Administrivia

Project 1 is due **tonight** by 11:59PM.

Project 2 is out and is due Monday 4/22 by 11:59PM.

Discussion sections are office hours (except A6, A4 is in CDS 907)

The final exam for this course is Wednesday 5/08 3–5PM in STO B50 (this room).

If you need accommodations for the exam, please send me an email ASAP.

Subroutines III: Activation Records and Closures

Principles of Programming Languages
Lecture 22

CAS CS 320

Objectives

Understand the relationship between **subroutines** and **lexically scoped** variable bindings.

Discuss the **call stack** as a construct for managing the execution of subroutines.

Look at how subroutines can **capture variables** in a language with higher-order functions and nested definitions.

Discuss **closures** as a construct for managing captured variables.

Keywords

formal parameters vs. actual parameters

local vs. global variables

nested vs. enclosing subroutines

scope

environment

activation records

call stack

Practice Problem

```
let f x y =
  let y = x + x in
  let x = x + y in
  x

let g () =
  let _ = print_endline "calling g..." in
  0

let _ = f (g ()) (g ())
```

What does this program print under call-by-value evaluation?
What does it print under call-by-name evaluation?

Answer

Call-by-value:

```
calling g...
calling g...
```

Call-by-name:

```
calling g...
calling g...
calling g...
```

```
let f x y =
  let y = x + x in
  let x = x + y in
  x

let g () =
  let _ = print_endline "calling g..." in
  0

let _ = f (g ()) (g ())
```

Recap/Review

Recall: Dynamic Scoping

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Recall: Dynamic Scoping

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Dynamic scoping refers to the idea that variables bindings are determined at run time based on the computational context.

Recall: Dynamic Scoping

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Dynamic scoping refers to the idea that variables bindings are determined at run time based on the computational context.

In its simplest form, dynamic scoping uses a global environment and *any* binding may be referred to *anywhere* in the program.

Recall: Dynamic Scoping

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Dynamic scoping refers to the idea that variables bindings are determined at run time based on the computational context.

In its simplest form, dynamic scoping uses a global environment and *any* binding may be referred to *anywhere* in the program.

This is a temporal view, i.e., was there a computation done beforehand which affected the value of a variable?

Recall: Dynamic Scoping

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Dynamic scoping refers to the idea that variables bindings are determined at run time based on the computational context.

In its simplest form, dynamic scoping uses a global environment and *any* binding may be referred to *anywhere* in the program.

This is a temporal view, i.e., was there a computation done beforehand which affected the value of a variable?

(It is uncommon in modern programming languages, but easier to implement)

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping refers the use of textual delimiters to define the scope of a binding

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping refers the use of textual delimiters to define the scope of a binding

A binding may be referred to within the delimited textual area of the code

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping refers the use of textual delimiters to define the scope of a binding

A binding may be referred to within the delimited textual area of the code

This is also called static scoping because, in theory, scoping errors can be found before the program is run

Recall: Lexical Scoping

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping refers the use of textual delimiters to define the scope of a binding

A binding may be referred to within the delimited textual area of the code

This is also called static scoping because, in theory, scoping errors can be found before the program is run

(This is far more common in modern programming languages)

Recall: Restricting Scope

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

function scope

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

scope of x

(OCaml)

Recall: Restricting Scope

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

function scope

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

scope of x

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

Recall: Restricting Scope

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

function scope

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

scope of x

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

» The binding defines its own scope
  (e.g. let-bindings, Project 3)

Recall: Restricting Scope

```python
x = 0
def f():
    x = 1
    return(x)
assert(f() == 1)
assert(x == 0)
```

function scope

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

scope of x

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

» The binding defines its own scope
  (e.g. let-bindings, Project 3)

» A subroutine or code block defines the scope of the bindings appearing in it
  (e.g. python function, Project 2)

Anatomy of a Function

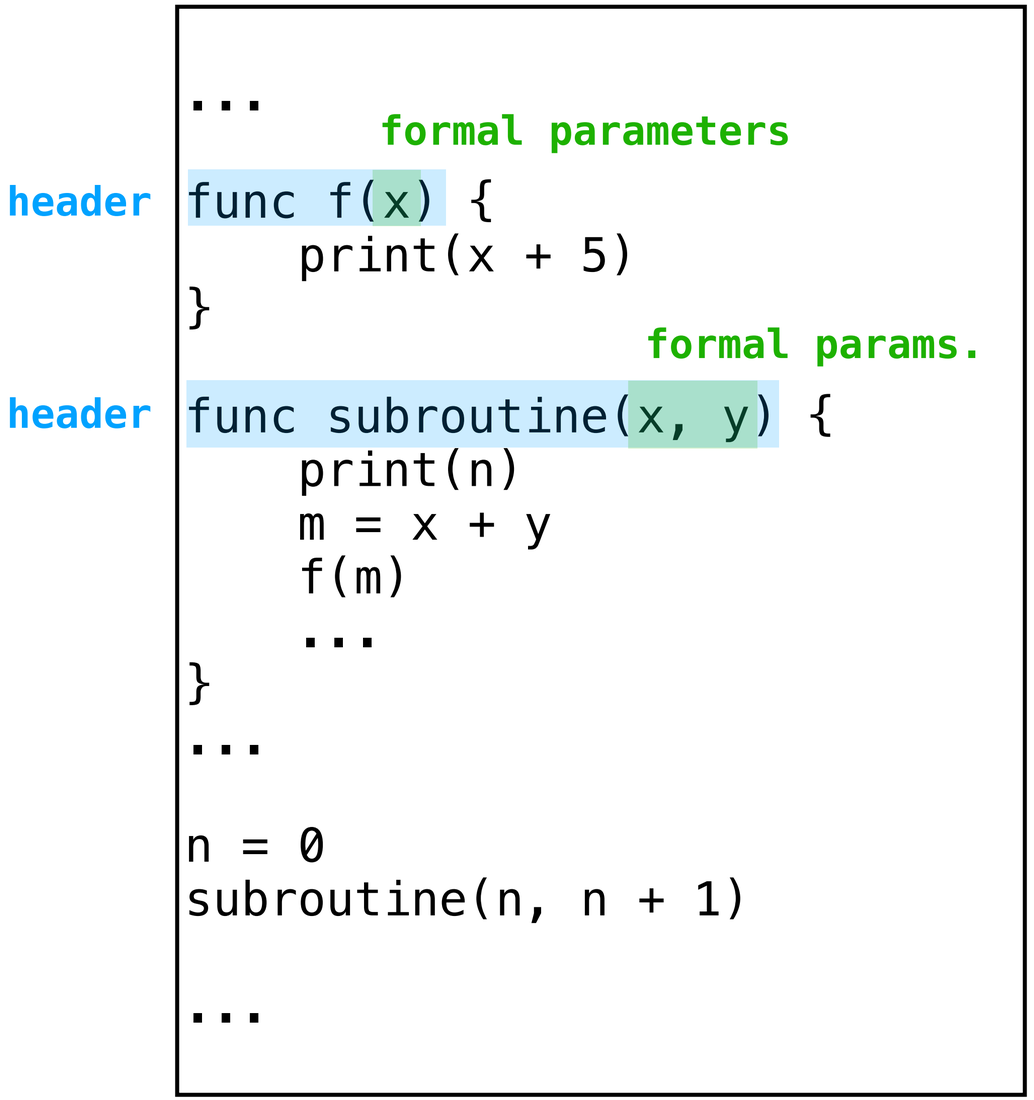
```
...

func f(x) {
    print(x + 5)
}

func subroutine(x, y) {
    print(n)
    m = x + y
    f(m)
    ...
}
...

n = 0
subroutine(n, n + 1)

...
```

Anatomy of a Function

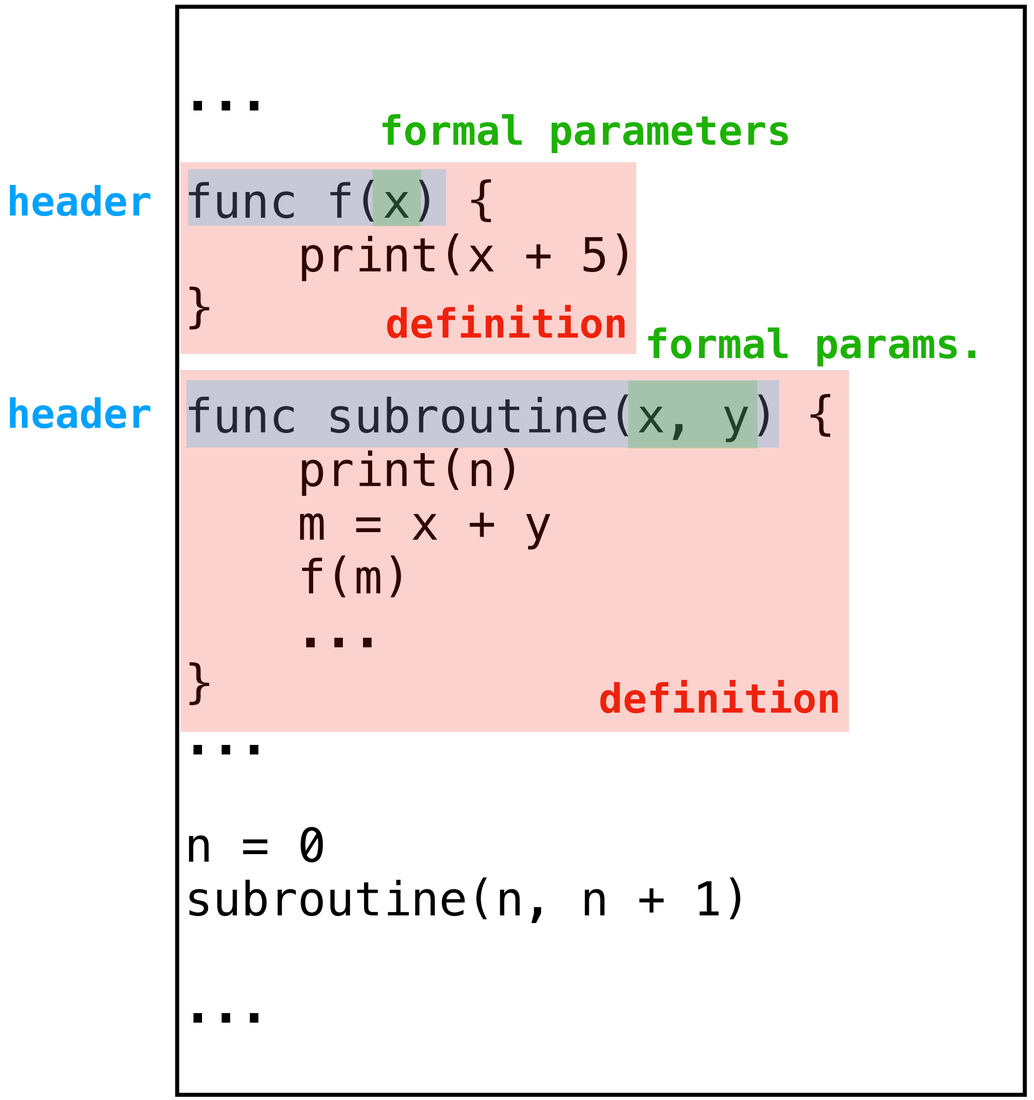
header
header

```
...

func f(x) {
    print(x + 5)
}

func subroutine(x, y) {
    print(n)
    m = x + y
    f(m)
    ...
}
...

n = 0
subroutine(n, n + 1)

...
```

