may have its own "run time system"
  - Built-in functions
  - Garbage collection
  - Error handling
- This is also a library. May not be written in the same language (typical choice is C/C++).

Startup/Init

- There is often an initialization/startup process
    - Initialization of the runtime
    - Initialization of global variables
- This is often handled in an implicit function called "_start()", "_init()" or something similiar.

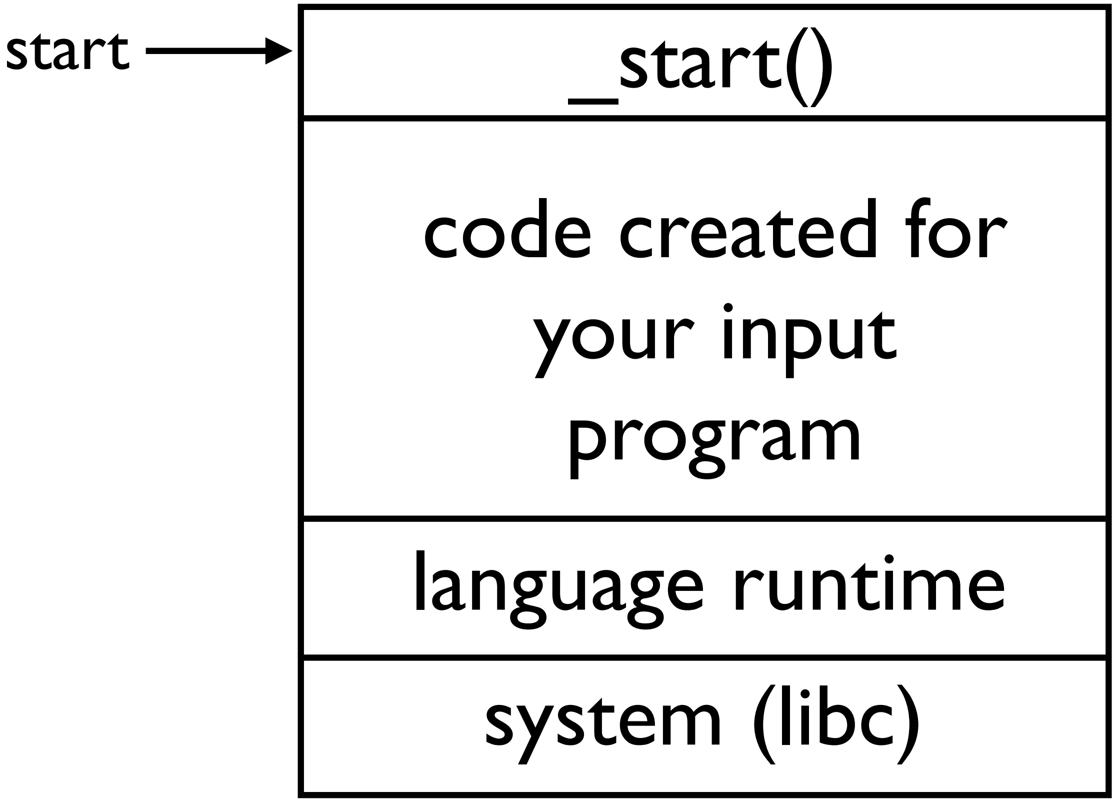
Program Startup

- Initialization example:

```
var x int = v1;
var y int = v2;
...
func main() int {
    // Written by the programmer
    ...
    return 0;
}

func __start() int {
    // Initialization (created by compiler)
    x = v1;     // Setting of global variables
    y = v2;
    return main();
}
```

Executable Program


start

_start()

