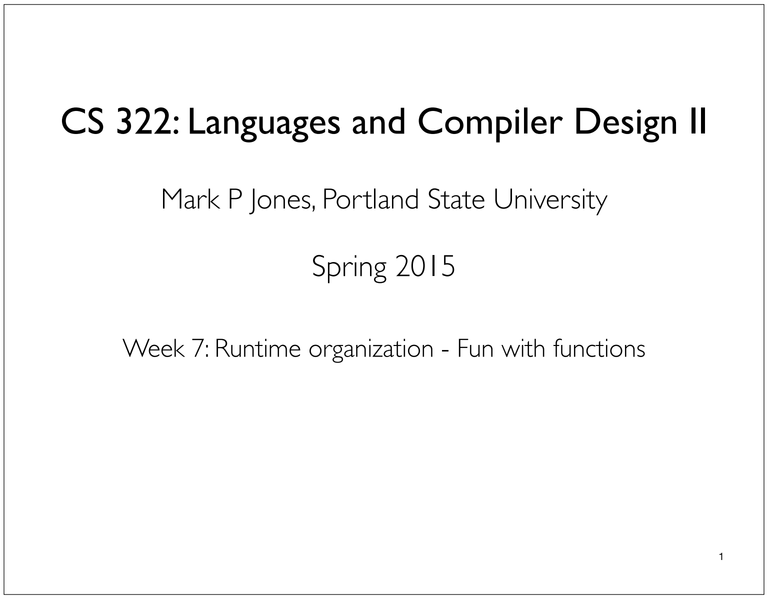
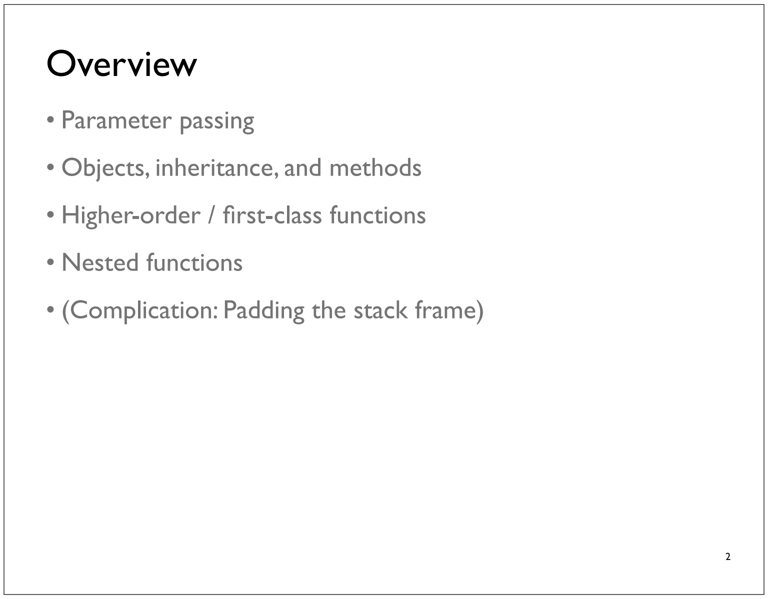