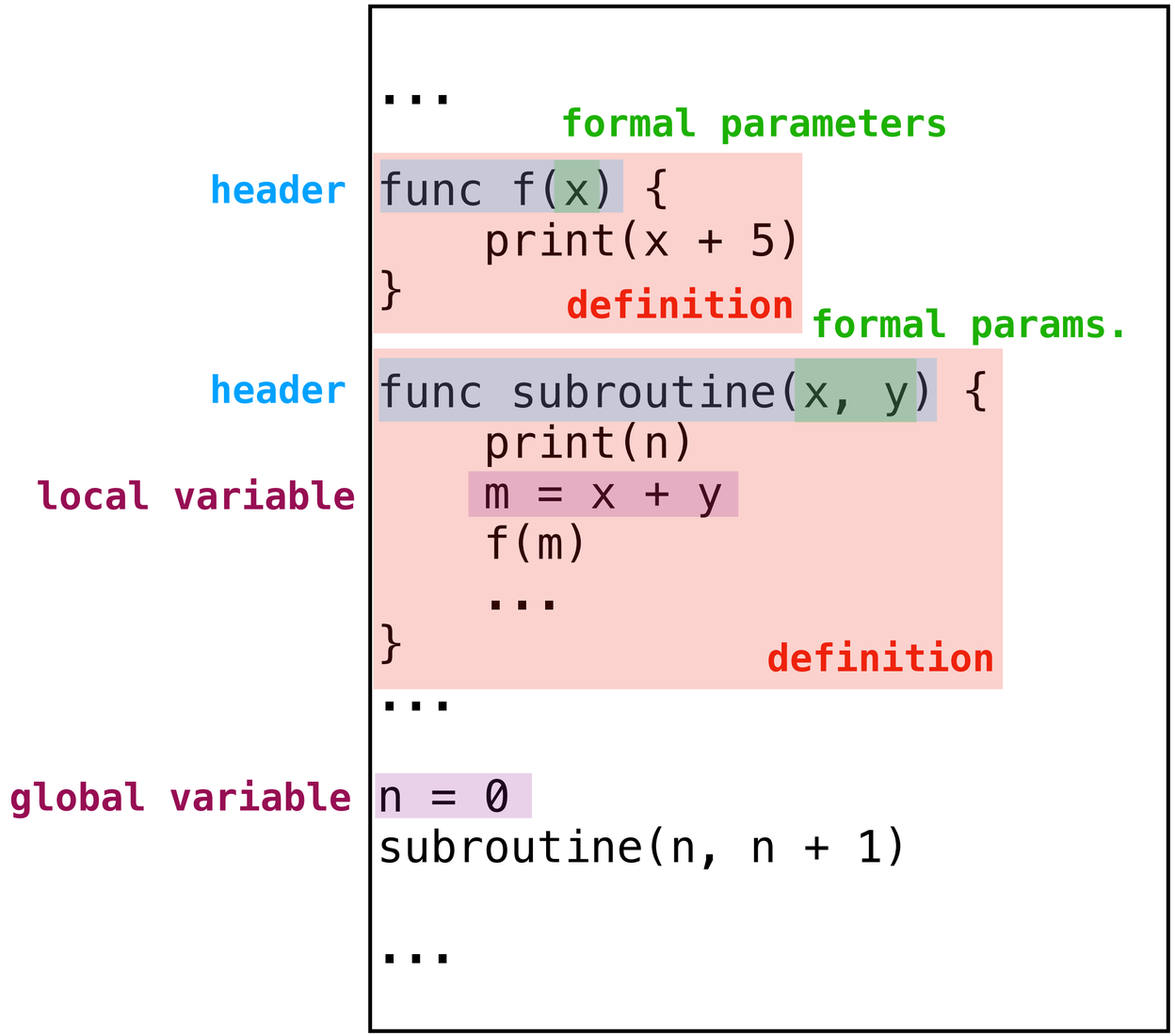
Anatomy of a Function



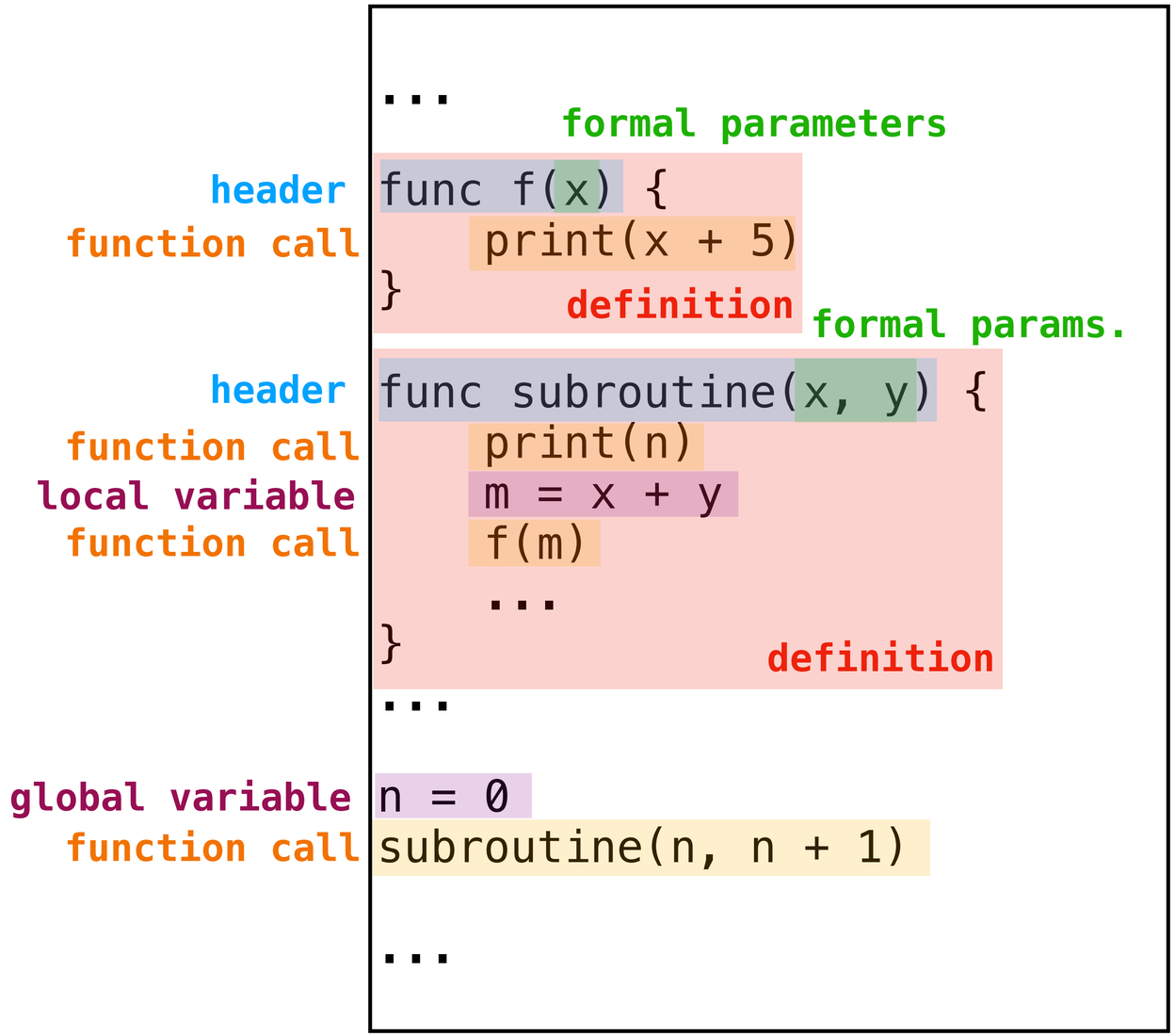
Anatomy of a Function



Anatomy of a Function



Anatomy of a Function



Anatomy of a Function

function call

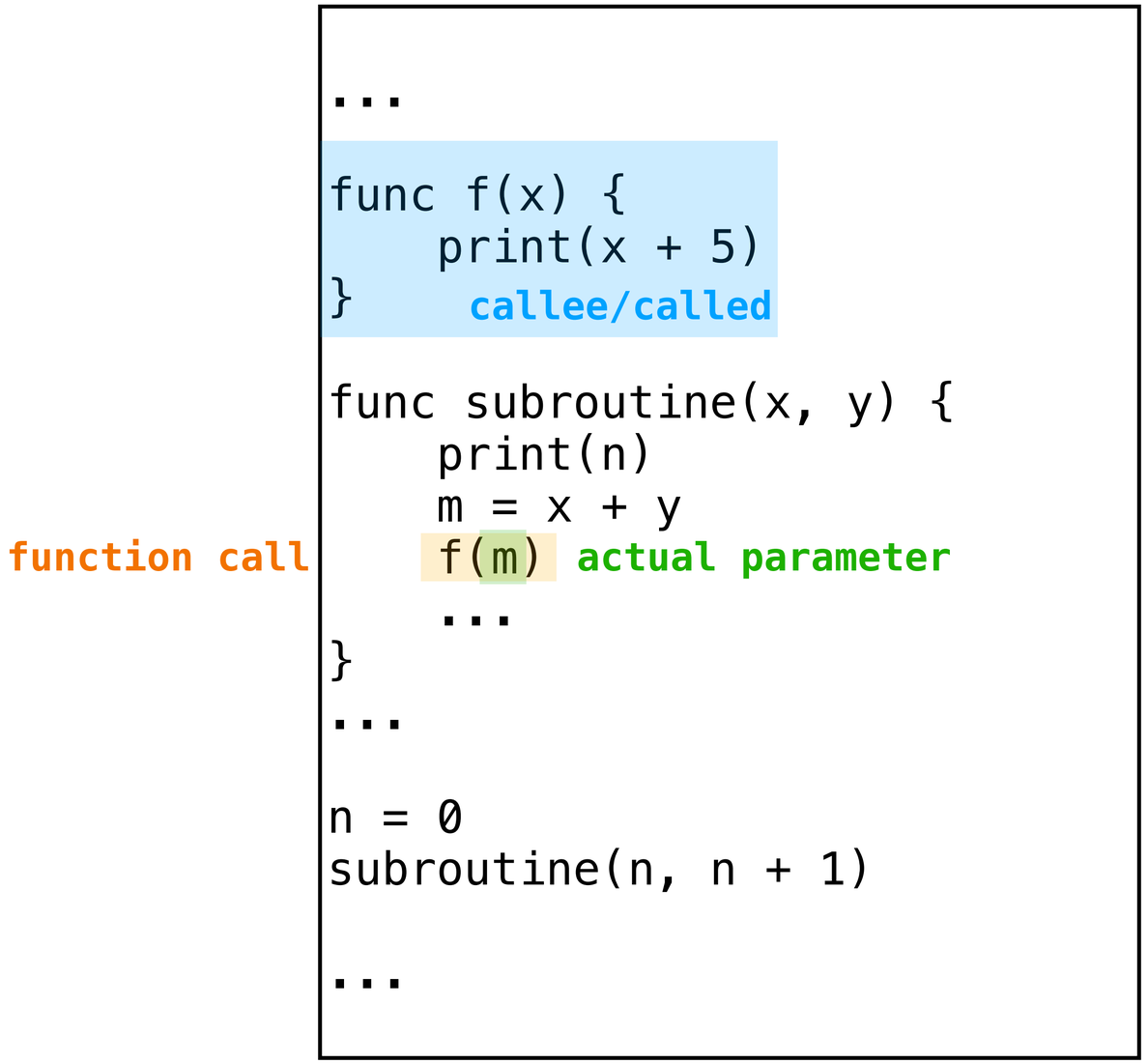
```
...

func f(x) {
    print(x + 5)
}

func subroutine(x, y) {
    print(n)
    m = x + y
    f(m)
    ...
}
...

n = 0
subroutine(n, n + 1)

...
```

Anatomy of a Function

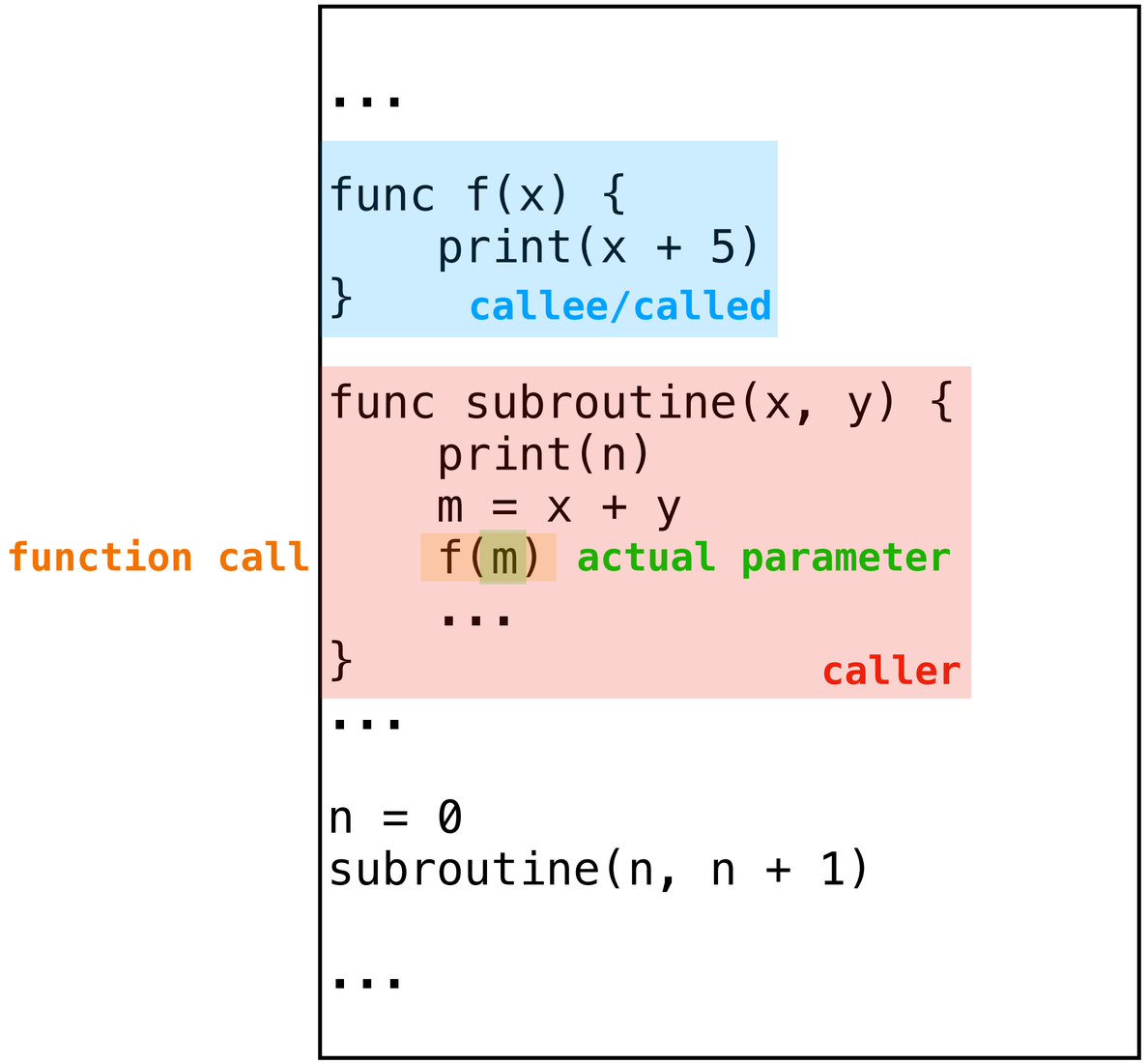
function call

```
...

func f(x) {
    print(x + 5)
}

func subroutine(x, y) {
    print(n)
    m = x + y
    f(m) actual parameter
    ...
}
...

n = 0
subroutine(n, n + 1)

...
```

Anatomy of a Function



Anatomy of a Function



Anatomy of a Function

function call

```
...

func f(x) {
    print(x + 5)
}

func subroutine(x, y) {
    print(n)
    m = x + y
    f(m)
    ...
}
...

n = 0
subroutine(n, n + 1)

...
```

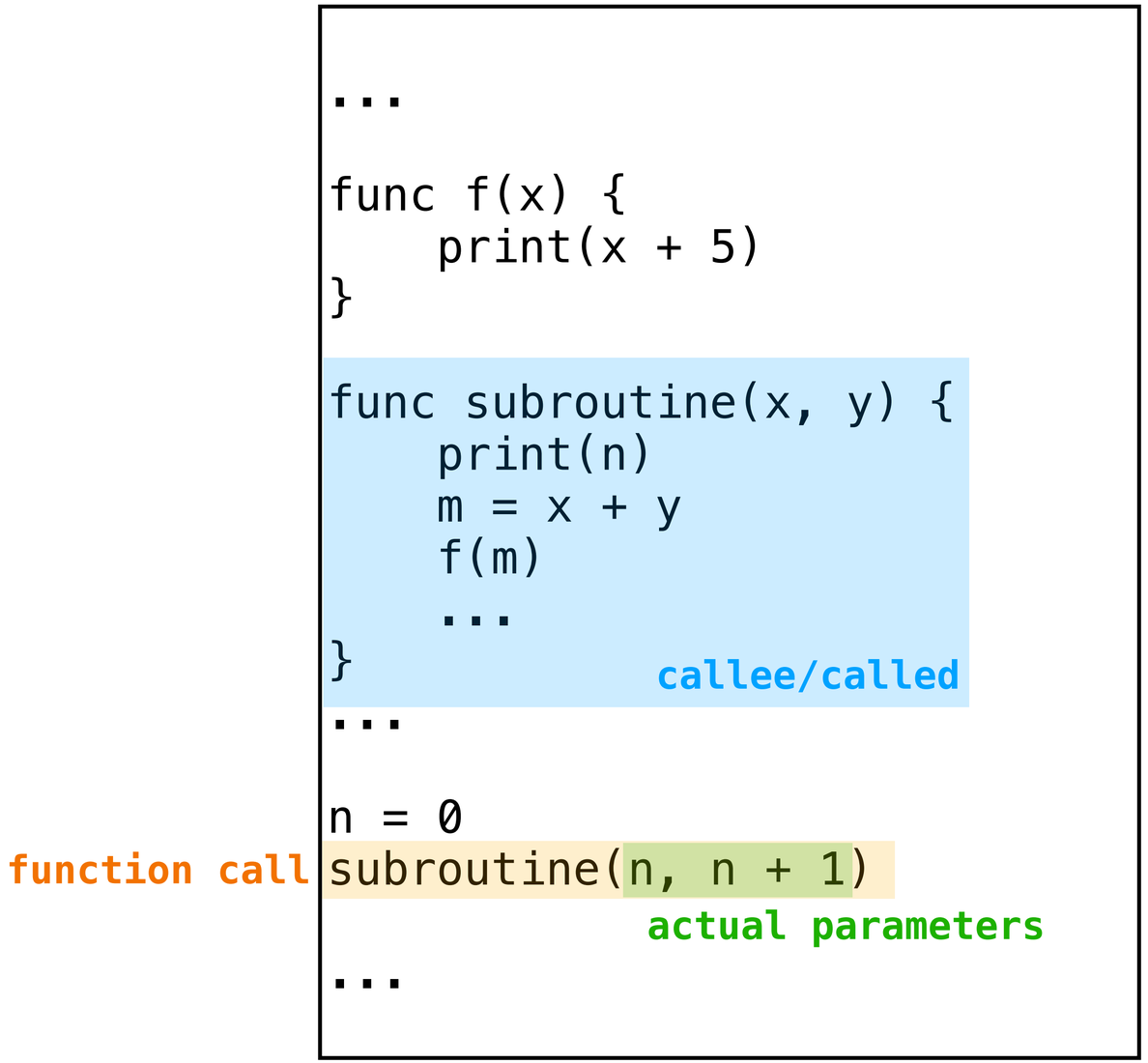
Anatomy of a Function

```
...

func f(x) {
    print(x + 5)
}

func subroutine(x, y) {
    print(n)
    m = x + y
    f(m)
    ...
}
...

n = 0
subroutine(n, n + 1)
...
```

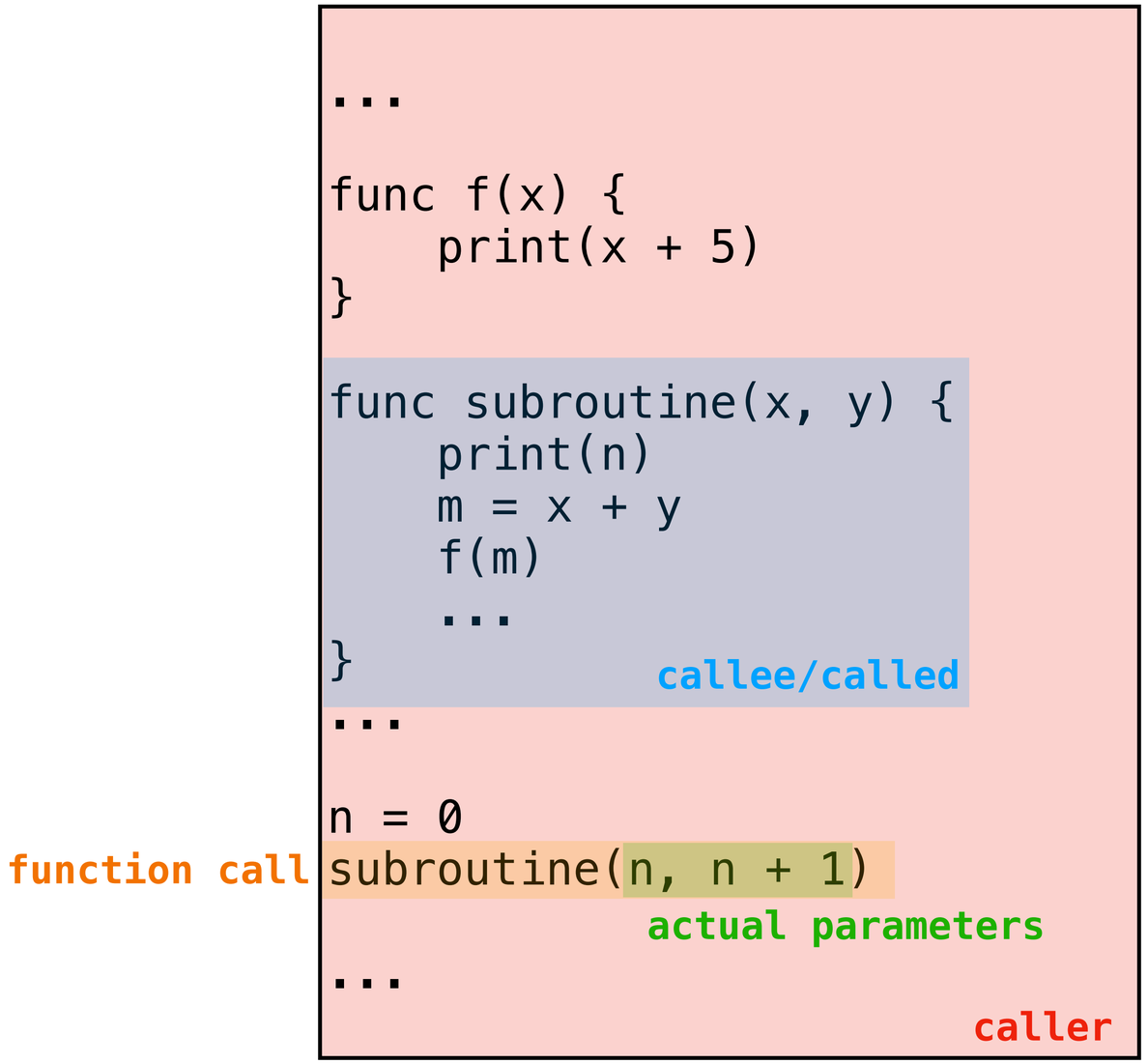
function call

actual parameters

Anatomy of a Function



Anatomy of a Function



Anatomy of a Function

```
...
func f(x) {
    n = 3
    func subroutine(x, y) {
        print(n)
        m = x + y
        f(m)
    }
    print(x + 5)
}
...
```

Anatomy of a Function

```
...
func f(x) {
    n = 3
    func subroutine(x, y) {
        print(n)
        m = x + y
        f(m)
    }
    print(x + 5)
}
...
```

nested subroutine

Anatomy of a Function

enclosing subroutine

```
...
func f(x) {
    n = 3
    func subroutine(x, y) {
        print(n)
        m = x + y
        f(m)
    }
    print(x + 5)
}
...
```

nested subroutine

Considerations for Implementing Functions

Considerations for Implementing Functions

How are parameters passed to our function? By value? By name? By reference?

Considerations for Implementing Functions

How are parameters passed to our function? By value? By name? By reference?

What variables are in scope when a function is called? Are they mutable?

Considerations for Implementing Functions

How are parameters passed to our function? By value? By name? By reference?

What variables are in scope when a function is called? Are they mutable?

What information do we to execute a function?

Considerations for Implementing Functions

How are parameters passed to our function? By value? By name? By reference?

What variables are in scope when a function is called? Are they mutable?

What information do we to execute a function?

How does a function refer to its environment?

Considerations for Implementing Functions

How are parameters passed to our function? By value? By name? By reference?

What variables are in scope when a function is called? Are they mutable?

What information do we to execute a function?

How does a function refer to its environment?

Are we allowed to nest function definitions?

Considerations for Implementing Functions

How are parameters passed to our function? By value? By name? By reference?

What variables are in scope when a function is called? Are they mutable?

What information do we to execute a function?

How does a function refer to its environment?

Are we allowed to nest function definitions?

Are we allowed to return functions?

Project 2 Overview

Project 2 Overview

» *Call-by-Value Parameter Passing*

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.

Project 2 Overview

» *Call-by-Value Parameter Passing*

  » Parameters are passed on a separate stack as values. Same with return values.