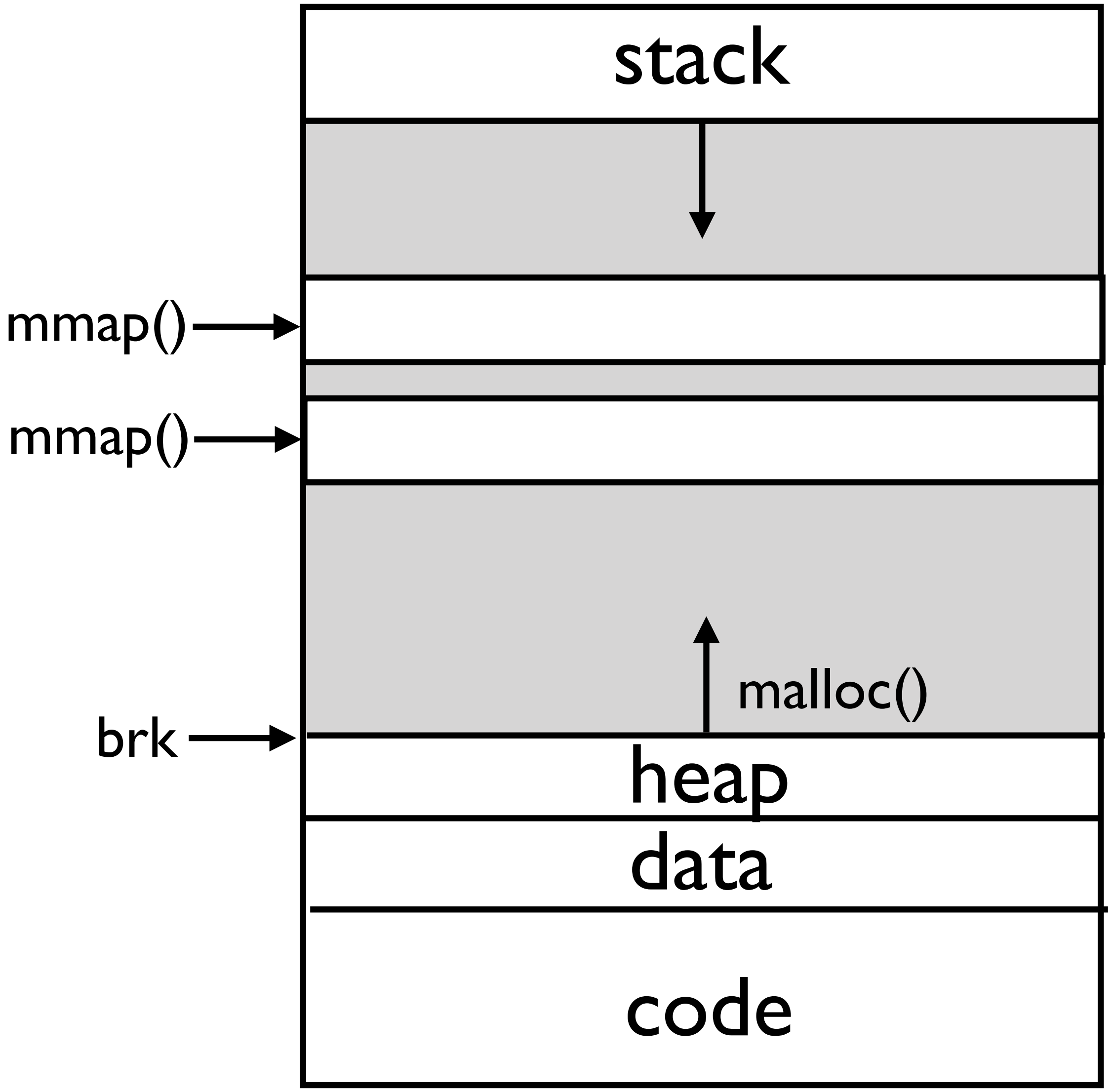
code created for your input program

language runtime

system (libc)

Memory Layout

- Stack (locals)
- Data (globals)
- Heap (objects)
- Address space is typically quite large (64 bits)



Function Scoping

- Most languages use lexical scoping
- Pertains to visibility of identifiers

```
var a = 13;
func foo() int {
    var x = 37;
    return a + x;    // Both a and x are visible
}

func bar() int {
    var y = 42;
    var c = a + y;    // a and y are visible.
    print(x);         // Error: x is not visible
}
```