CS 322: Languages and Compiler Design II

Mark P Jones, Portland State University

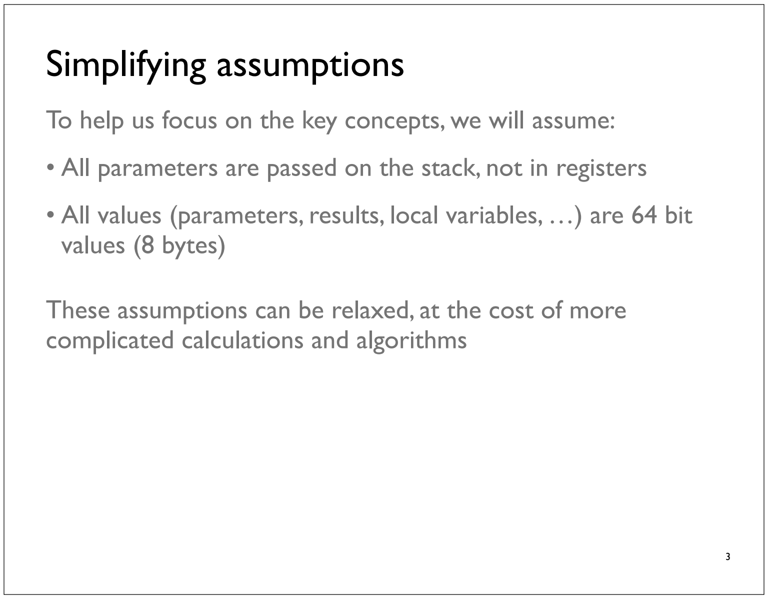
Spring 2015

Week 7: Runtime organization - Fun with functions



Overview

- Parameter passing
- Objects, inheritance, and methods
- Higher-order / first-class functions
- Nested functions
- (Complication: Padding the stack frame)

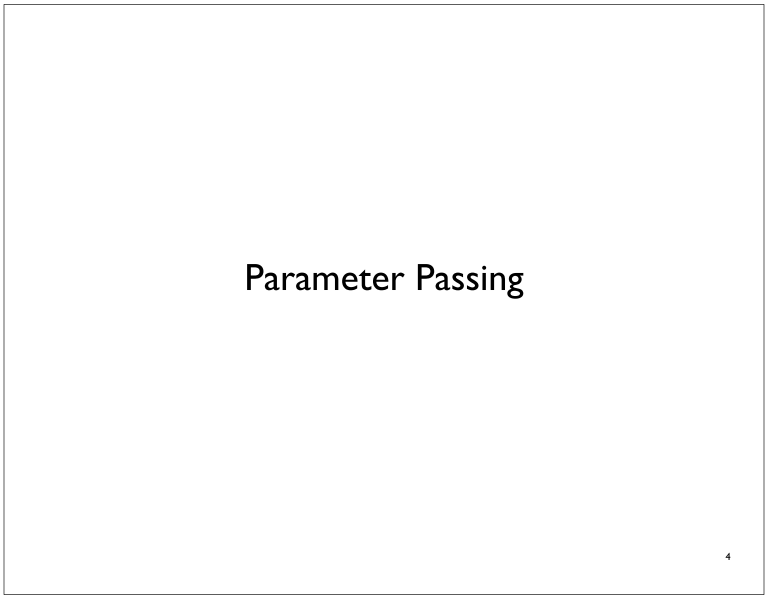


Simplifying assumptions

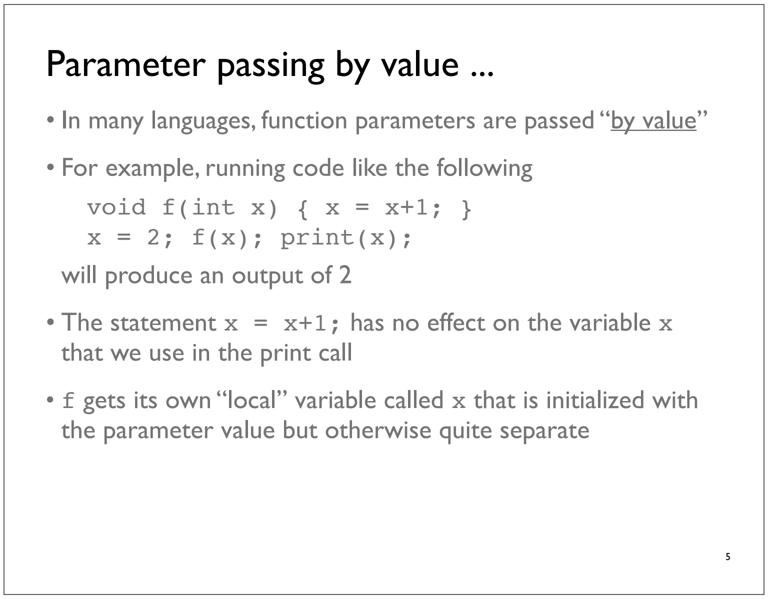
To help us focus on the key concepts, we will assume:

- All parameters are passed on the stack, not in registers
- All values (parameters, results, local variables, …) are 64 bit values (8 bytes)

These assumptions can be relaxed, at the cost of more complicated calculations and algorithms



Parameter Passing



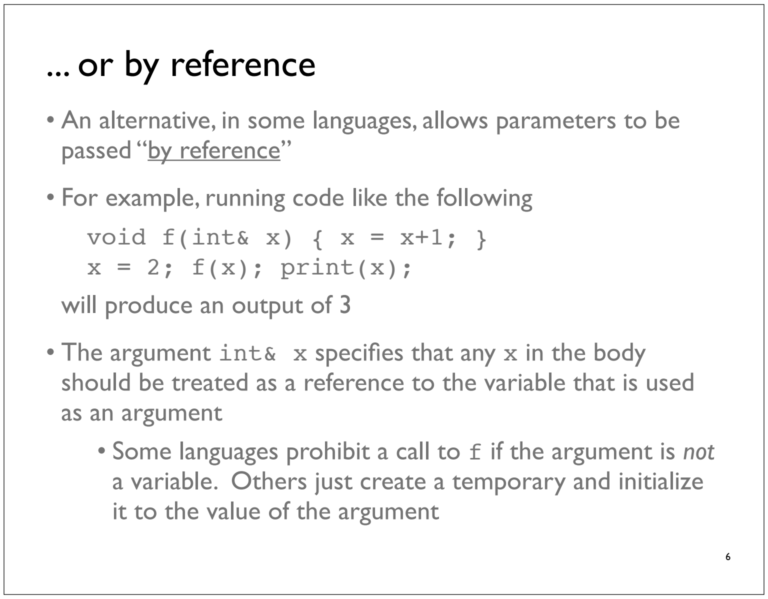
Parameter passing by value ...

- In many languages, function parameters are passed "by value"
- For example, running code like the following

```
void f(int x) { x = x+1; }
x = 2; f(x); print(x);
```

will produce an output of 2

- The statement `x = x+1;` has no effect on the variable `x` that we use in the print call
- `f` gets its own "local" variable called `x` that is initialized with the parameter value but otherwise quite separate



... or by reference

- An alternative, in some languages, allows parameters to be passed "by reference"
- For example, running code like the following

```
void f(int& x) { x = x+1; }
x = 2; f(x); print(x);
```

will produce an output of 3

- The argument `int& x` specifies that any `x` in the body should be treated as a reference to the variable that is used as an argument
    - Some languages prohibit a call to `f` if the argument is *not* a variable. Others just create a temporary and initialize it to the value of the argument

Passing pointers

- Call-by-reference can be implemented using pointers
- Our running example becomes:

```
void f(int* x) { *x = *x+1; }
x = 2; f(&x); print(x);
```

which again produces an output of 3

- Call by reference differs in the following ways:
    - The compiler takes care of figuring out where pointers must be dereferenced/followed
    - There is no possibility of having a null pointer argument passed in as a parameter

Further comments

- Call-by-reference can be used to define functions that return multiple values:

```
A f(B x, C& y) {  // returns an A and a C
   ...
}
```

- But this is a little clunky: perhaps we didn't want to have to pass in an initial value in `y`
- Some languages allow function parameters to be labeled with "in", "out", or "inout" annotations
- Other parameter passing methods include: call-by-name; call-by-need ("lazy evaluation"); call-by-text; call-by-copy-restore; ...

Aside: a curious asymmetry

- Most (all?) programming languages have some notion of functions ...
- ... and most of these allow functions with multiple arguments ...
- ... but relatively few allow functions with multiple results ...
- There is no technical reason for this!

Implementing Objects

Implementing objects

We have made considerable use of objects and classes in this course. How are these features implemented?

- How are objects represented?
- How do we access the instance variables of an object?
- How do we call an object's methods?
- How does inheritance (derived classes) work?
- How does a program choose which version of a method body (virtual function) should be executed?