» *Lexically Scoping*

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.
- *Lexically Scoping*
  - Bindings not within a function definition have *global scope*, they are accessible everywhere in the program

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.
- *Lexically Scoping*
  - Bindings not within a function definition have *global scope*, they are accessible everywhere in the program
  - Bindings within a function definition are given *function scope*, they are accessible only within the function

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.
- *Lexically Scoping*
  - Bindings not within a function definition have *global scope*, they are accessible everywhere in the program
  - Bindings within a function definition are given *function scope*, they are accessible only within the function
- *Nonlocal Fetches, Local Updates*

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.
- *Lexically Scoping*
  - Bindings not within a function definition have *global scope*, they are accessible everywhere in the program
  - Bindings within a function definition are given *function scope*, they are accessible only within the function
- *Nonlocal Fetches, Local Updates*
  - All bindings in scope are accessible for fetching

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.
- *Lexically Scoping*
  - Bindings not within a function definition have *global scope*, they are accessible everywhere in the program
  - Bindings within a function definition are given *function scope*, they are accessible only within the function
- *Nonlocal Fetches, Local Updates*
  - All bindings in scope are accessible for fetching
  - Only local bindings are accessible to mutable updating. **Updates shadow nonlocal bindings in scope**

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.
- *Lexically Scoping*
  - Bindings not within a function definition have *global scope*, they are accessible everywhere in the program
  - Bindings within a function definition are given *function scope*, they are accessible only within the function
- *Nonlocal Fetches, Local Updates*
  - All bindings in scope are accessible for fetching
  - Only local bindings are accessible to mutable updating. **Updates shadow nonlocal bindings in scope**
- *First-Class Functions*

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.
- *Lexically Scoping*
  - Bindings not within a function definition have *global scope*, they are accessible everywhere in the program
  - Bindings within a function definition are given *function scope*, they are accessible only within the function
- *Nonlocal Fetches, Local Updates*
  - All bindings in scope are accessible for fetching
  - Only local bindings are accessible to mutable updating. **Updates shadow nonlocal bindings in scope**
- *First-Class Functions*
  - function definitions may be nested

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.
- *Lexically Scoping*
  - Bindings not within a function definition have *global scope*, they are accessible everywhere in the program
  - Bindings within a function definition are given *function scope*, they are accessible only within the function
- *Nonlocal Fetches, Local Updates*
  - All bindings in scope are accessible for fetching
  - Only local bindings are accessible to mutable updating. **Updates shadow nonlocal bindings in scope**
- *First-Class Functions*
  - function definitions may be nested
  - functions may be returned by other functions, which may capture variables

Project 2 Overview

- *Call-by-Value Parameter Passing*
  - Parameters are passed on a separate stack as values. Same with return values.
- *Lexically Scoping*
  - Bindings not within a function definition have *global scope*, they are accessible everywhere in the program
  - Bindings within a function definition are given *function scope*, they are accessible only within the function
- *Nonlocal Fetches, Local Updates*
  - All bindings in scope are accessible for fetching
  - Only local bindings are accessible to mutable updating. **Updates shadow nonlocal bindings in scope**
- *First-Class Functions*
  - function definitions may be nested
  - functions may be returned by other functions, which may capture variables

This is similar to Python

A Toy Language

```
<prog>  ::= { <com> }
<com>   ::= . | ▷ <ident> | <ident>
          | : <prog> ; | # | Return
          | <val>
<val>   ::= ...
<ident> ::= ...
```

Our Language

```
2 ▷ x
(def test): ▷ n
  x ▷ y
  n .
; ▷ test

3 test #
```

Python

```
x = 2
def test(n):
  y = x
  print(n)


test(3)
```

The following toy language is a fragment of the language from Project 2, dealing only with subroutines and variables.

(we will use this in several examples)

The Call Stack

Motivation

Motivation

What happens when a function is called? What information does a function need in order to be executed properly?

Motivation

What happens when a function is called? What information does a function need in order to be executed properly?

Intuition: When we run a program, we imagine running each line one by one until we get to a function call, when we jump into the function.

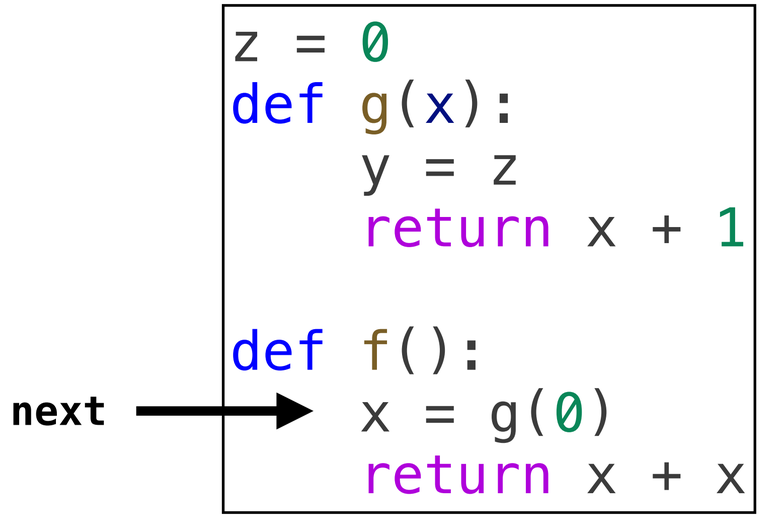
Motivation

What happens when a function is called? What information does a function need in order to be executed properly?

Intuition: When we run a program, we imagine running each line one by one until we get to a function call, when we jump into the function.

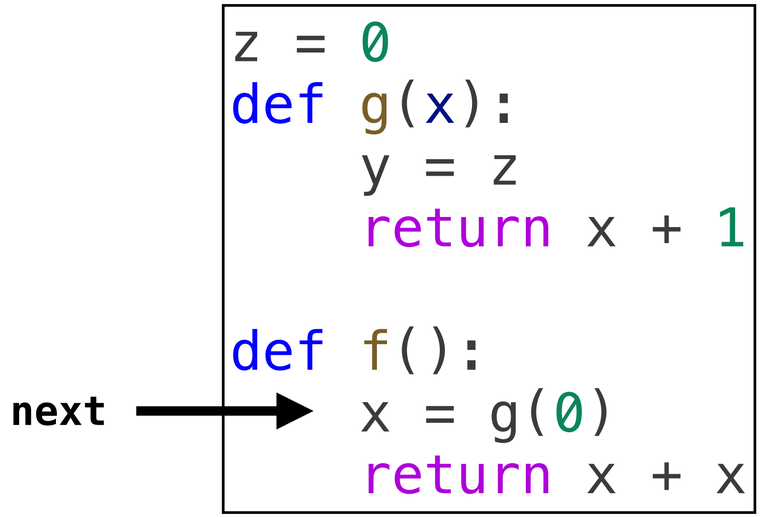
The call stack organizes the back-and-forth of jumping into and out of functions.

Before Transferring Control to the Callee



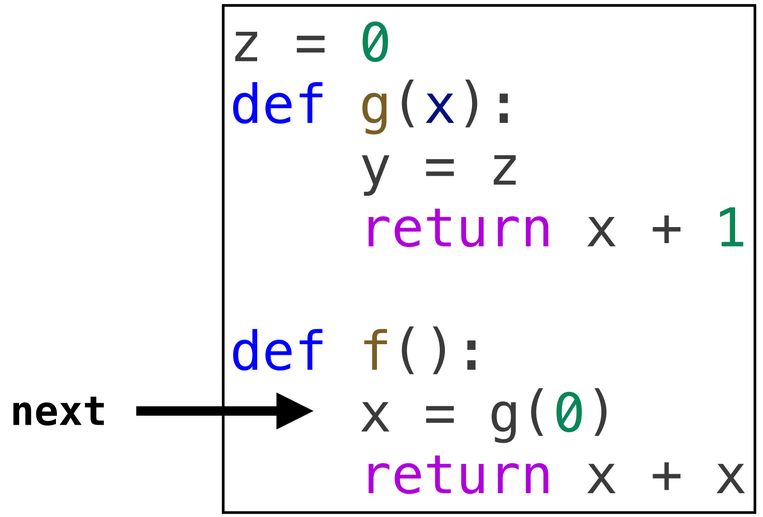
Before Transferring Control to the Callee

» create an **activation record** for the callee



Before Transferring Control to the Callee

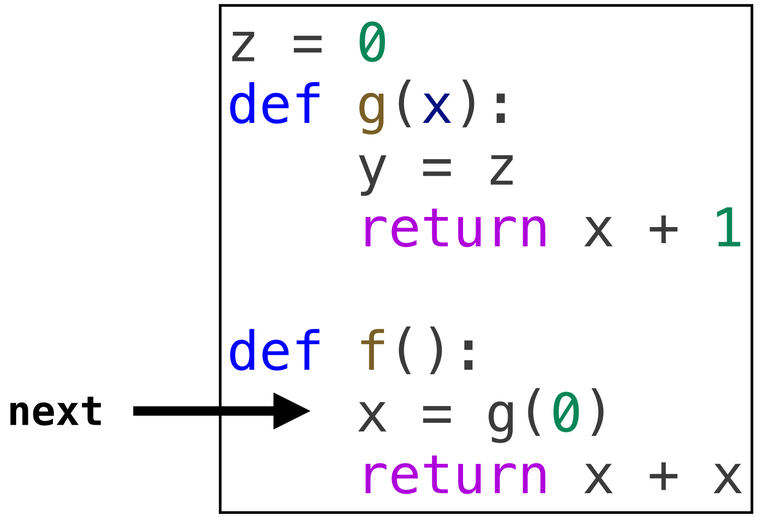
- create an **activation record** for the callee
  - *where do we put the info the callee needs to run?*



activation record for call to g

Before Transferring Control to the Callee

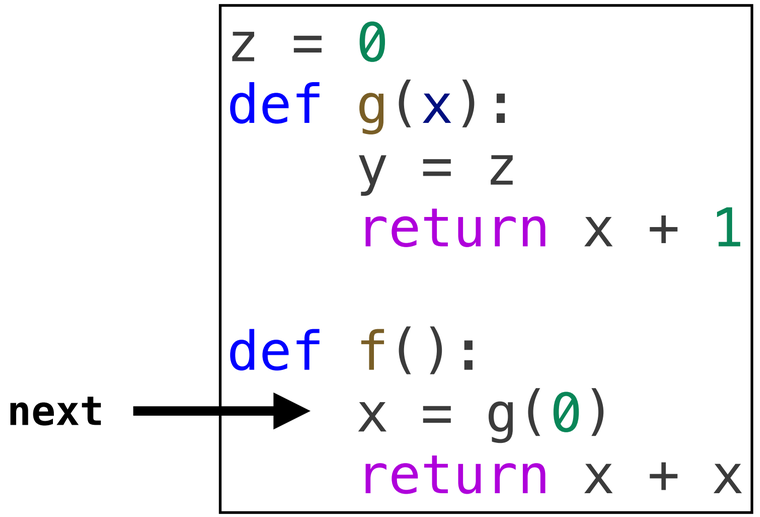
- create an **activation record** for the callee
  - *where do we put the info the callee needs to run?*
- link **formal parameters** with **actual parameters** (possibly within the activation record)



activation record for call to g

Before Transferring Control to the Callee

- create an **activation record** for the callee
  - *where do we put the info the callee needs to run?*
- link **formal parameters** with **actual parameters** (possibly within the activation record)
  - *what are we applying the function to?*

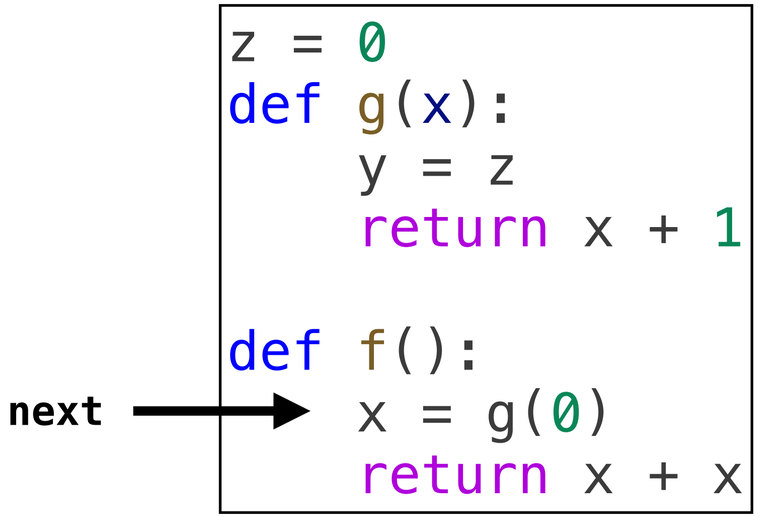


activation record for call to g

parameter x is 0

Before Transferring Control to the Callee

- create an **activation record** for the callee
  - *where do we put the info the callee needs to run?*
- link **formal parameters** with **actual parameters** (possibly within the activation record)
  - *what are we applying the function to?*
- save the **execution status** of the caller (possibly within the activation record)

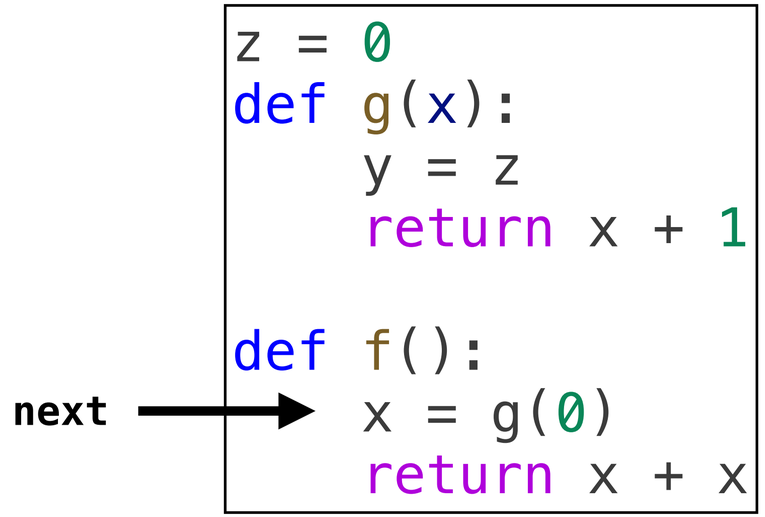


activation record for call to g

parameter x is 0

Before Transferring Control to the Callee

- create an **activation record** for the callee
  - *where do we put the info the callee needs to run?*
- link **formal parameters** with **actual parameters** (possibly within the activation record)
  - *what are we applying the function to?*
- save the **execution status** of the caller (possibly within the activation record)
  - *how much of the caller function have we already executed? Where to do we need to **return** to?*



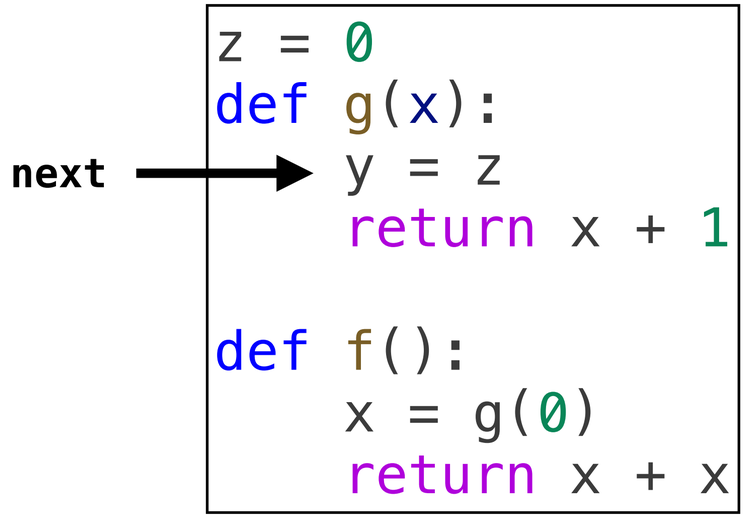
activation record for call to g

parameter x is 0

return to executing f when done

During Control

- Update local bindings in the activation record of the function
- Fetch bindings in the environment possibly by looking at the activation records of enclosing scopes



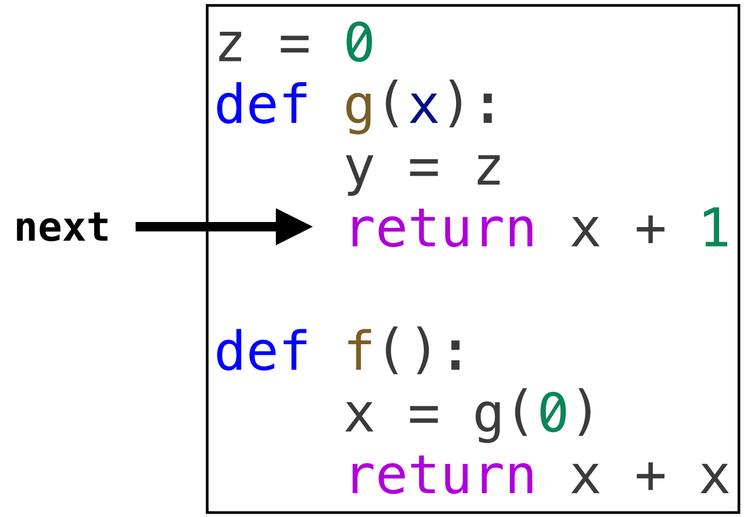
activation record for call to g

parameter x is 0

return to executing f when done

y is set to the value of z (0)

Before Returning Control to the Caller



activation record for call to g

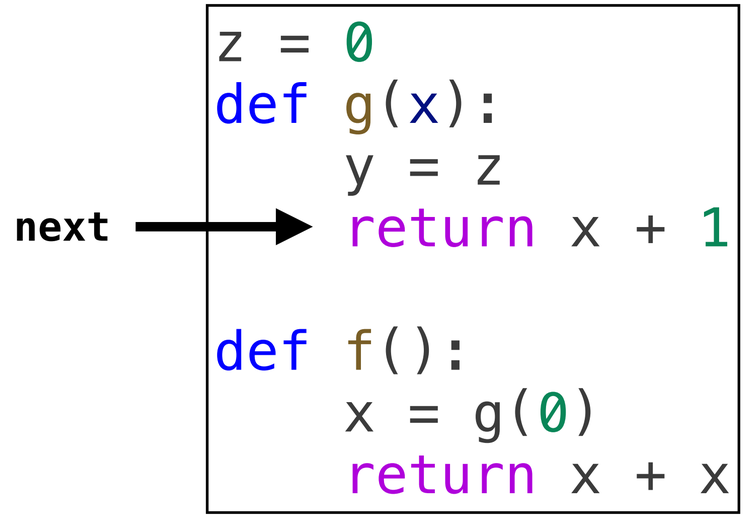
parameter x is 0

return to executing f when done

y is set to the value of z (0)

Before Returning Control to the Caller

» Set the return value of the function
  (possibly in the activation record)



activation record for call to g

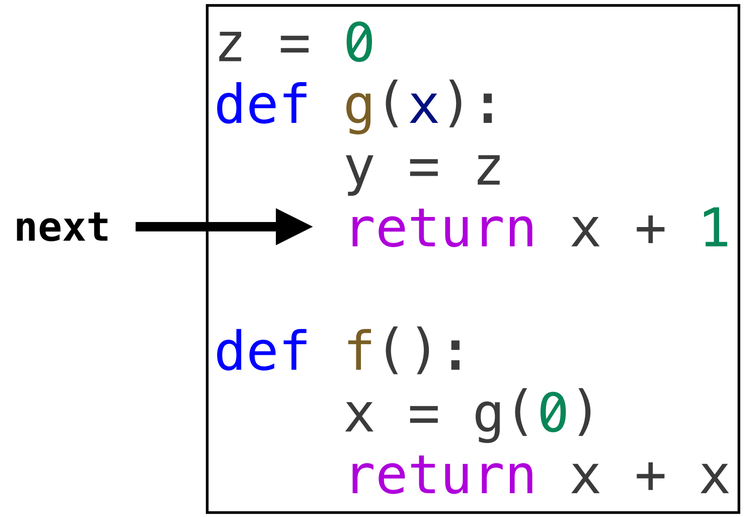
parameter x is 0

return to executing f when done

y is set to the value of z (0)