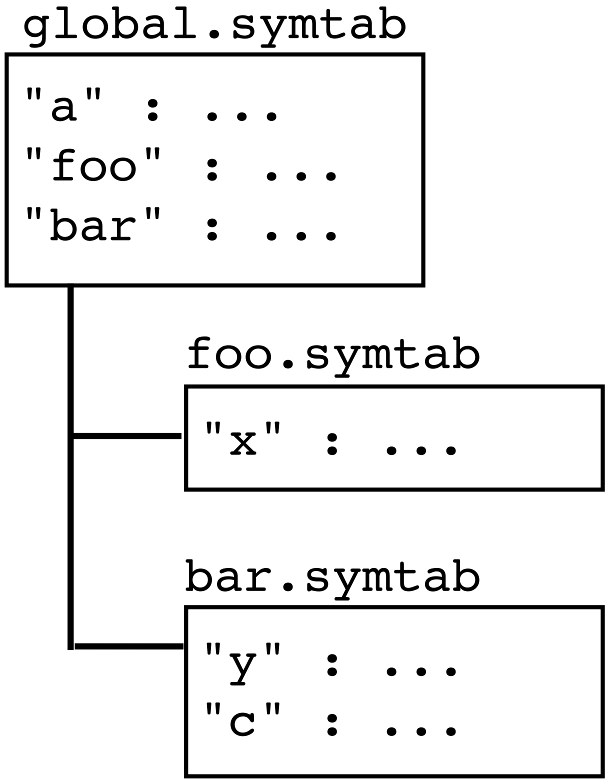
- Identifiers defined in enclosing source code context of a particular statement are visible

Scope Implementation

- In the compiler: nested tables

```
var a = 13;
func foo() int {
    x = 37;
    return a + x;
}

func bar() int {
    y = 42;
    var c = y + a;
    print(x);
}
```



```
global.symtab
"a"   : ...
"foo" : ...
"bar" : ...

foo.symtab
"x" : ...

bar.symtab
"y" : ...
"c" : ...
```

- Symbol table lookup checks all parents
- The nesting is by syntactic/lexical structure

Function Runtime

- Each invocation of a function creates a new environment of local variables
- Known as an activation frame (or record)
- Activation frames make up the call stack

Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```

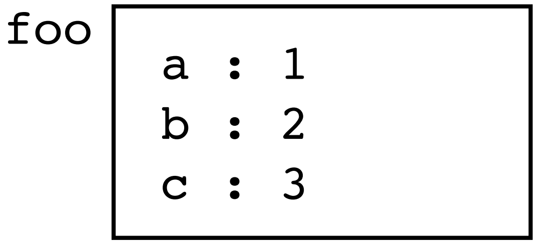
Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```



foo

```
a : 1
b : 2
c : 3
```

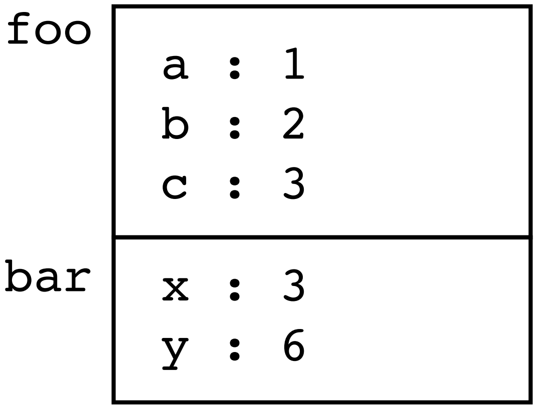
Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```