Representing objects

- Simple structures, for example:

```
class Date {
   int day;
   int month;
   int year;
}
```

can be represented by a single block of memory, divided into fields:

day	month	year

- ... this is much like the implementation of arrays except that, at the language level, different types (and sizes) of elements may appear in a single structure while all elements of an array have the same type

Accessing fields

- The layout of a structure can be described by an environment mapping variables to pairs containing the type and offset of each field:

  { day ⟼ (int, 0), month ⟼ (int, 8), year ⟼ (int, 16) }

- This information is computed during static analysis and is stored as part of the representation of the Date type

Checking and compiling fields

- To compile an expression e.m we need to check that:
  - e evaluates to an object of some class type T
  - T contains a field m of type S with offset o
- If a is the address that e produces, then the variable e.m of type S will be stored at address (a + o)

Member functions

We can associate functions with classes:

```
class Date {
   int day;
   int month;
   int year;
   void set(int, int, int);
   ...
}

Date today;
...
today.set(11, 5, 2015);
```

Putting the definitions of the functions in the same place as the type definition makes the relationship between the different entities more explicit

Compiling member functions

Member functions can be treated just like ordinary functions except that:

- The compiled assigns a new name to the function, known only to the compiler, to avoid clashes with other global or member functions of the same name
  - (Also known as "name mangling")
- The compiler adds an extra argument to the function that provides a pointer to a specific object of the associated type
  - (The parameter is often referred to as this or self)

For example

The following program:

```
class Date {
   int day;
   int month;
   int year;
   void set(int d, int m, int y) {
      day = d; month = m; year = y;
   }
   ...
}

Date today;
...
today.set(11, 5, 2015);
```

can be implemented by ...

For example

... the following code:

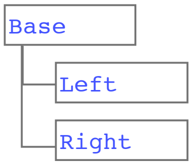
```
class Date {
   int day;
   int month;
   int year;
   void mf_set_Date(Date this, int d, int m, int y) {
      this.day = d; this.month = m; this.year = y;
   }
   ...
}

Date today;
...
mf_set_Date(today, 11, 5, 2015);
```

Note that references to a component x of a class in a member function are just abbreviations for this.x

Derived classes (subclasses)

Consider a simple hierarchy of classes:

```
class Base {
   int x, y;
}
class Left extends Base {
   int l;
}
class Right extends Base {
   int r;
}
```

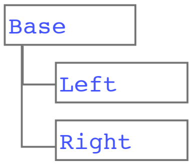


How should we represent objects of the classes Base, Left, and Right?

Nested representations?

Nest one structure inside another:

```
class Base {
   int x, y;
}

class Left extends Base {
   Base super;
   int  l;
}

class Right extends Base {
   Base super;
   int  r;
}
```



If l is an object of type Left, then references to l.x in the source are translated to l.super.x

Overlapping layouts

- Or perhaps we can be more careful about object layout to make sure that the common parts of a structure are always stored at the same offsets in related classes

Base	x	y	
Left	x	y	l
Right	x	y	r

- The x field is always at offset 0, y is always at offset 8, etc...
- Remember: we can always use a (pointer to a) Right or Left object where a Base is required

Virtual functions/methods

Suppose that we extend the current example to include virtual functions/methods:

```
class Base {
   int x, y;
   Base() {...}
   int f(int i) {...}
   void g() {...}
}
```

```
class Left extends Base {
   int l;
   Left() {...}
   int f(int i) {...}
}

class Right extends Base {
   int  r;
   Right() {...}
   void g() {...}
}
```

If b has Base, how do we implement a call b.f(y)?

Calling a virtual function b.f(y)

- If f were an ordinary member function, then the meaning of the call would be determined by the type of b:
  - For example, if b has type Base, then we would arrange to call the f function that is defined in the Base class; this can be determined at compile time
- If f is a virtual function, then there are other possibilities to consider:
  - If b was constructed using new Base() or new Right(), then we want to use the implementation in Base
  - If b was constructed using new Left(), then we want to use the implementation in Left
- How do we choose?