Before Returning Control to the Caller

- Set the return value of the function (possibly in the activation record)
  - *What value does this function return?*



activation record for call to g

parameter x is 0

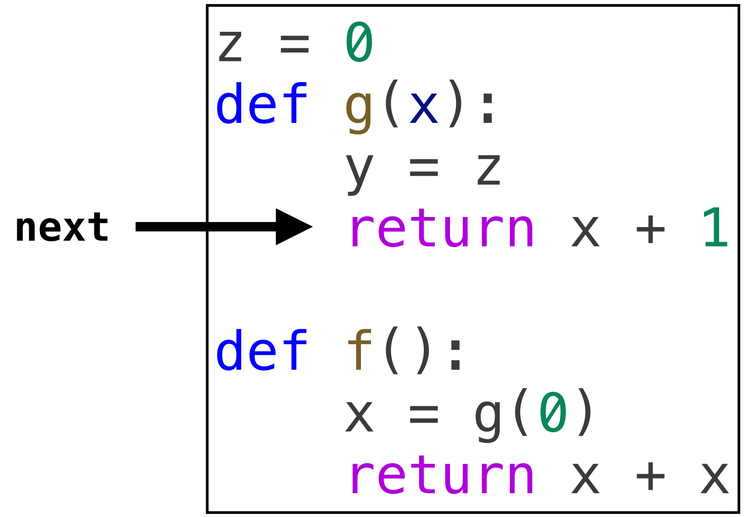
return to executing f when done

y is set to the value of z (0)

return the value 1

Before Returning Control to the Caller

- Set the return value of the function (possibly in the activation record)
  - *What value does this function return?*
- Discard the activation record of the callee



activation record for call to g

parameter x is 0

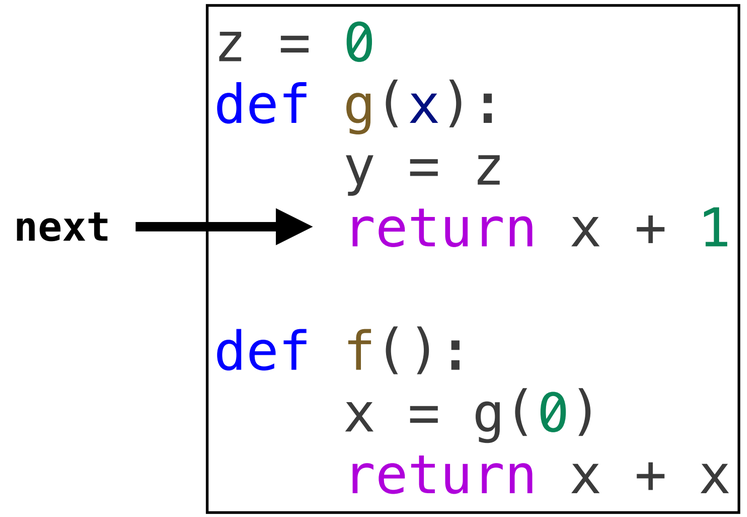
return to executing f when done

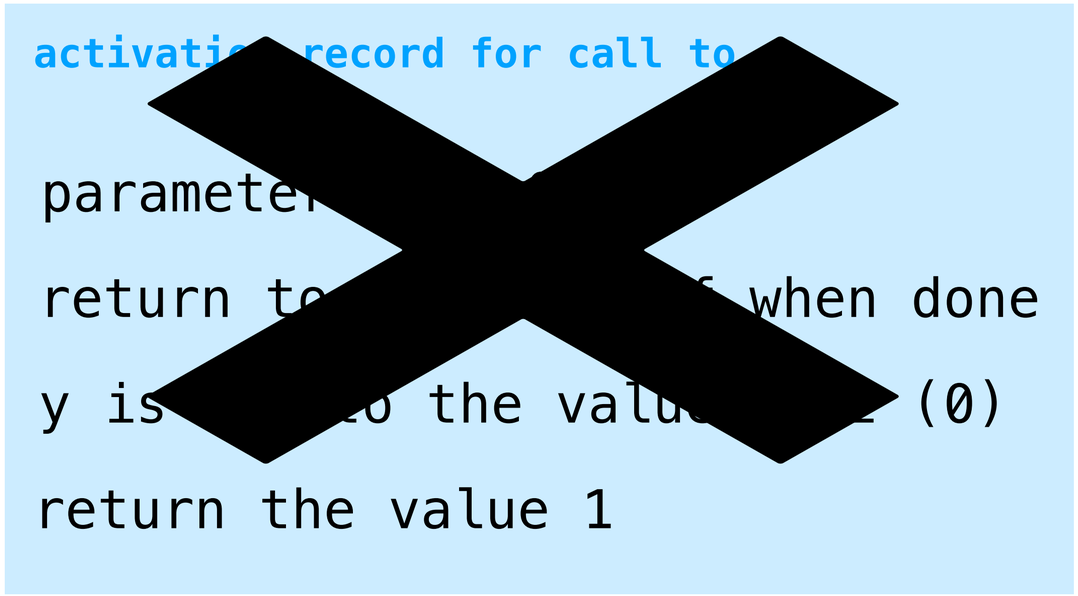
y is set to the value of z (0)

return the value 1

Before Returning Control to the Caller

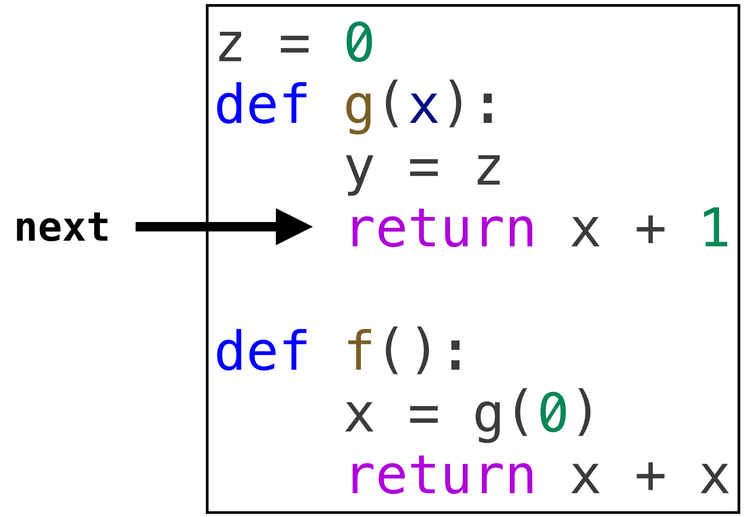
- Set the return value of the function (possibly in the activation record)
  - *What value does this function return?*
- Discard the activation record of the callee
  - *Make sure local variables are no longer in scope*

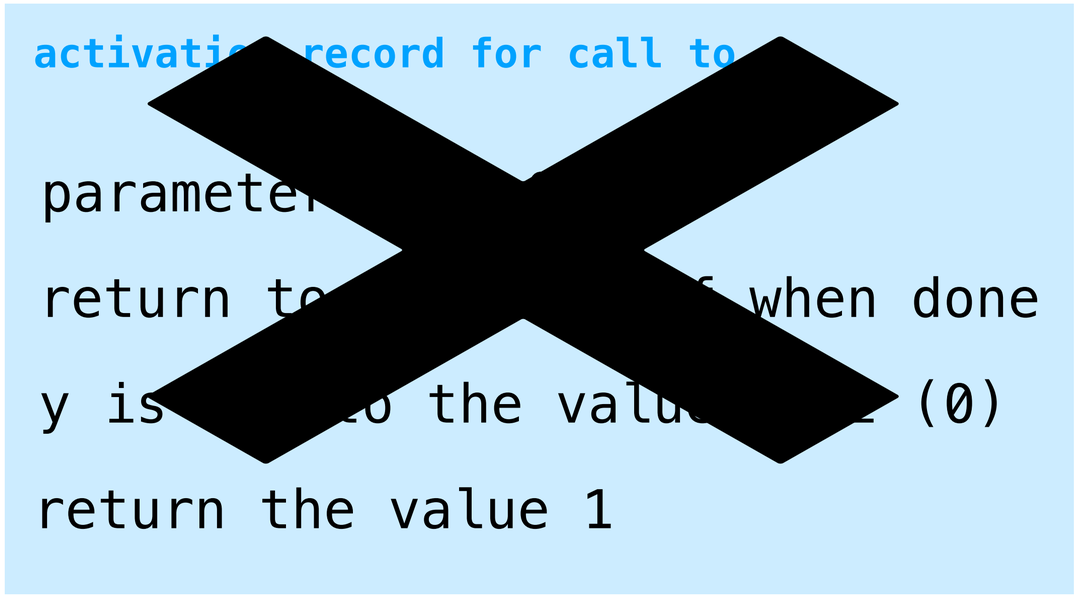




Before Returning Control to the Caller

- Set the return value of the function (possibly in the activation record)
  - *What value does this function return?*
- Discard the activation record of the callee
  - *Make sure local variables are no longer in scope*
- Restore the activation record of the caller



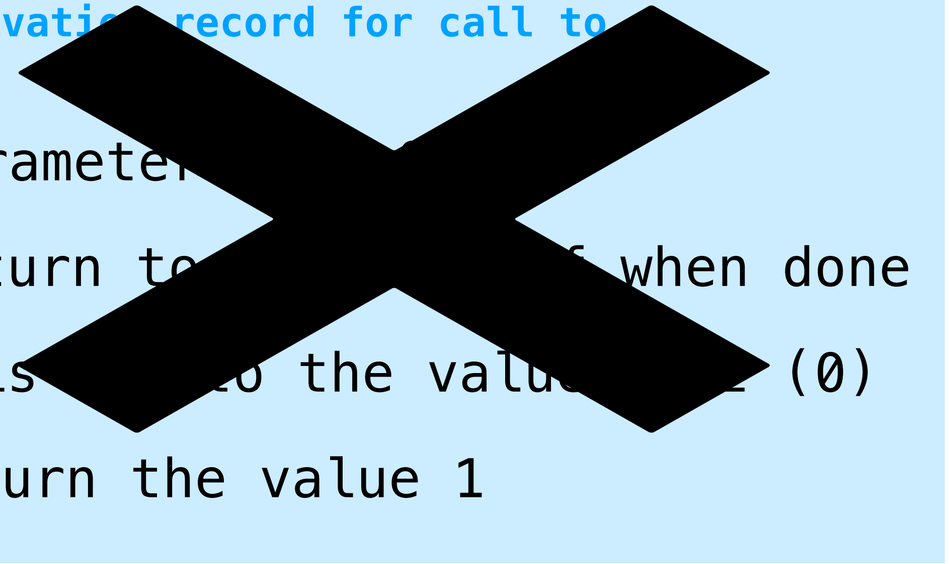


Before Returning Control to the Caller

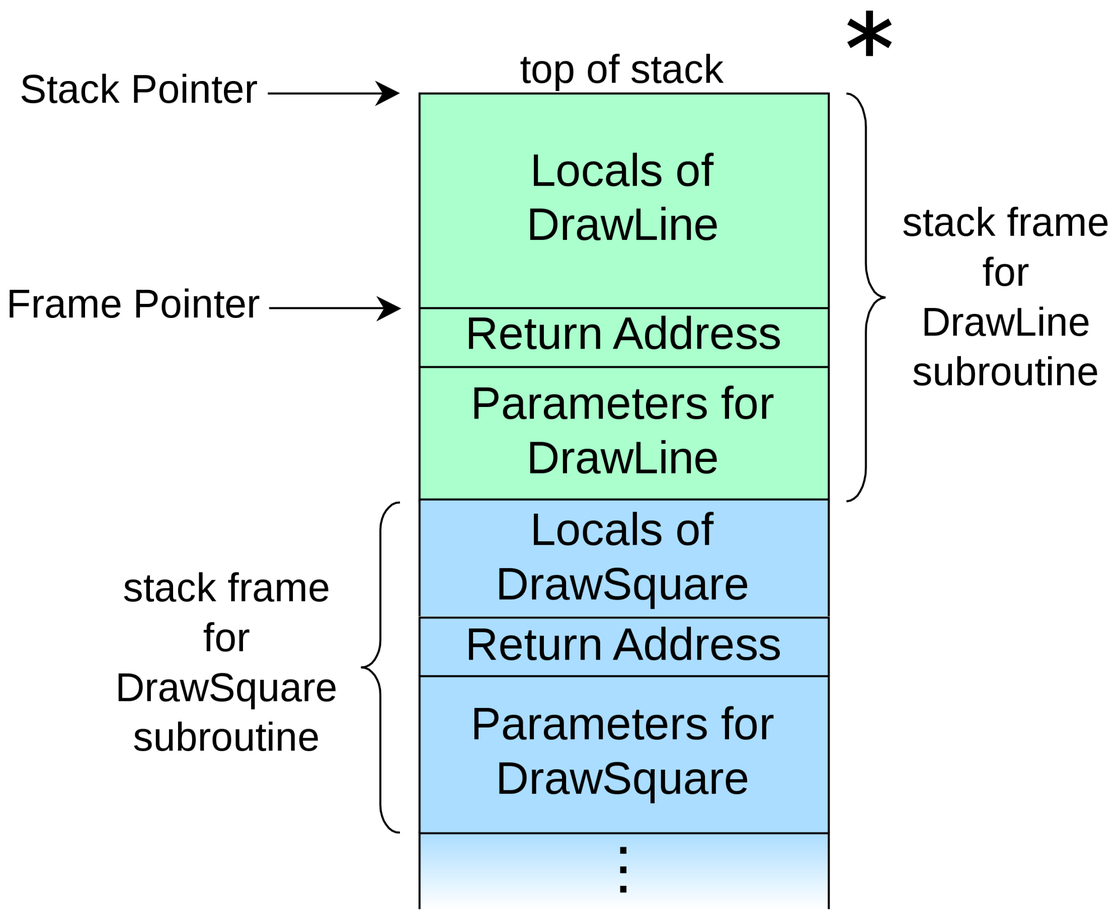
- Set the return value of the function (possibly in the activation record)
  - *What value does this function return?*
- Discard the activation record of the callee
  - *Make sure local variables are no longer in scope*
- Restore the activation record of the caller
  - *Continue running the code of the caller*

```
z = 0
def g(x):
    y = z
    return x + 1

def f():
    x = g(0)
    return x + x
```

next →



Activation Records

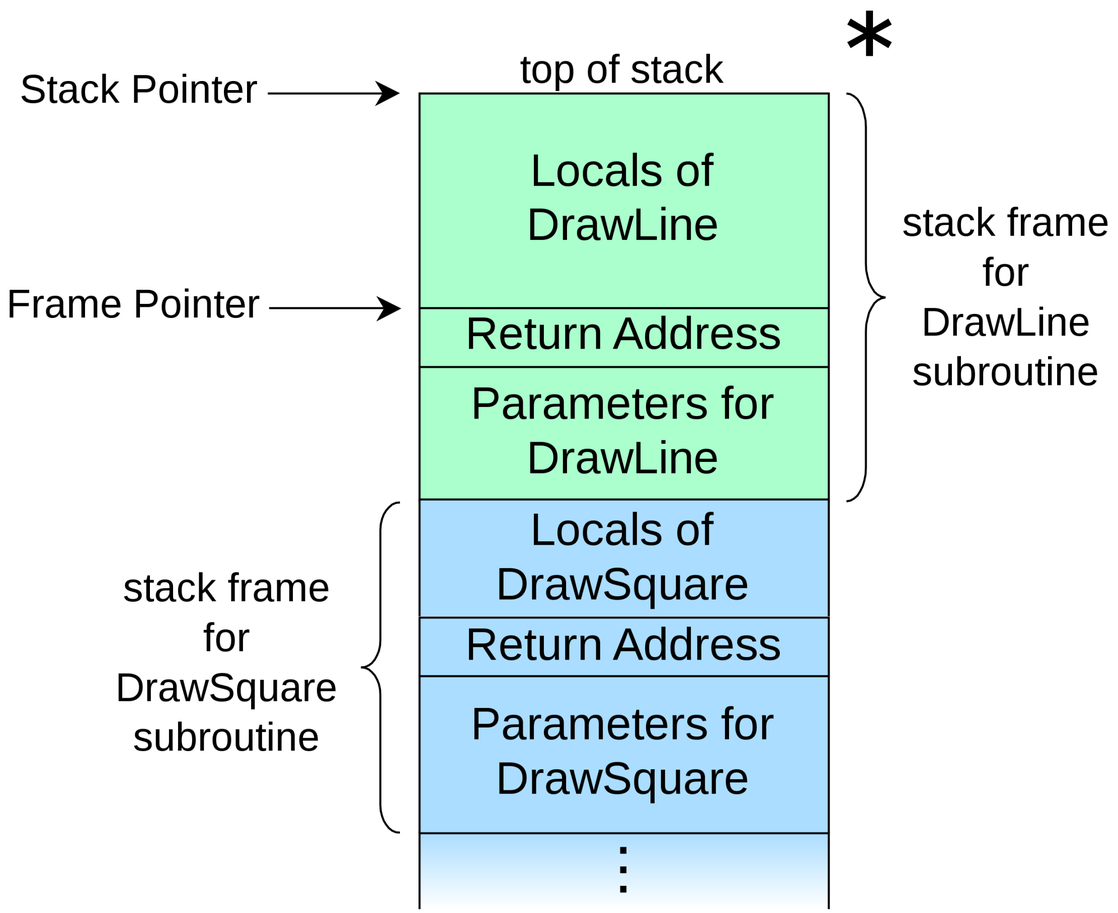


*since this is not a compilers course we will use this as a conceptual framework instead of an actual implementation guideline

https://commons.wikimedia.org/wiki/File:Call_stack_layout.svg

Activation Records

Also called a **stack frame**, an activation record contains the information necessary to execute a subroutine, e.g.,



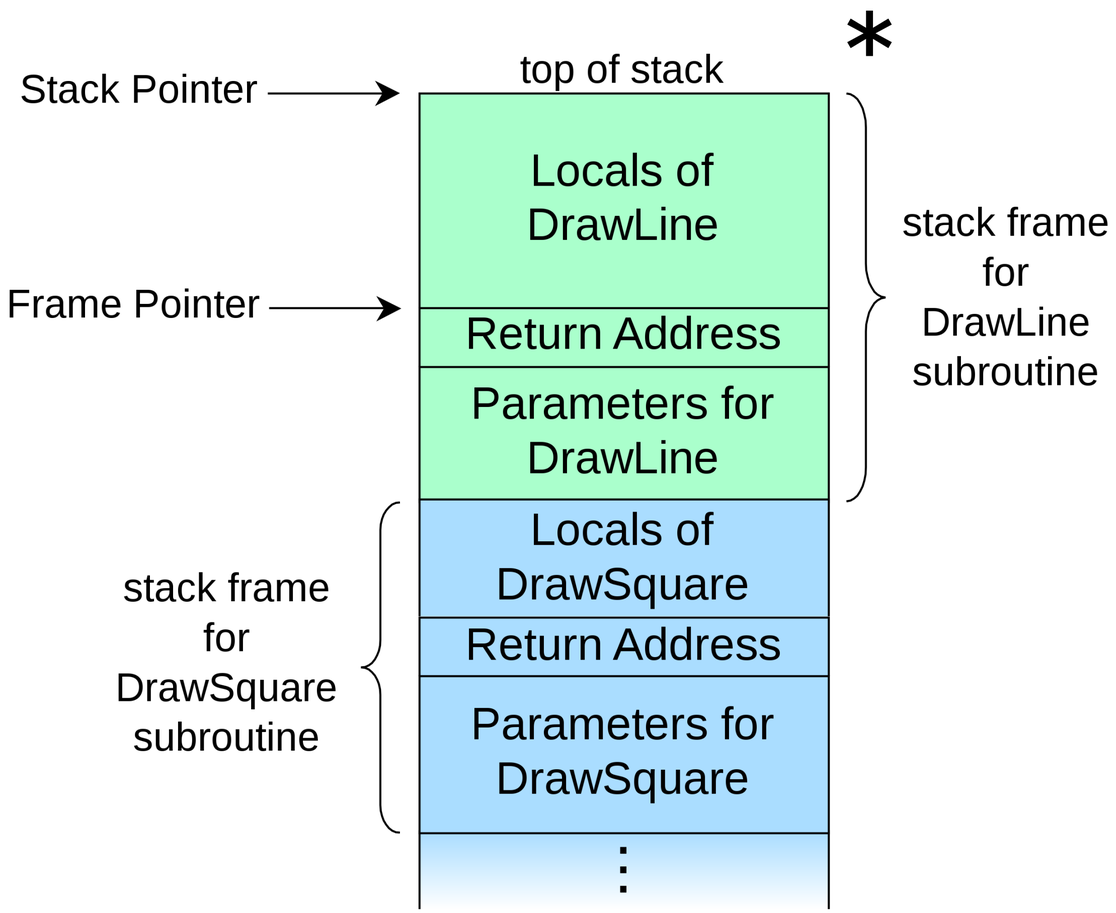
***since this is not a compilers course we will use this as a conceptual framework instead of an actual implementation guideline**

https://commons.wikimedia.org/wiki/File:Call_stack_layout.svg

Activation Records

Also called a **stack frame**, an activation record contains the information necessary to execute a subroutine, e.g.,