foo

a : 1
b : 2
c : 3

bar

x : 3
y : 6

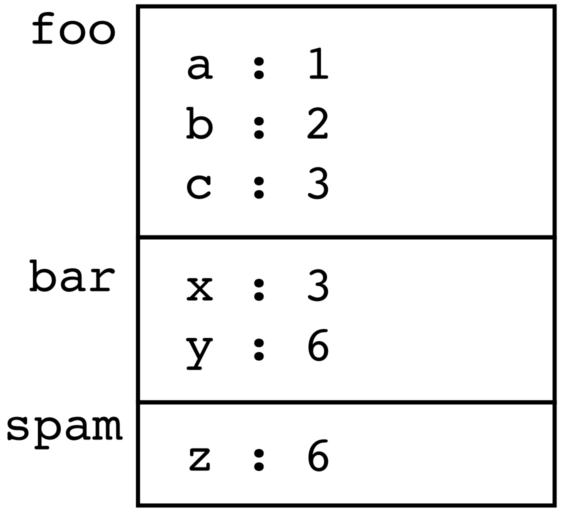
Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```



foo

a : 1
b : 2
c : 3

bar

x : 3
y : 6

spam

z : 6

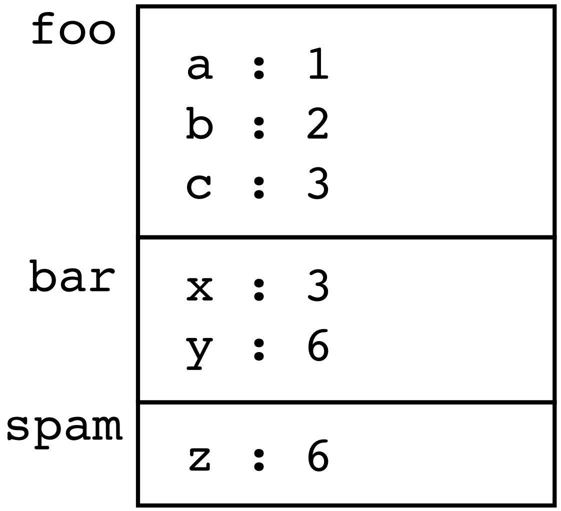
Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```



foo

a : 1
b : 2
c : 3

bar

x : 3
y : 6

spam

z : 6

Note: Frames are NOT related to scoping of variables (functions don't see the variables defined inside other functions).

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)
```

(caller)

```
def foo(x,y):
    z = x + y
    return z
```

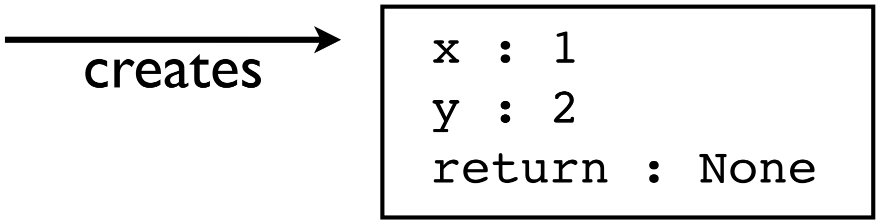
(callee)

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```



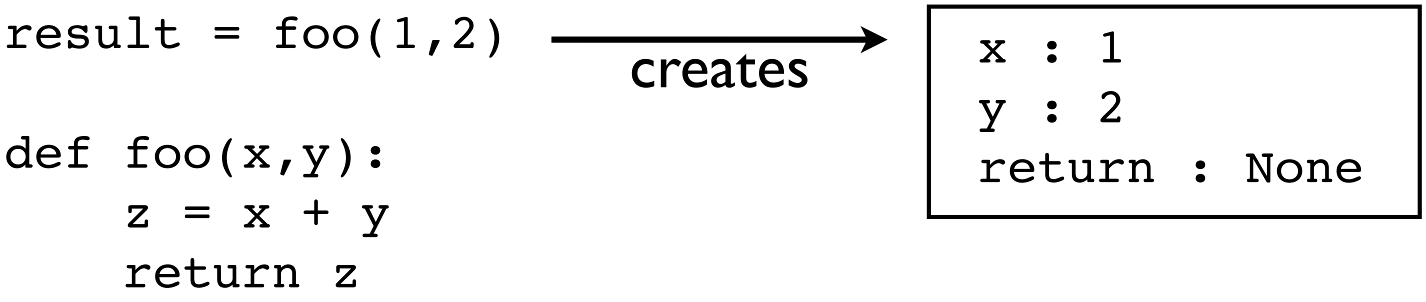
creates

```
x : 1
y : 2
return : None
```

Caller is responsible for creating a new frame and populating it with input arguments.

Frame Management

- Management of Activation Frames is managed by both the caller and callee



```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

creates

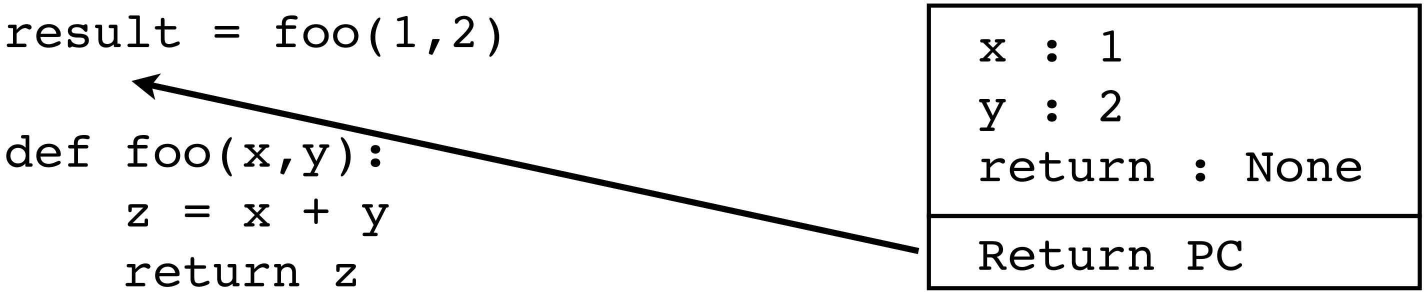
```
x : 1
y : 2
return : None
```

<u>Semantic Issue: What does the frame contain?</u>
Copies of the arguments? (Pass by value)
Pointers to the arguments? (Pass by reference)

Depends on the language

Frame Management

- Management of Activation Frames is managed by both the caller and callee



```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

```
x : 1
y : 2
return : None
Return PC
```

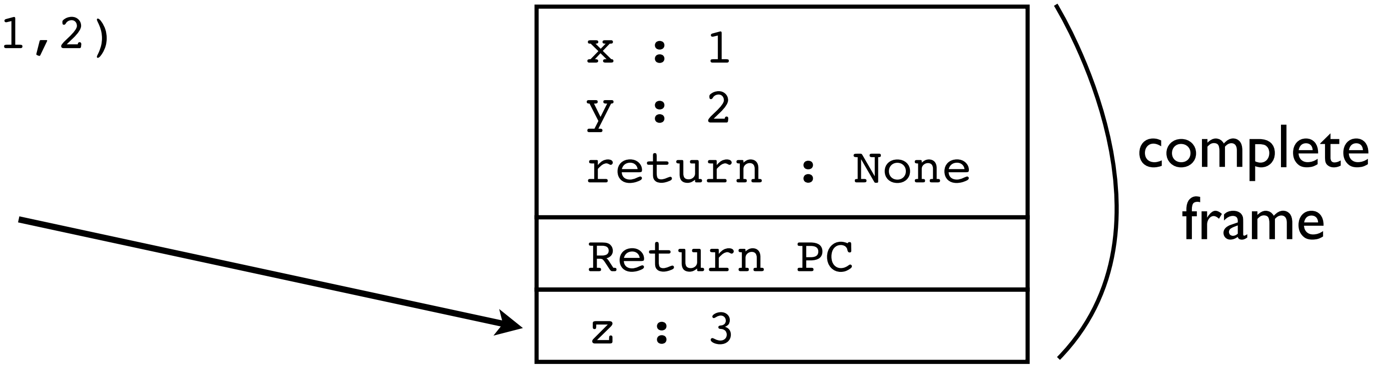
Return address (PC) recorded in the frame (so you can get back to the caller upon return)

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```