» actual parameters (not in the case of Project 2)



***since this is not a compilers course we will use this as a conceptual framework instead of an actual implementation guideline**

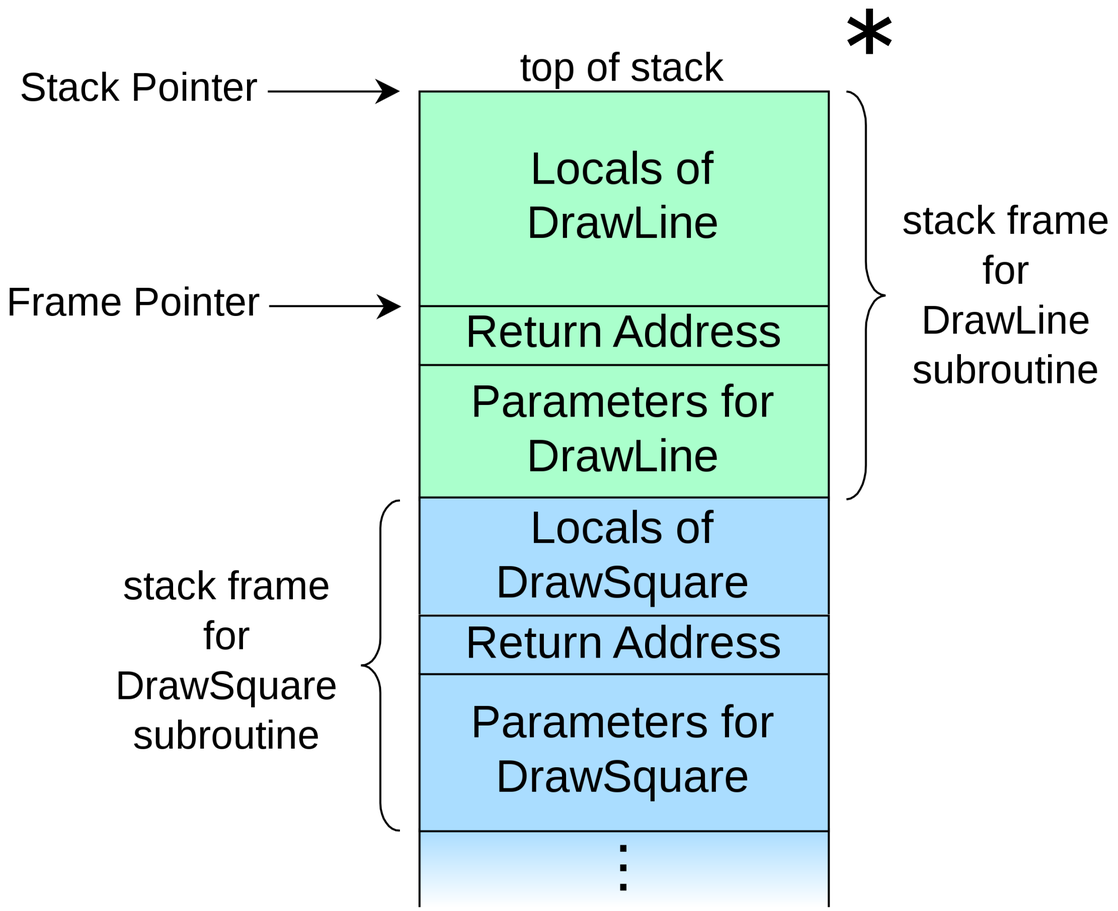
https://commons.wikimedia.org/wiki/File:Call_stack_layout.svg

Activation Records

Also called a **stack frame**, an activation record contains the information necessary to execute a subroutine, e.g.,

» actual parameters (not in the case of Project 2)

» local variables created during execution



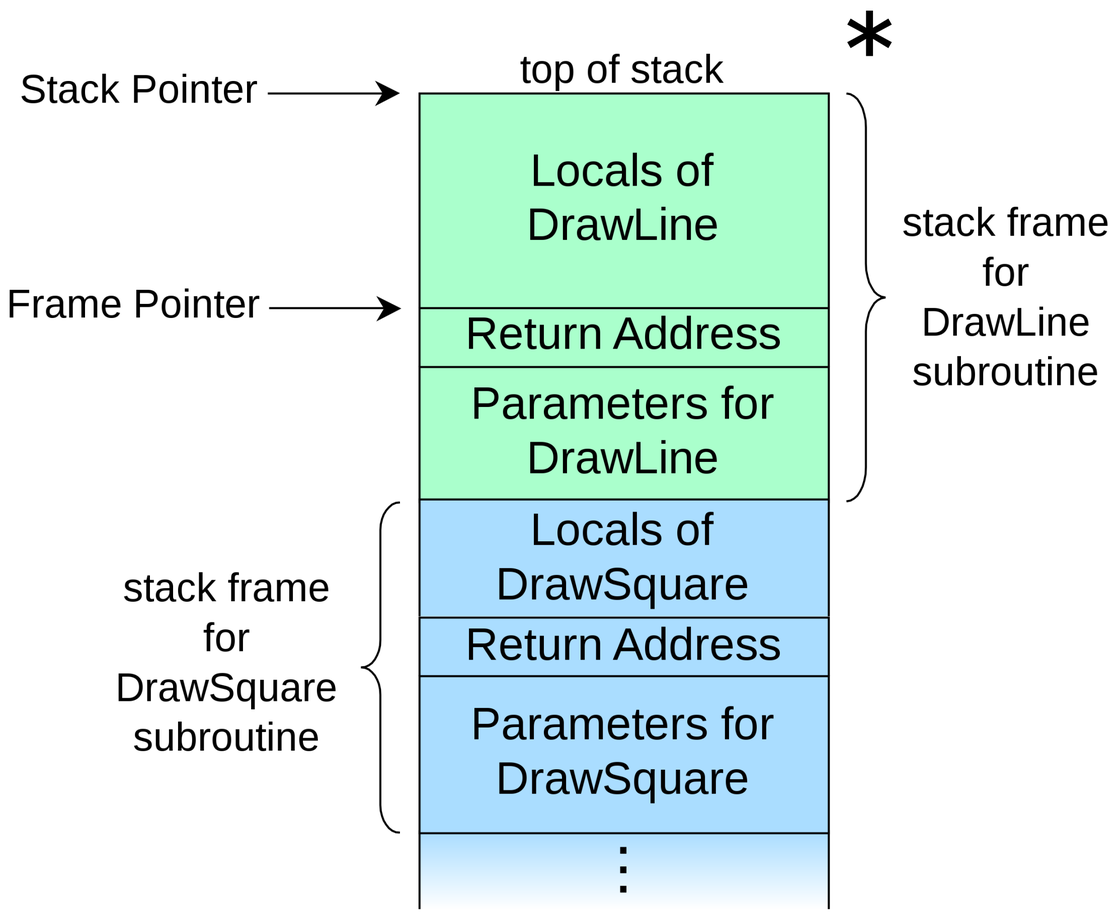
***since this is not a compilers course we will use this as a conceptual framework instead of an actual implementation guideline**

https://commons.wikimedia.org/wiki/File:Call_stack_layout.svg

Activation Records

Also called a **stack frame**, an activation record contains the information necessary to execute a subroutine, e.g.,

- actual parameters (not in the case of Project 2)
- local variables created during execution
- information about enclosing subroutines (e.g., activation records of other subroutines when nesting is allowed)



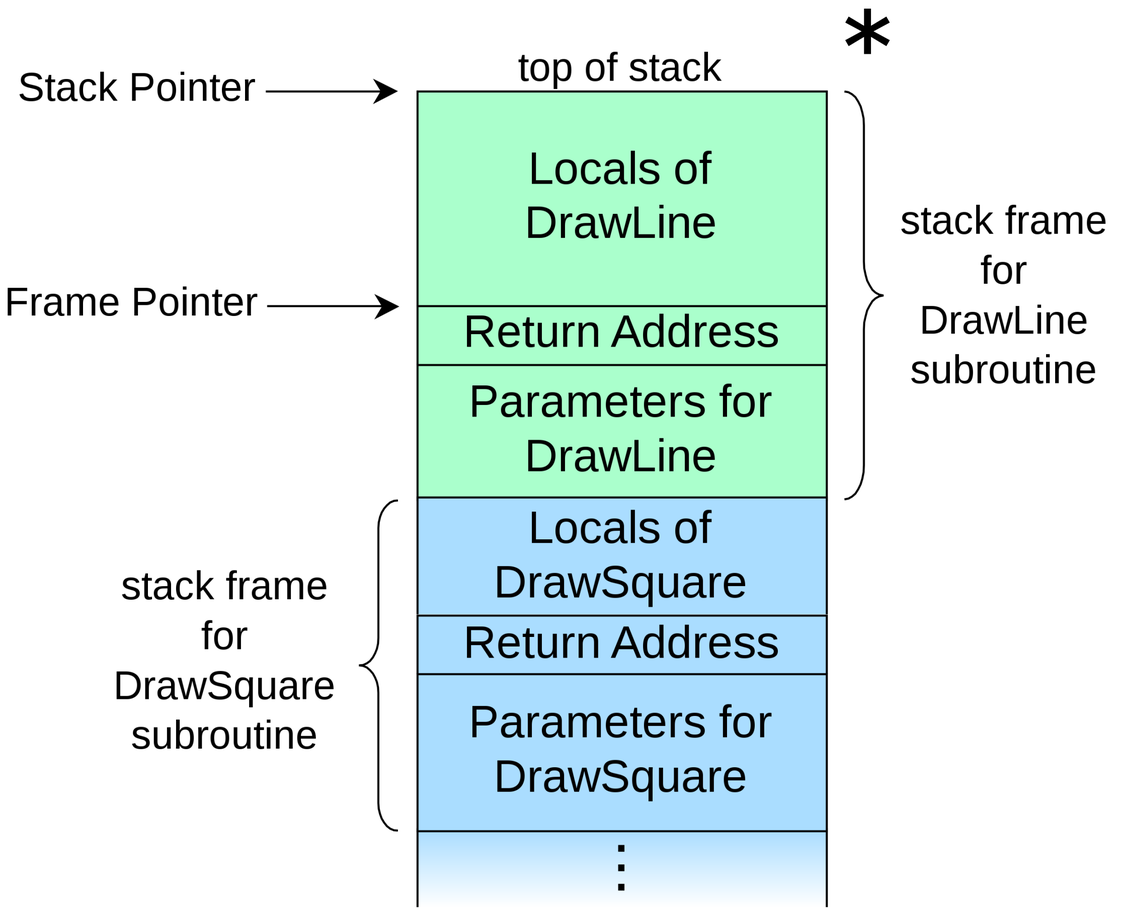
***since this is not a compilers course we will use this as a conceptual framework instead of an actual implementation guideline**

https://commons.wikimedia.org/wiki/File:Call_stack_layout.svg

Activation Records

Also called a **stack frame**, an activation record contains the information necessary to execute a subroutine, e.g.,

- actual parameters (not in the case of Project 2)
- local variables created during execution
- information about enclosing subroutines (e.g., activation records of other subroutines when nesting is allowed)
- the caller location for returning



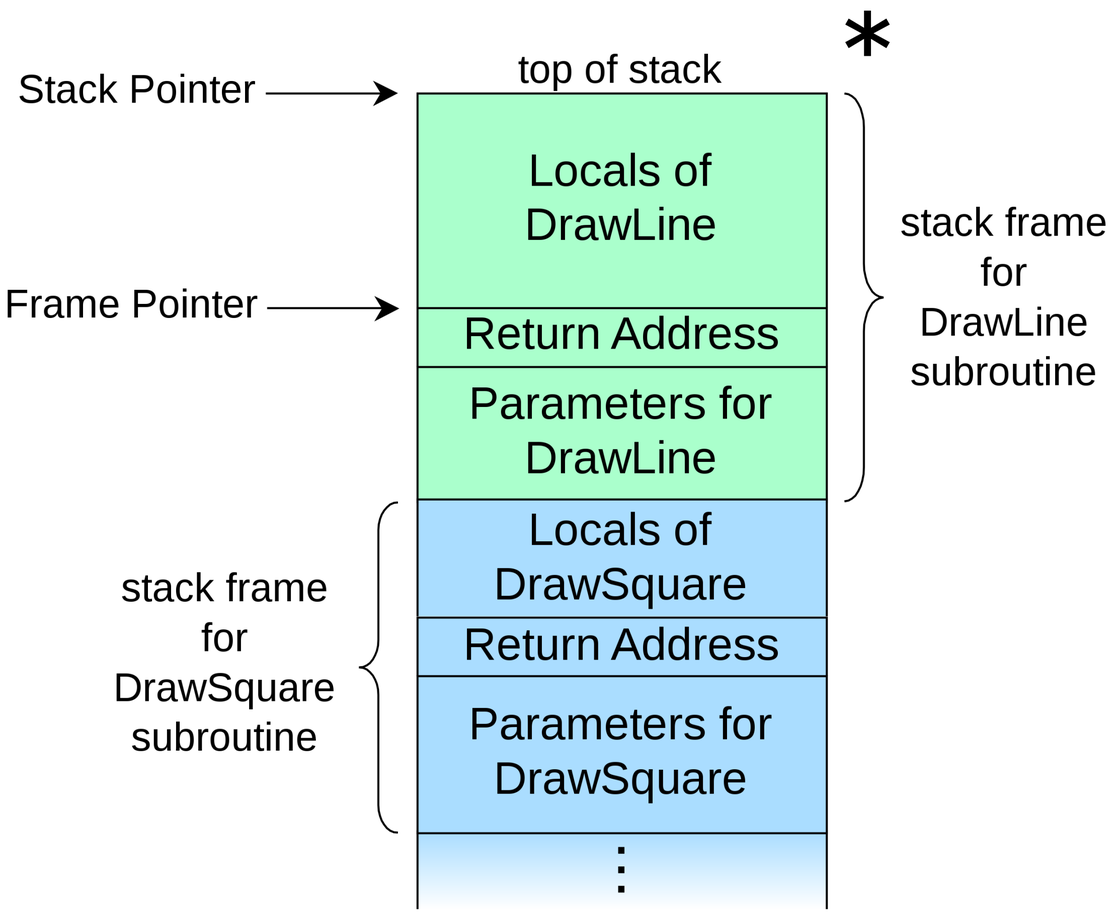
***since this is not a compilers course we will use this as a conceptual framework instead of an actual implementation guideline**

https://commons.wikimedia.org/wiki/File:Call_stack_layout.svg

Activation Records

Also called a **stack frame**, an activation record contains the information necessary to execute a subroutine, e.g.,

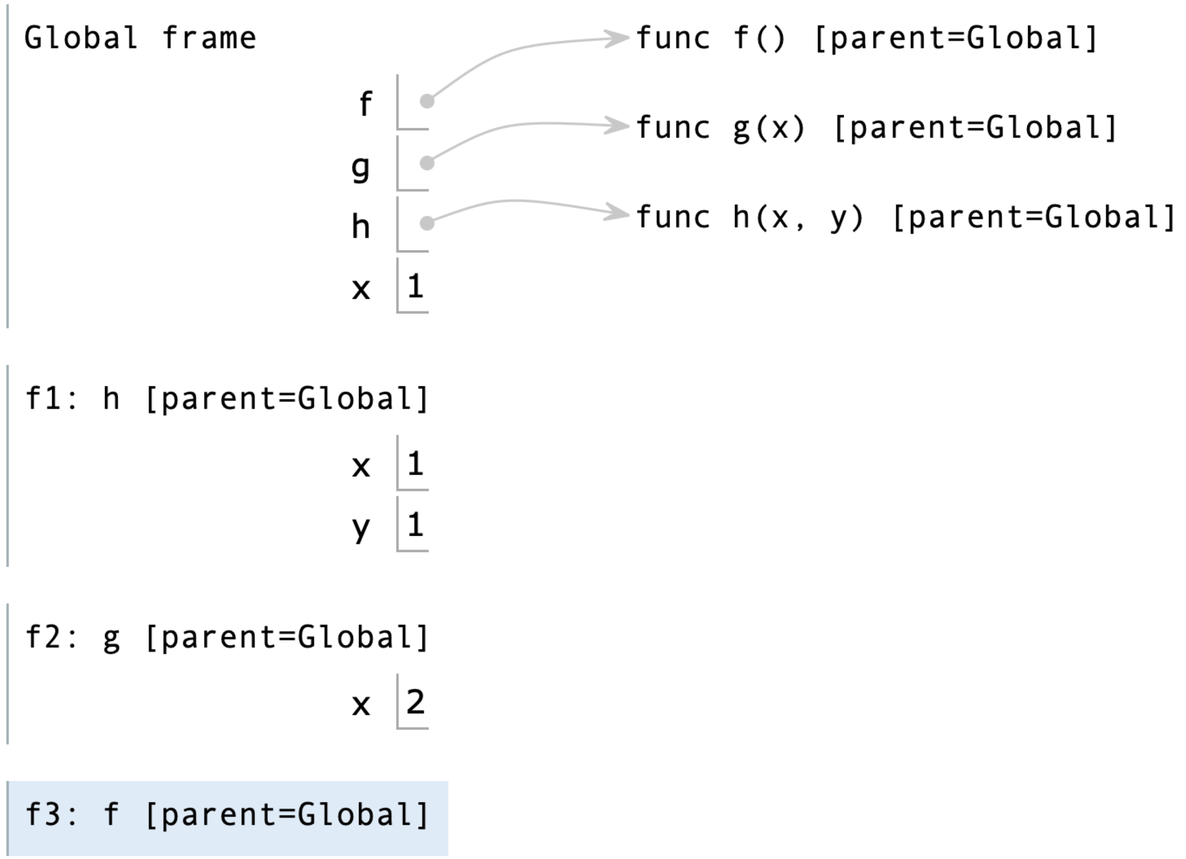
- actual parameters (not in the case of Project 2)
- local variables created during execution
- information about enclosing subroutines (e.g., activation records of other subroutines when nesting is allowed)
- the caller location for returning
- possibly a return value



***since this is not a compilers course we will use this as a conceptual framework instead of an actual implementation guideline**

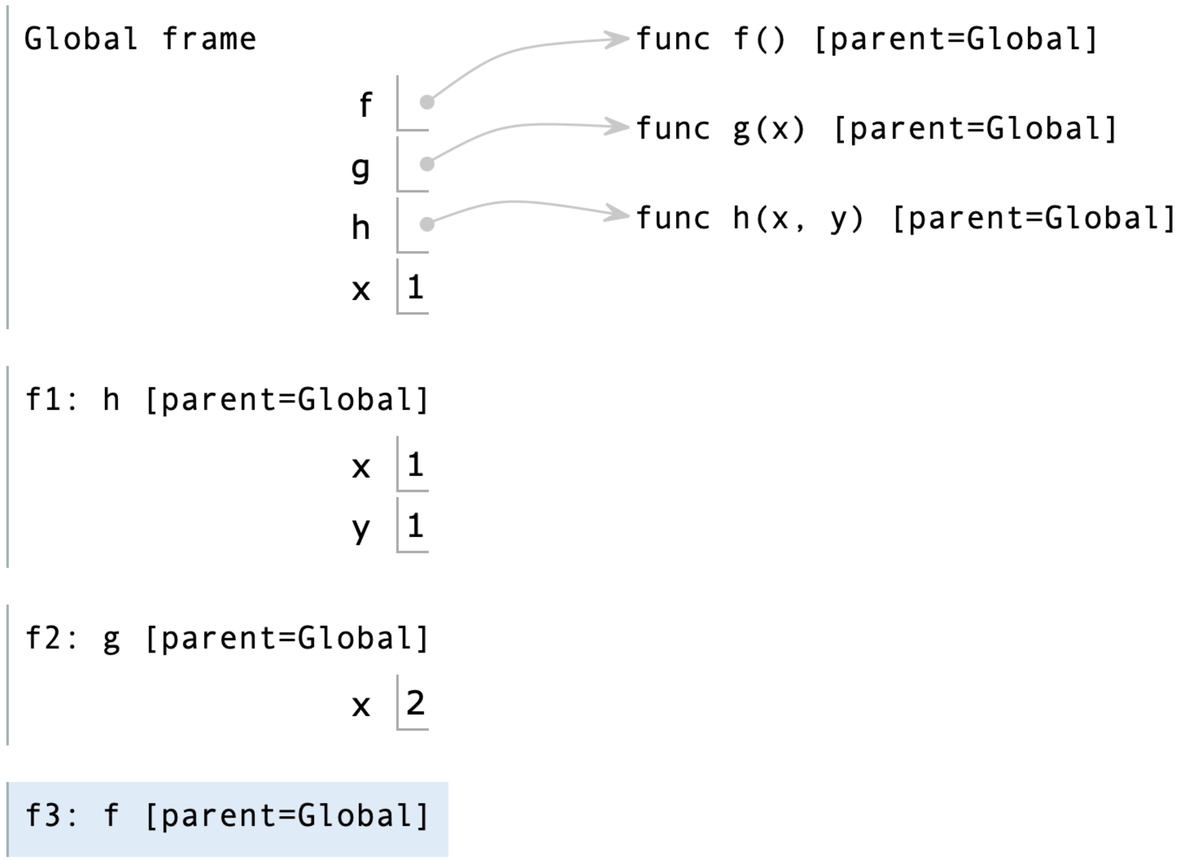
https://commons.wikimedia.org/wiki/File:Call_stack_layout.svg

The Call Stack



The Call Stack

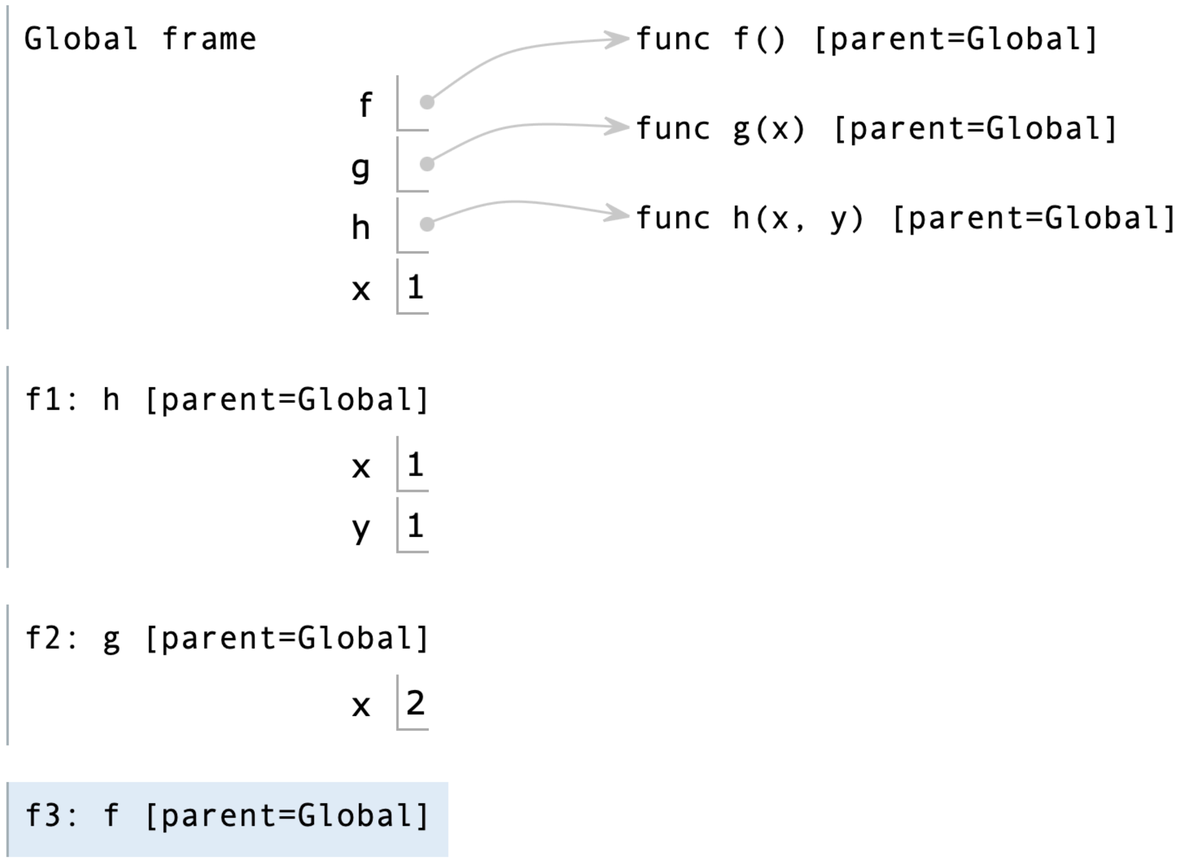
The **call stack** is a data structure for maintaining subroutine calls



The Call Stack

The **call stack** is a data structure for maintaining subroutine calls

It is a stack of activation records which are pushed when a function is called, and popped when a function returns

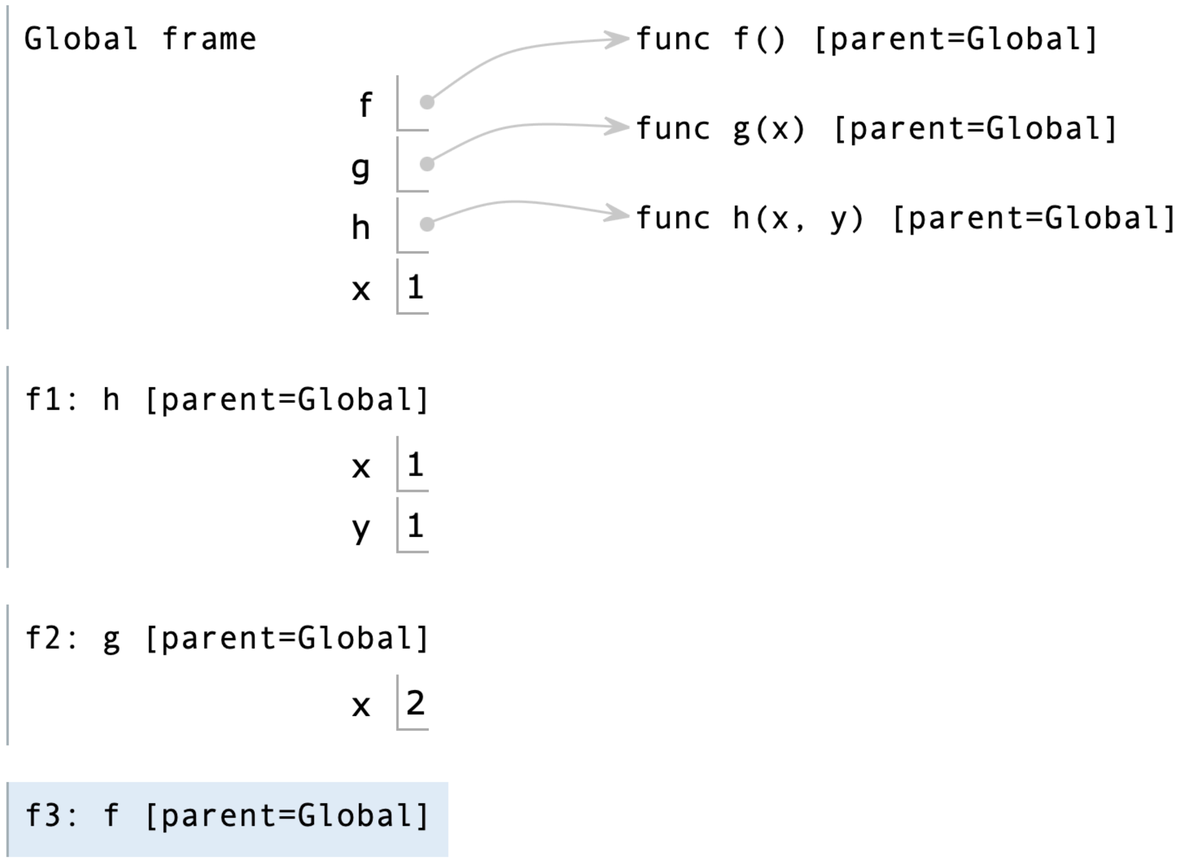


The Call Stack

The **call stack** is a data structure for maintaining subroutine calls

It is a stack of activation records which are pushed when a function is called, and popped when a function returns

At the bottom of the stack is the global frame, which contains bindings available anywhere in the program



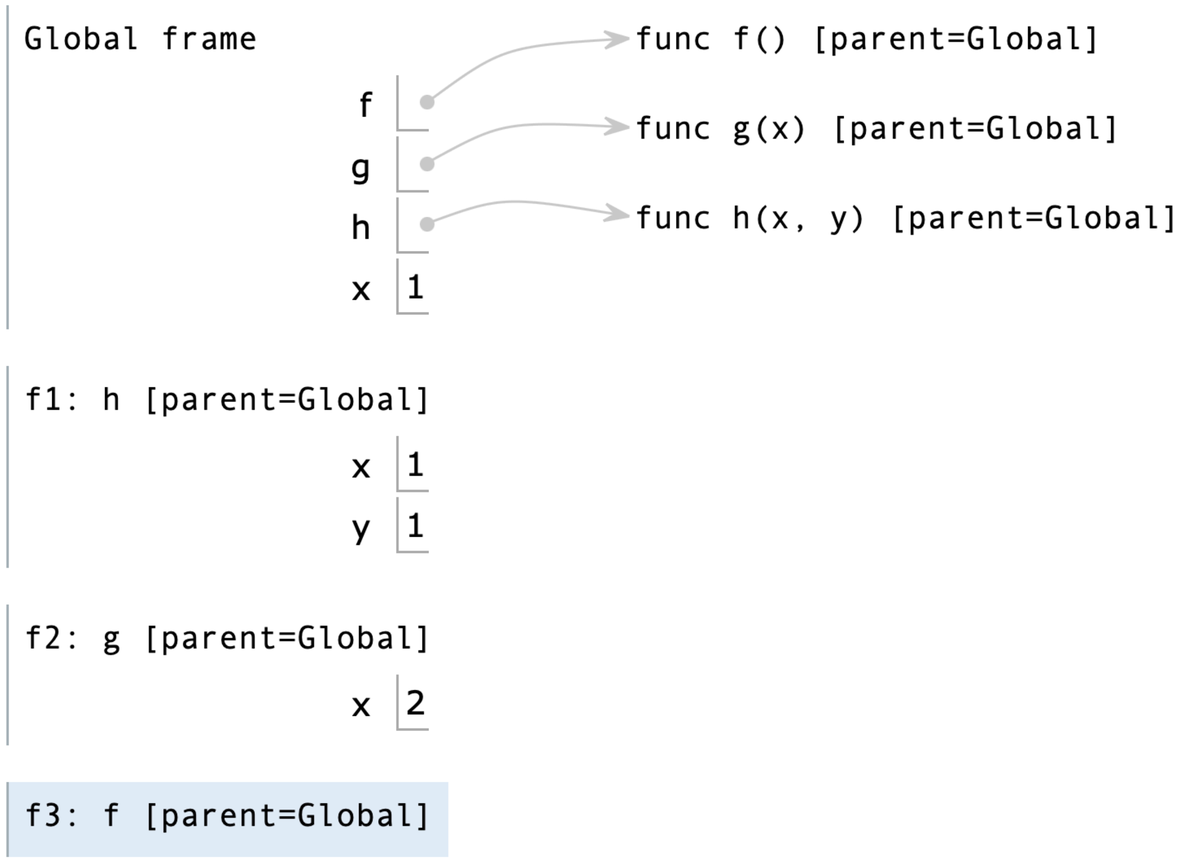
The Call Stack

The **call stack** is a data structure for maintaining subroutine calls

It is a stack of activation records which are pushed when a function is called, and popped when a function returns

At the bottom of the stack is the global frame, which contains bindings available anywhere in the program

Question. *Why a stack?*



demo

(python tutor, example-1.py)

An Aside: Stack Overflow

Recursive functions have the potential to add *many* activation records to the call stack.

Stack overflow occurs when the activation records created by a sequence of function calls do not fit on the call stack.

Python

```python
def fact(n):
  if n == 0:
    return 1
  return fact(n - 1) * n
```

Our Language

```
(fact): ▷ n
  (if) 0 n = ?
    1 Return ;
  (else)
    -1 n + fact # n *
    Return ;
; ▷ fact
```

demo

(python tutor, example-2.py)

Understanding Check

```python
def f(x):
    if x == 0:
        return 1
    return f(x // 4) + f(x // 2)

f(16)
```

Python

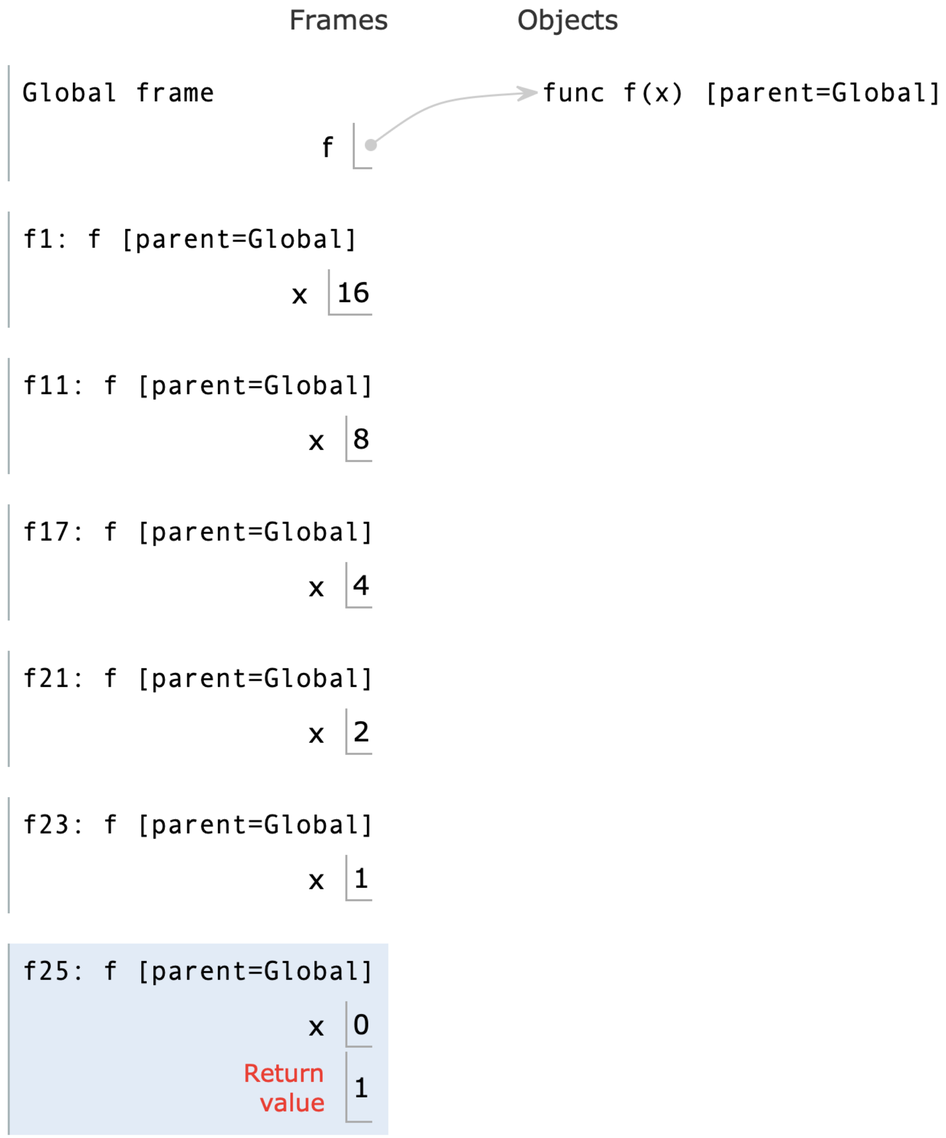
```
(f): ▷ x
  (if) 0 x = ?
    1 Return ;
  (else)
    4 x / f # ▷ a
    2 x / f # ▷ b
    a b + Return
  ;
; ▷ f

16 f #
```

Our Language

What is the maximum number of activation records on the call stack during the execution of this program (including the global frame)?

Answer

7


Frames
Objects
Global frame
f
func f(x) [parent=Global]
f1: f [parent=Global]
x
16
f11: f [parent=Global]
x
8
f17: f [parent=Global]
x
4
f21: f [parent=Global]
x
2
f23: f [parent=Global]
x
1
f25: f [parent=Global]
x
0
Return value
1

The Environment

Motivation

Motivation

What variables are in scope? How do we access them? Are variables mutable?

Motivation

What variables are in scope? How do we access them? Are variables mutable?

Intuition: Variables should be in scope if they were within the context where the function was defined.

Motivation

What variables are in scope? How do we access them? Are variables mutable?

Intuition: Variables should be in scope if they were within the context where the function was defined.

We will consider two cases in this course:

Motivation

What variables are in scope? How do we access them? Are variables mutable?

Intuition: Variables should be in scope if they were within the context where the function was defined.

We will consider two cases in this course:

» Locally Mutable (Project 2)

Motivation

What variables are in scope? How do we access them? Are variables mutable?