```
x : 1
y : 2
return : None
Return PC
z : 3
```

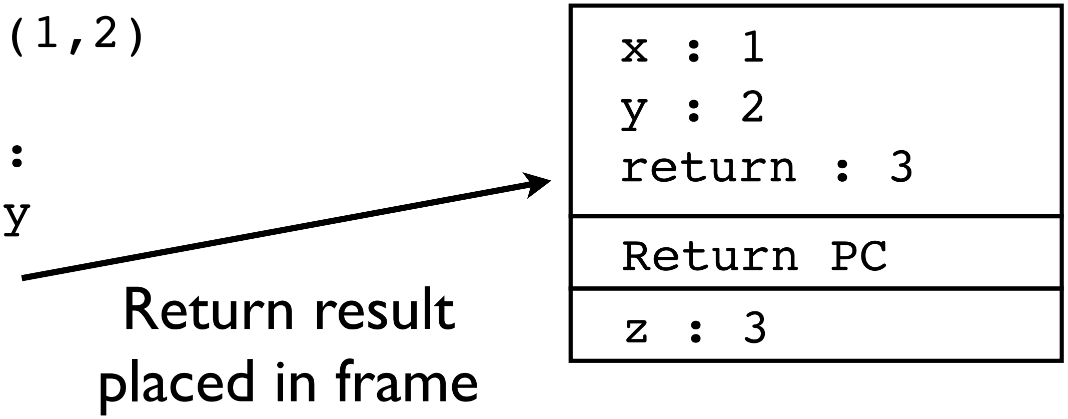
Local variables get added to the frame by the callee

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```



Return result placed in frame

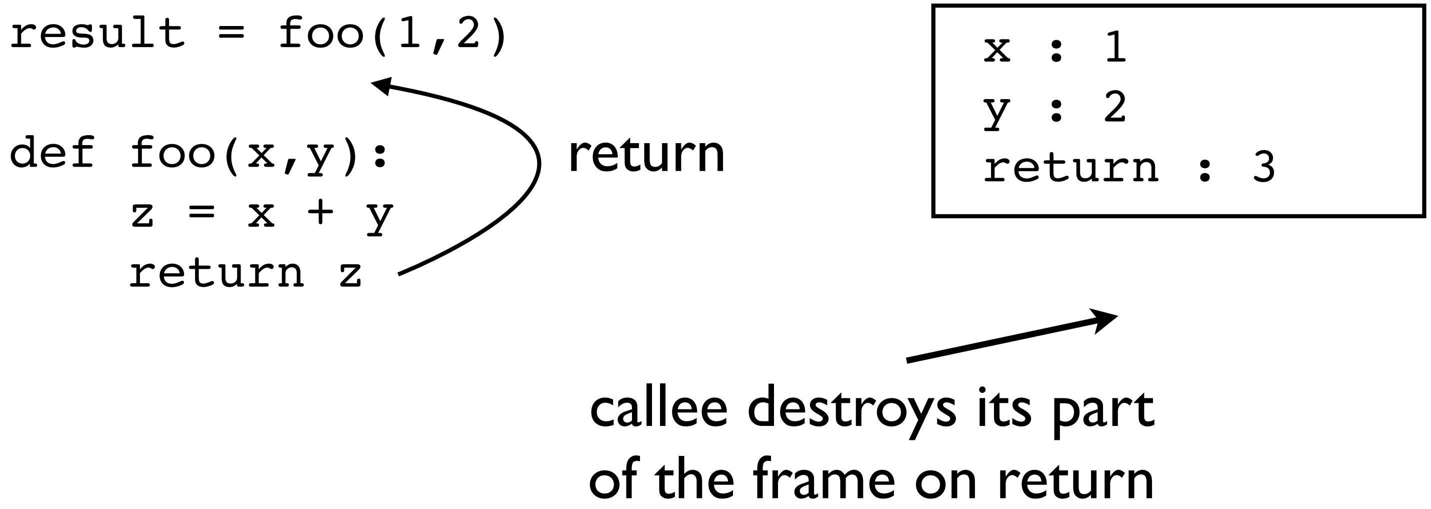
```
x : 1
y : 2
return : 3
```

```
Return PC
```

```
z : 3
```

Frame Management

- Management of Activation Frames is managed by both the caller and callee



```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

return

```
x : 1
y : 2
return : 3
```

callee destroys its part of the frame on return

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

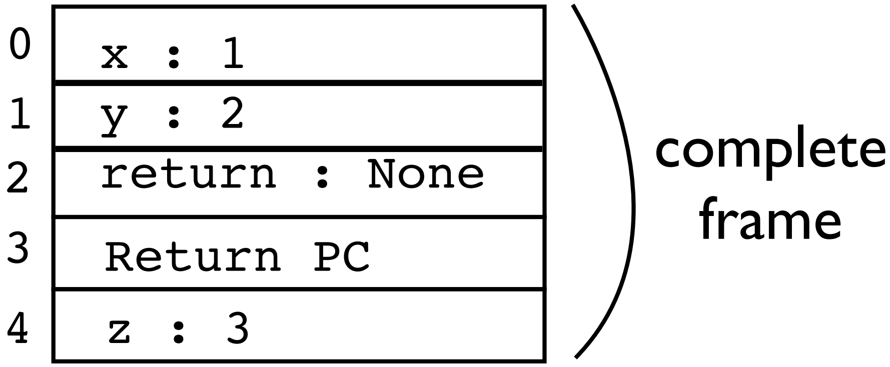
caller destroys
remaining frame on
assignment of result

Frame Management

- Implementation Detail : Frame often organized as an array of numeric "slots"

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```



0	x : 1
1	y : 2
2	return : None
3	Return PC
4	z : 3

- Slot numbers used in low-level instructions

- Determined at compile-time

Frame Example

- Python Disassembly

```
def foo(x,y):
    z = x + y
    return z

>>> import dis
>>> dis.dis(foo)
  2           0 LOAD_FAST                0 (x)
              3 LOAD_FAST                1 (y)
              6 BINARY_ADD
              7 STORE_FAST               2 (z)

  3          10 LOAD_FAST                2 (z)
             13 RETURN_VALUE
>>>
```

numbers refer to "slots" in the activation frame

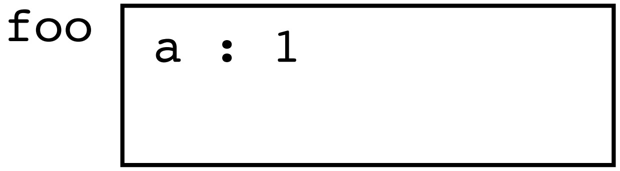
Tail Call Optimization

- Sometimes the compiler can eliminate frames

```
def foo(a):
    ...
    return bar(a-1)

def bar(a):
    ...
    return result

foo(1)
```



foo

a : 1

compiler detects that no more statements follow

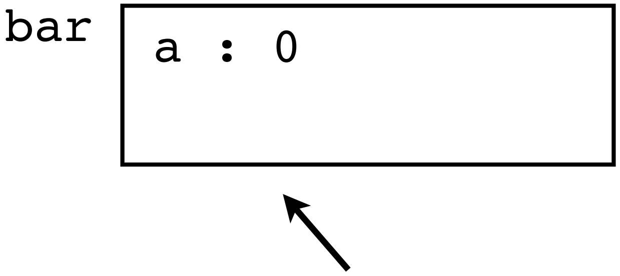
Tail Call Optimization

- Sometimes the compiler can eliminate frames

```
def foo(a):
    ...
    return bar(a-1)

def bar(a):
    ...
    return result

foo(1)
```



compiler reuses the same stack frame and just jumps to the next procedure (goto)

- Note: Python does not do this (although people often wish that it did)

Closures

- Nested functions are "interesting"

```
def add(x):
    def f(y):
        return x + y
    return f
```

- Example:

```
>>> a = add(2)
>>> a(3)
5
>>>
```

- The "x" variable must live someplace

- It does not exist on the stack.