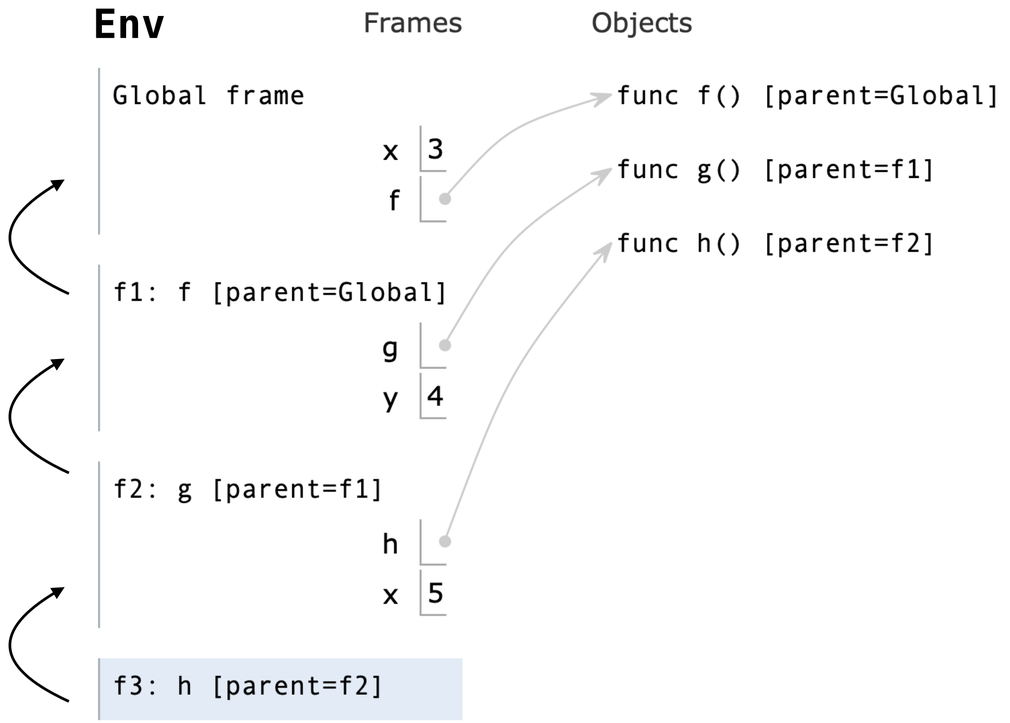
Intuition: Variables should be in scope if they were within the context where the function was defined.

We will consider two cases in this course:

» Locally Mutable (Project 2)

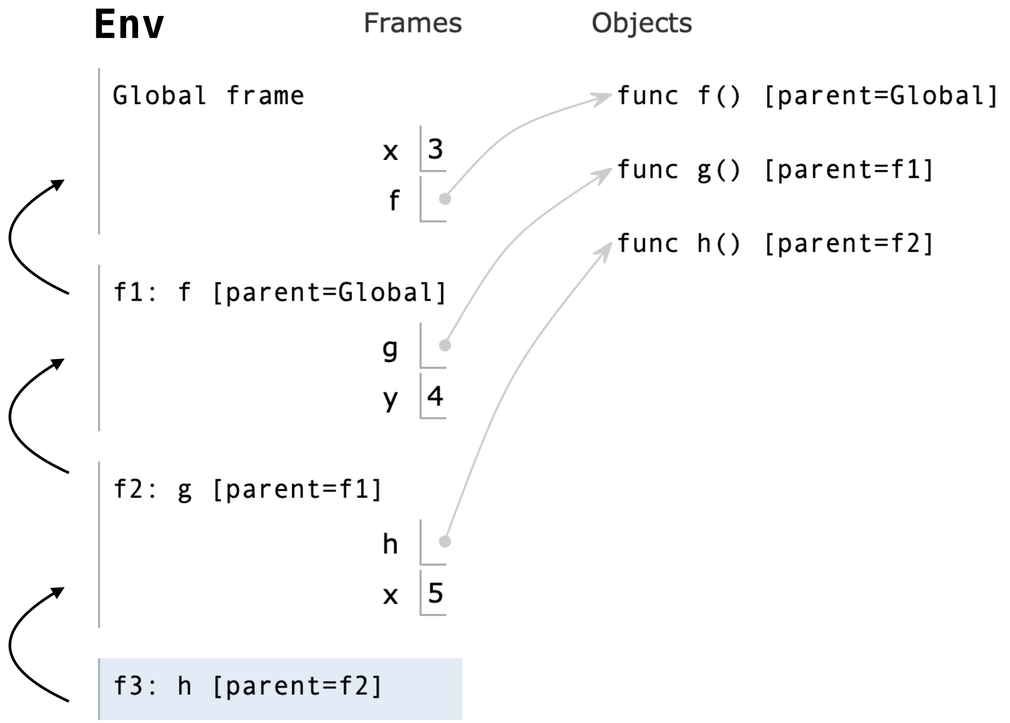
» Always immutable (e.g., let-bindings) (Project 3)

Fetching from the Environment



Fetching from the Environment

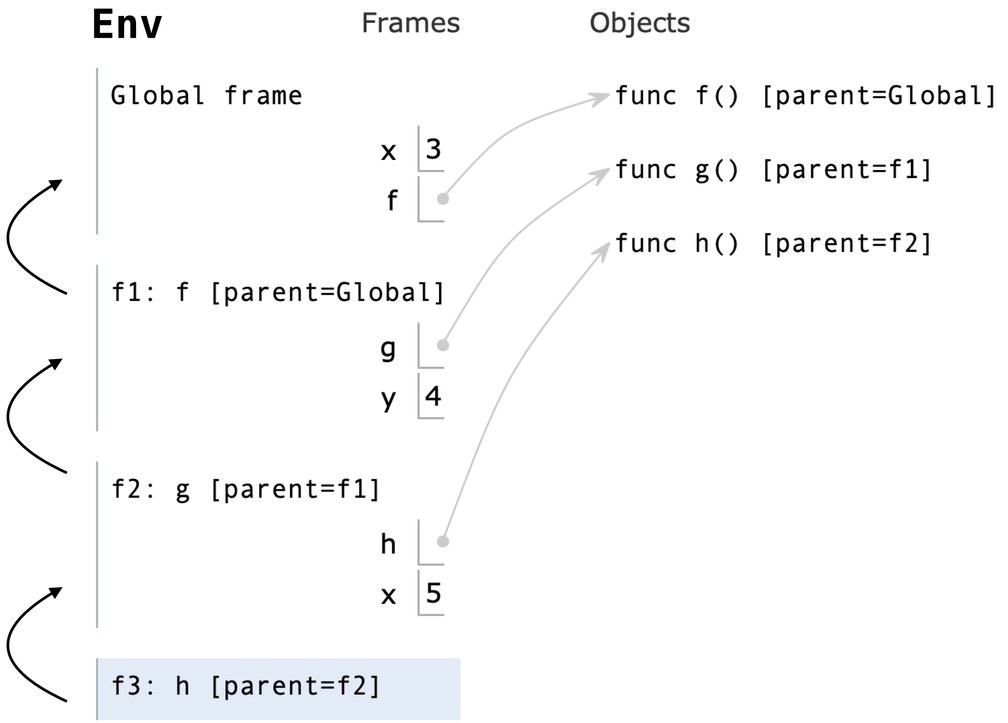
`fetch(Env, x)` algorithm:



Fetching from the Environment

fetch(Env, x) algorithm:

» Check locally (within the activation record of the called function)

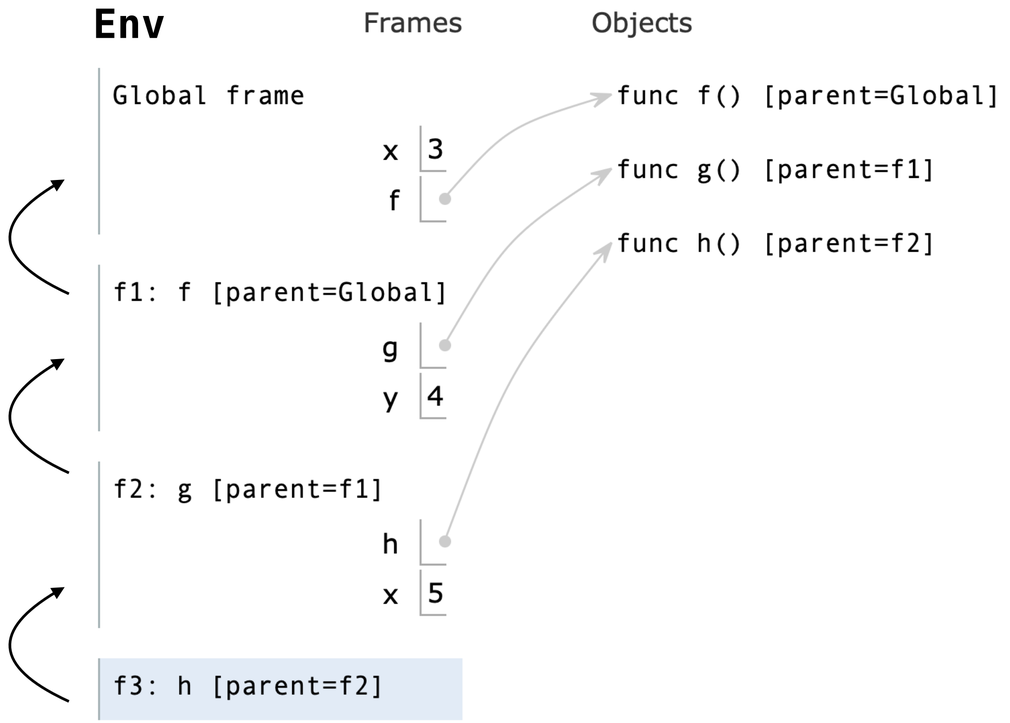


Fetching from the Environment

fetch(Env, x) algorithm:

» Check locally (within the activation record of the called function)

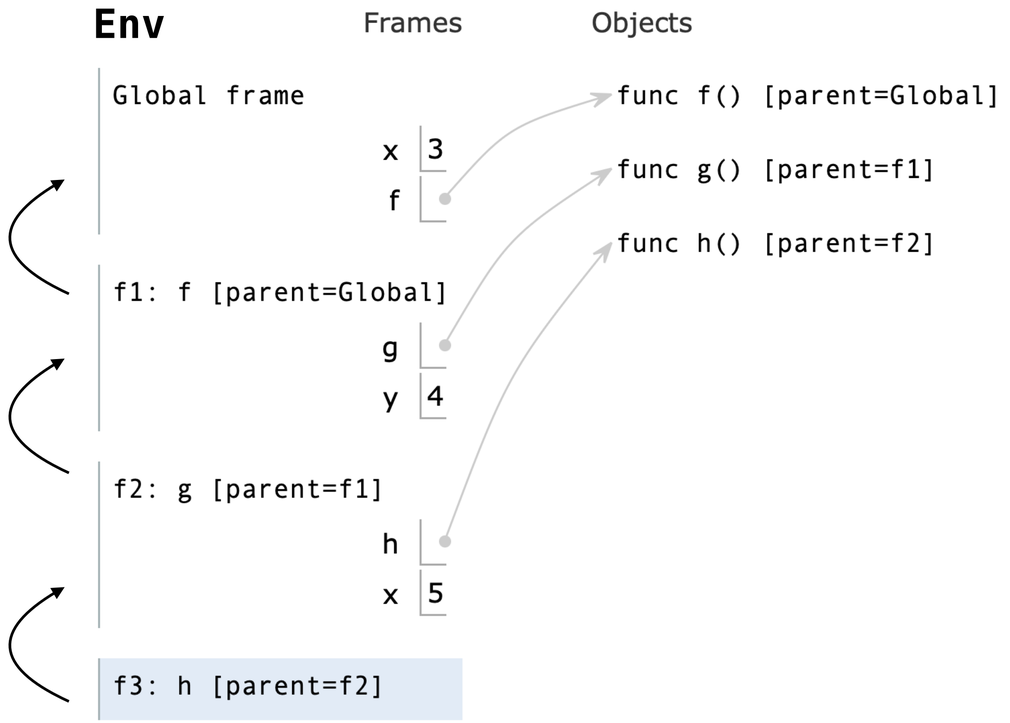
» If failed, check record of enclosing subroutine (the activation record for the function where the called function was defined)



Fetching from the Environment

fetch(Env, x) algorithm:

- Check locally (within the activation record of the called function)
- If failed, check record of enclosing subroutine (the activation record for the function where the called function was defined)
- Repeat until you get to the global frame. Fail if you did not find a binding

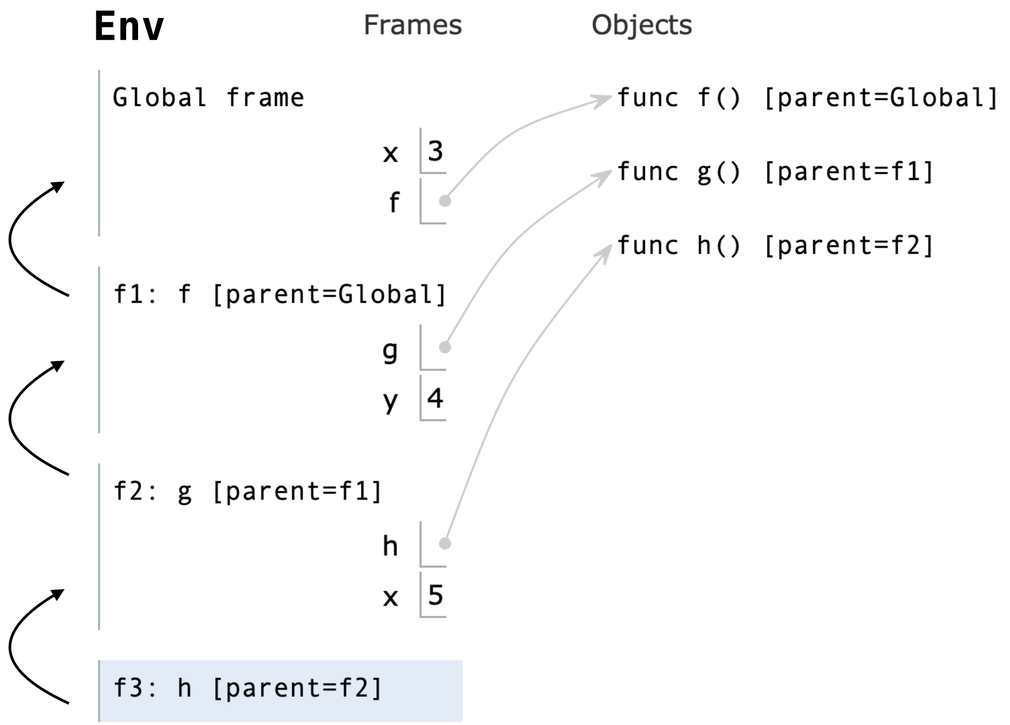


Fetching from the Environment

fetch(Env, x) algorithm:

- Check locally (within the activation record of the called function)
- If failed, check record of enclosing subroutine (the activation record for the function where the called function was defined)
- Repeat until you get to the global frame. Fail if you did not find a binding

Each record has a way to get to the record in which the callee was defined.

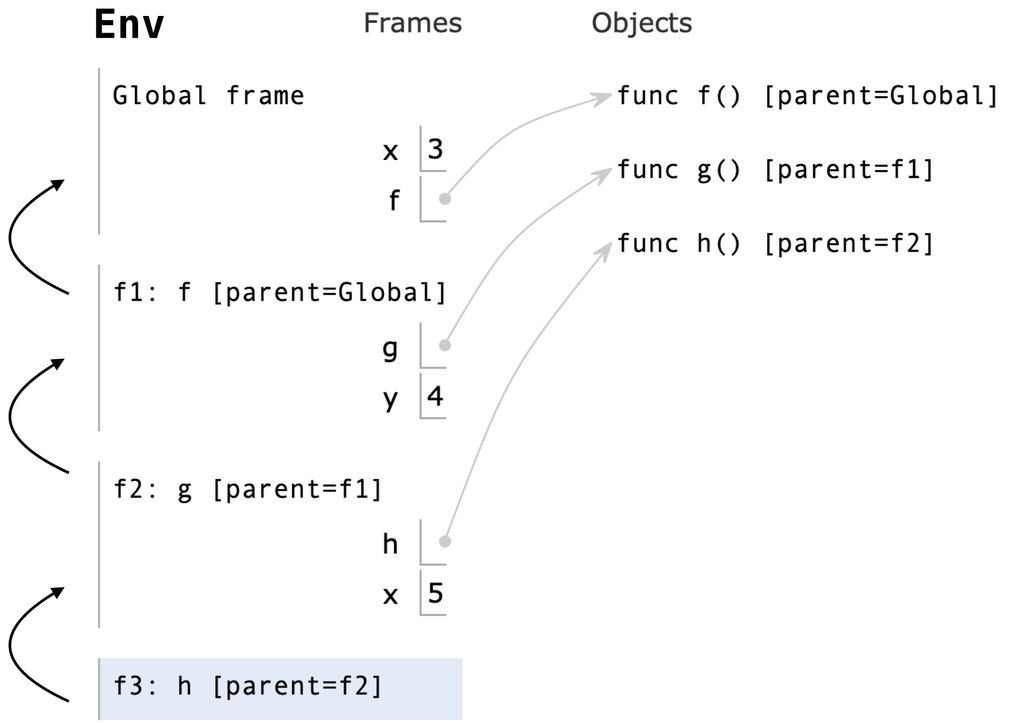


Fetching from the Environment

fetch(Env, x) algorithm:

- Check locally (within the activation record of the called function)
- If failed, check record of enclosing subroutine (the activation record for the function where the called function was defined)
- Repeat until you get to the global frame. Fail if you did not find a binding

Each record has a way to get to the record in which the callee was defined.



what is the value of x?

Fetching from the Environment

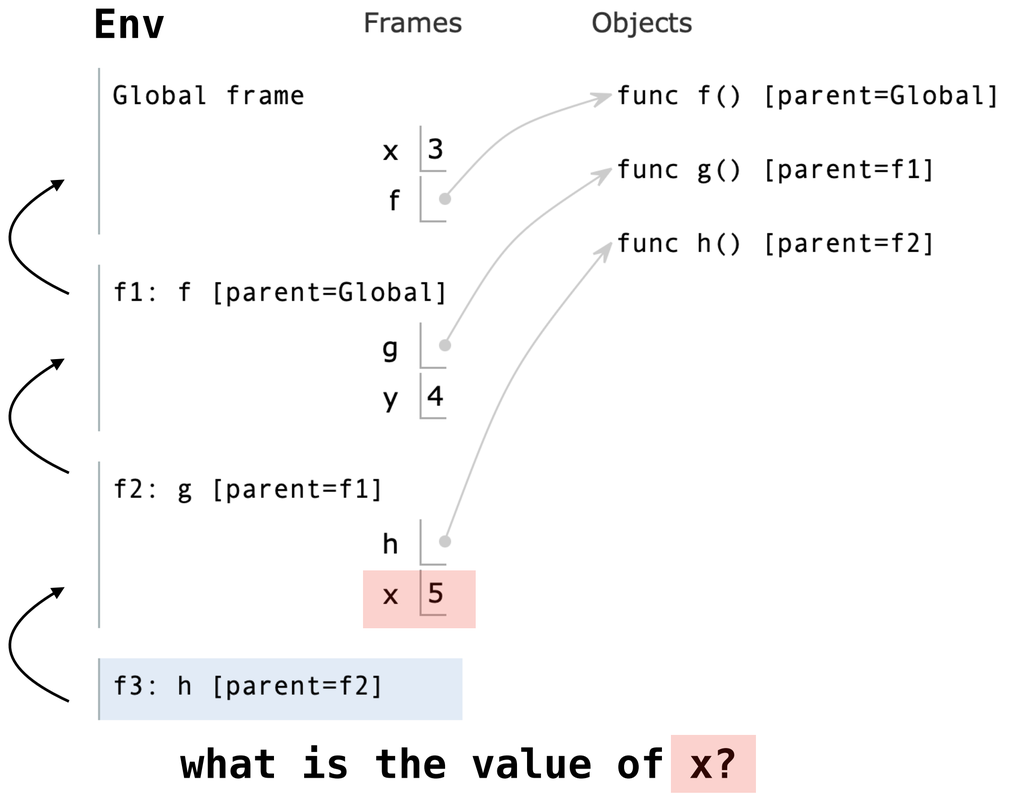
fetch(Env, x) algorithm:

» Check locally (within the activation record of the called function)

» If failed, check record of enclosing subroutine (the activation record for the function where the called function was defined)

» Repeat until you get to the global frame. Fail if you did not find a binding

Each record has a way to get to the record in which the callee was defined.