Closures

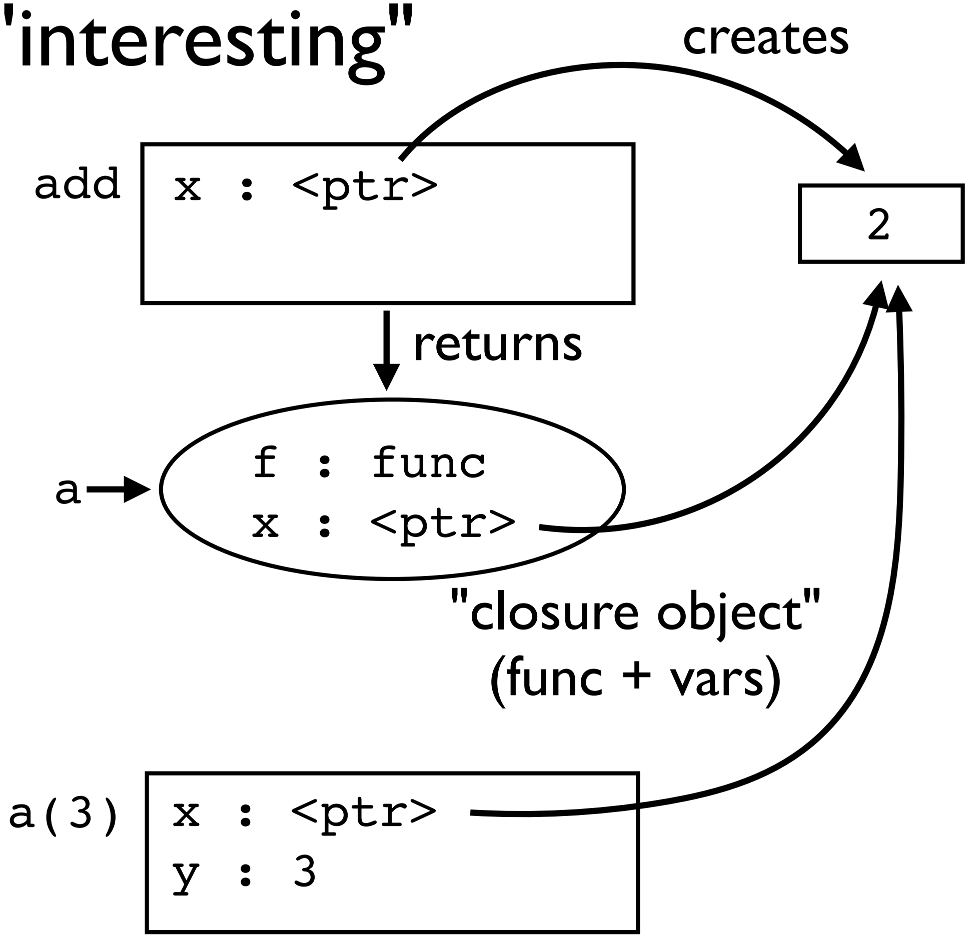
- Nested functions are "interesting"

```
def add(x):
    def f(y):
        return x + y
    return f
```

- Example:

```
>>> a = add(2)
>>> a(3)
5
>>>
```

- Indirect reference to a value stored "off stack"



ABIs

- Application Binary Interface
- A precise specification of function/procedure call semantics related to activation frames
- Language agnostic
- Critical part of creating programming libraries, DLLs, modules, etc.
- Different than an API (higher level)