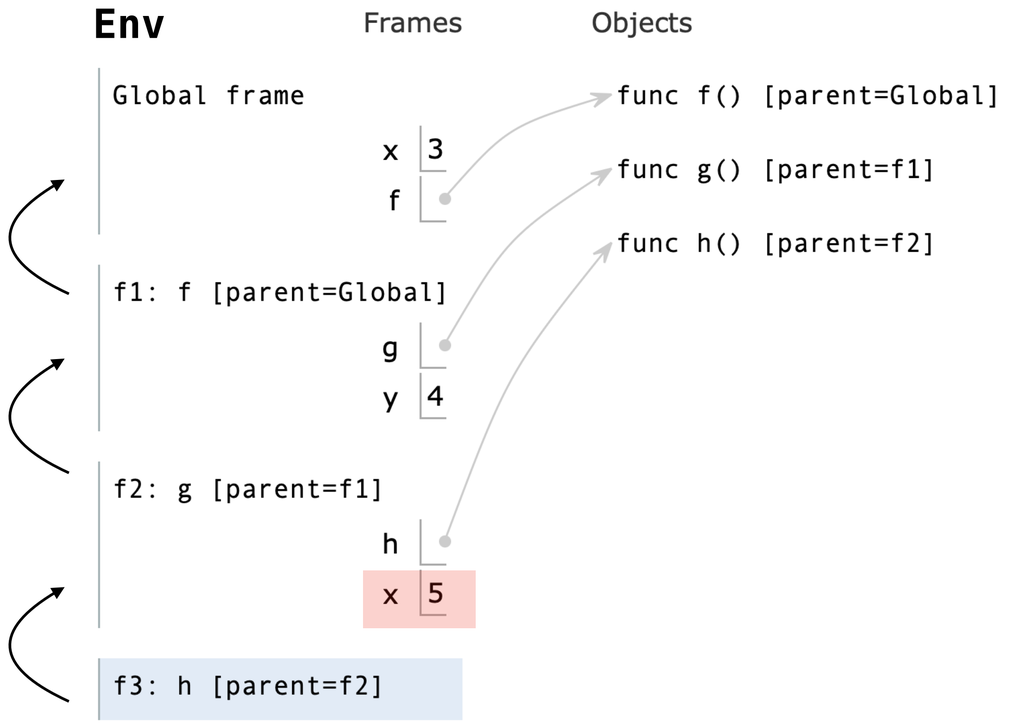
what is the value of x?

Fetching from the Environment

fetch(Env, x) algorithm:

- Check locally (within the activation record of the called function)
- If failed, check record of enclosing subroutine (the activation record for the function where the called function was defined)
- Repeat until you get to the global frame. Fail if you did not find a binding

Each record has a way to get to the record in which the callee was defined.



what is the value of x?

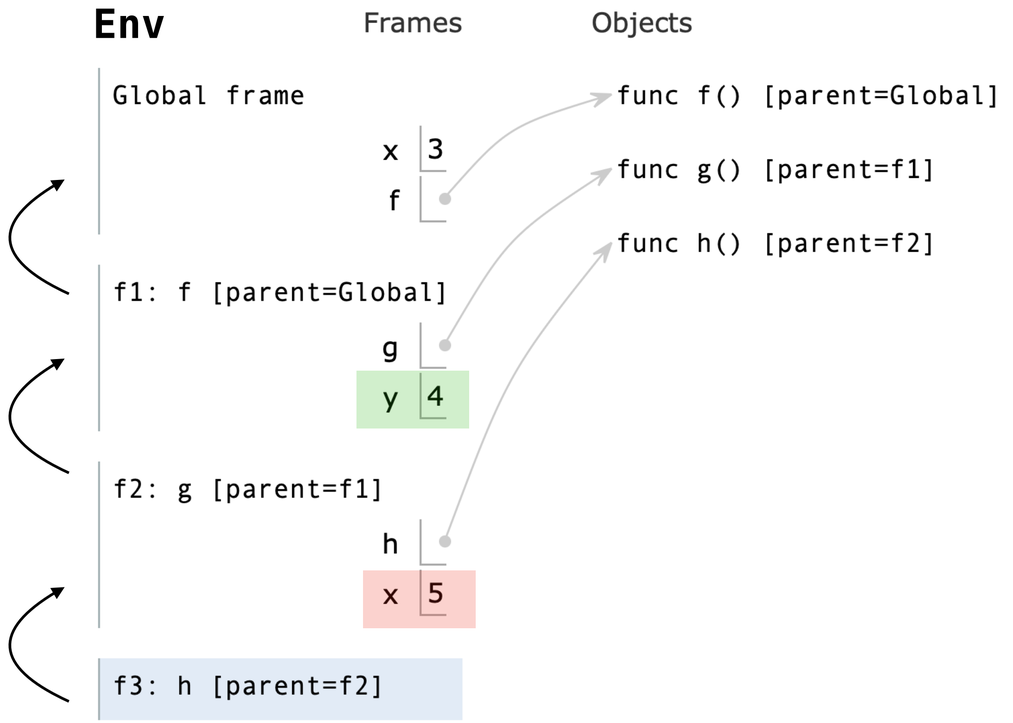
what is the value of y?

Fetching from the Environment

fetch(Env, x) algorithm:

- Check locally (within the activation record of the called function)
- If failed, check record of enclosing subroutine (the activation record for the function where the called function was defined)
- Repeat until you get to the global frame. Fail if you did not find a binding

Each record has a way to get to the record in which the callee was defined.



what is the value of x?

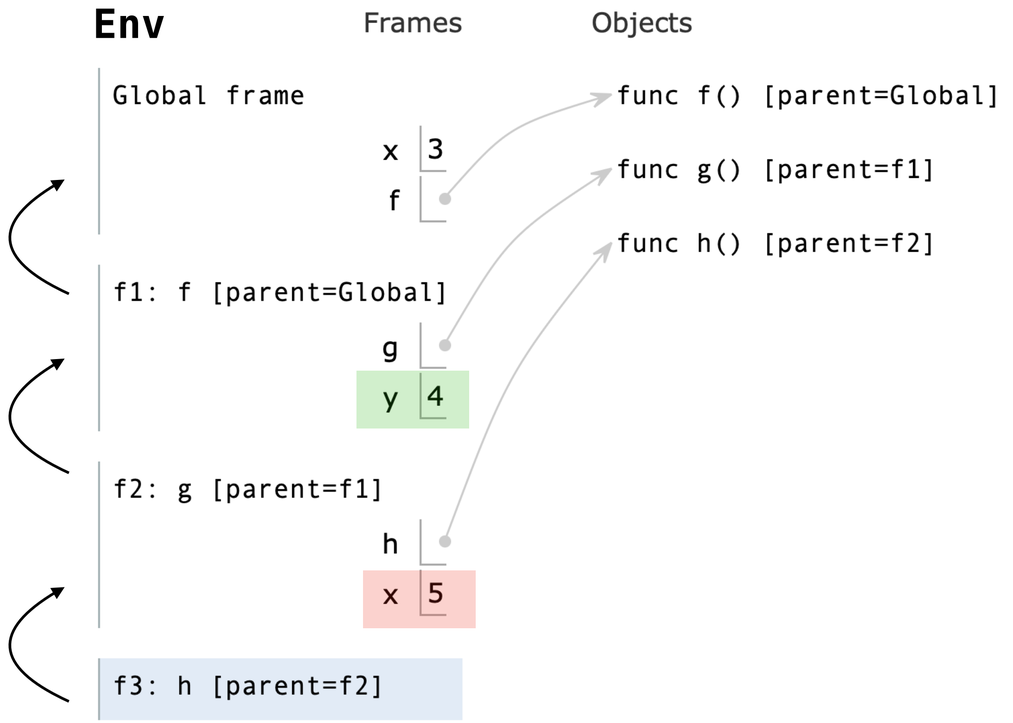
what is the value of y?

Fetching from the Environment

fetch(Env, x) algorithm:

- Check locally (within the activation record of the called function)
- If failed, check record of enclosing subroutine (the activation record for the function where the called function was defined)
- Repeat until you get to the global frame. Fail if you did not find a binding

Each record has a way to get to the record in which the callee was defined.



what is the value of x?

what is the value of y?

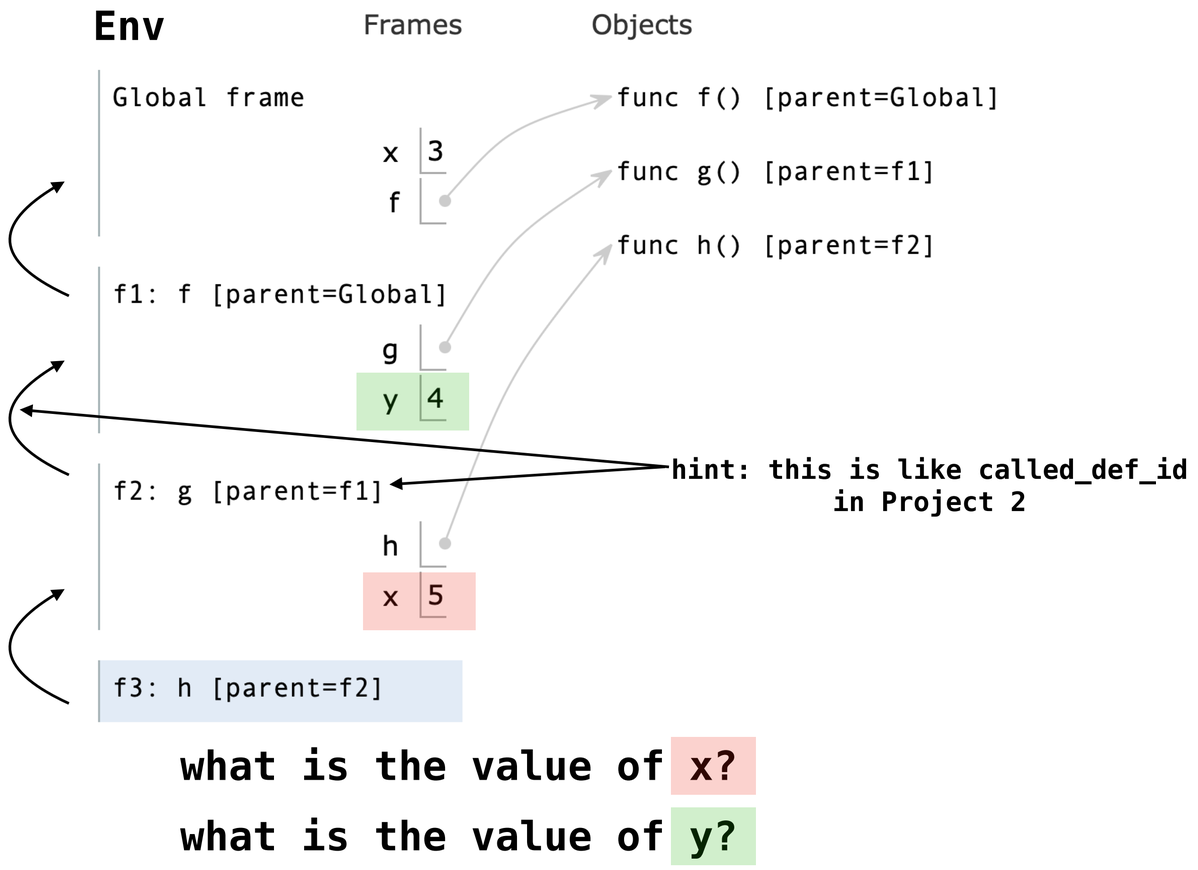
Take the first binding you see, going to higher nestings.

Fetching from the Environment

fetch(Env, x) algorithm:

- Check locally (within the activation record of the called function)
- If failed, check record of enclosing subroutine (the activation record for the function where the called function was defined)
- Repeat until you get to the global frame. Fail if you did not find a binding

Each record has a way to get to the record in which the callee was defined.



what is the value of x?

what is the value of y?

Take the first binding you see, going to higher nestings.

demo

(python tutor, example-5.py)

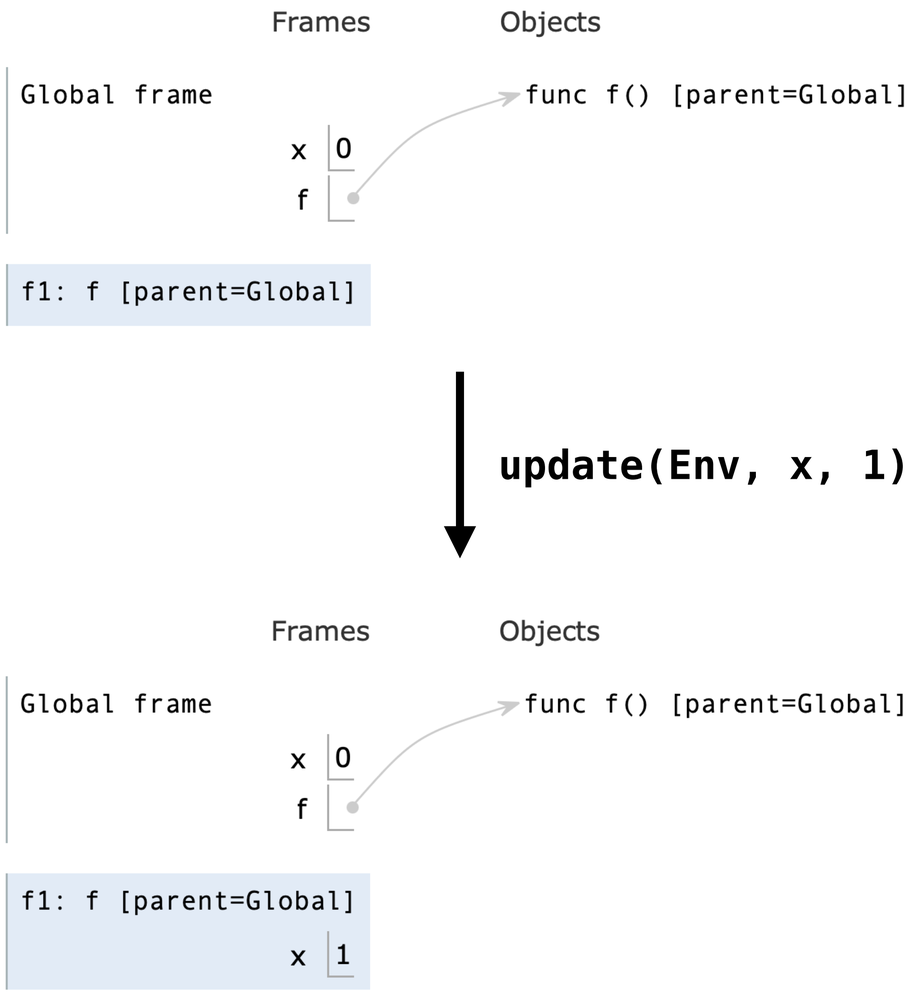
Updating the Environment

update(Env, x, v) algorithm:

» add the binding (x, v) to the topmost activation stack, rebinding if it already exists

updates always shadow existing assignments in lower records

Note. Its up to you if you want to use mutating or non-mutating updates, but think about what will happen with while loops...



demo

(python tutor, example-6.py)

A Note on Nested Subroutines

```python
def f():
    def g():
        return x

    x = 1
    y = g()
    x = 2
    z = g()
    return y == z


out = f()
```

Python

```
(f):
  (g):
    x Return
  ; ▷ g
  1 ▷ x
  g # ▷ y
  2 ▷ x
  g # ▷ z
  y z = Return
; ▷ f

f # ▷ out
```

Our Language

One important consequence of variables being mutable is that mutating variables should change the behavior of functions which use them.

Important. This is not dynamic scoping, x is in (lexical) scope for g.

demo

(python tutor, example-4.py)

Understanding Check

```python
def f():
    x = 0
    print(x)

def g():
    print(x)

x = 1
f()
g()
```

Python

```
(f):
  0 ▷ x
  x .
; ▷ f

(g): x . ; ▷ g

1 ▷ x
f #
g #
```

Our Language

What does this print under dynamic scoping?
under lexical scoping?

Answer

Dynamic:

0
1

Lexical:

1
1

```
(f):
  0 ▷ x
  x .
; ▷ f

(g): x . ; ▷ g

1 ▷ x
f #
g #
```

Closures

Recall: Higher-Order Programming

Recall: Higher-Order Programming

Functions are **first-class values:**

Recall: Higher-Order Programming

Functions are **first-class values:**

1. They can be bound to names

Recall: Higher-Order Programming

Functions are **first-class values**:

1. They can be bound to names
2. They can be returned by another function.

Recall: Higher-Order Programming

Functions are **first-class values:**

1. They can be bound to names
2. They can be returned by another function.
3. They can be passed as arguments to another function

Recall: Higher-Order Programming

Functions are **first-class values:**

1. They can be bound to names
2. They can be returned by another function.
3. They can be passed as arguments to another function

(In OCaml and in our Toy Language)

Recall: Higher-Order Programming

Functions are **first-class values:**

1. They can be bound to names
2. They can be returned by another function.
3. They can be passed as arguments to another function

(In OCaml and in our Toy Language)

There's some trickiness to this...

demo

(python tutor, example-3.py)

Closures

```
(Env, P, ...)
```

Closures

(Env, P, ...)

A **closure** is a subroutine together with an environment and other data which may be useful for executing the function (name, pointer to activation record where the function is defined

Closures

(Env, P, ...)

A **closure** is a subroutine together with an environment and other data which may be useful for executing the function (name, pointer to activation record where the function is defined

Env contains captured bindings, the bindings which were defined in records that don't exist when the function is called

Closures

(Env, P, ...)

A **closure** is a subroutine together with an environment and other data which may be useful for executing the function (name, pointer to activation record where the function is defined

Env contains captured bindings, the bindings which were defined in records that don't exist when the function is called

(This is only a problem is the function is returned because that is when the defining record is discarded)

demo

(python tutor, example-3.py)

An Aside: Nonlocal Updates

In our language, updates always binds variables locally, shadowing preexisting bindings (same with "vanilla" python)

In some languages, we can update the variables nonlocally, this makes for more complicated semantics.

Closures are not enough for this.

demo

(python tutor, example-7.py)

reminder: we are not implementing this

Understanding Check

```
def f():
    x = 0
    def g():
        x = x + 1
        print(x)
    return g

h = f()
h()
h()
```

Python

```
(f):
  0 ▷ x
  (g):
    1 x + ▷ x
    x .
  ; ▷ g
; ▷ f
f # ▷ h
h #
h #
```

Our Language

What do these programs print (according to the semantics we've described so far)?

Answer

They both print:

1
1

(you should try to understand why this is the case from the level of semantics of Project 2)

Example from Project 2 (If there's time)

```
0 ▷ x
(f):
  1 ▷ x
  2 ▷ z
; ▷ f
f #
```