Storing tags in objects

- One approach is to store an extra tag field in each object:

Base	BASE	x	y	
Left	LEFT	x	y	l
Right	RIGHT	x	y	r

- It doesn't really matter what these tags are (they could be integers, pointers, ...) so long as different classes use different tags so that we can tell them apart

… and using a lookup table

- In addition, we store a table listing the versions of each function that should be called, depending on the object tag:

	BASE	LEFT	RIGHT
F	Base.f	Left.f	Base.f
G	Base.g	Base.g	Right.g

- Now we can implement a call like `b.f(y)` using:

```
(lookup[tag(b)][F])(b,y)
```

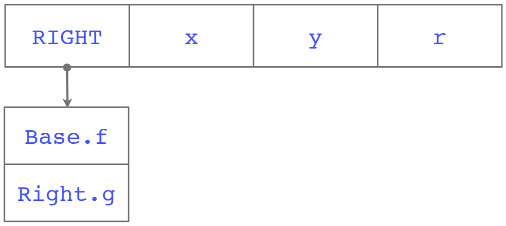
Using arrays and tags

This method has some problems:

- If we are compiling just one file in a large program, how do we figure out the dimensions of the lookup table? How do we choose appropriate values for the indices?
- Assuming that we solve the first problem, there are likely to be a lot of unused entries in the table, which is a waste of space.

Virtual function tables

- By contrast, we do know exactly how many virtual functions there are for any given class:



- Now we can implement a call like `b.f(y)` using:

```
((b.tag)[0])(b,y)
```

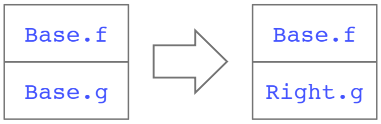
… continued

- The array of virtual function addresses is often called a virtual function table (vtable)
- The same virtual function table can be shared by all instances of a given class; a suitable pointer will be loaded in to the appropriate position when storage for that object is first allocated
- The vtable structure can also be used to store other details about the object structure (such as its `size`, or `forward` and `scavenge` methods for garbage collection purposes)
- We need to fix the order of the entries in the vtable at compile time

Inheritance

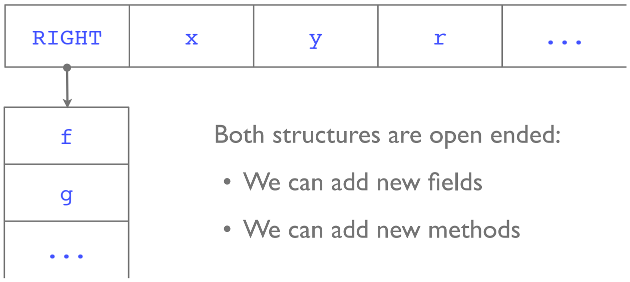
The virtual function table for a derived class can be obtained from the table for the base class

- Copy entries that are not overridden in the derived class
- Replace entries corresponding to new implementations of base class virtual functions
- Append new entries for new virtual functions



Extensibility

The beauty of this scheme is its extensibility. Any object that is derived from `Right` will have at least the following structure:

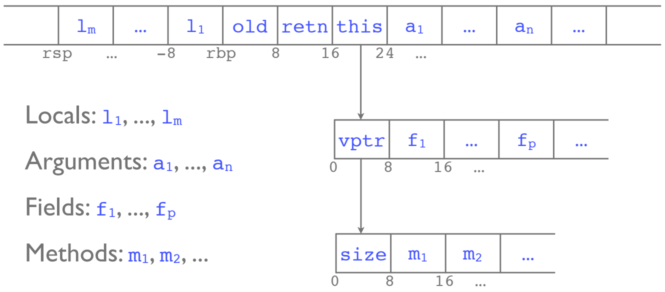


Both structures are open ended:

- We can add new fields
- We can add new methods

Method stack frames

Combining these ideas with the choices of data structures that we've seen for implementing simple functions, the stack frame for a typical method looks something like this:



Locals: l_1, …, l_m

Arguments: a_1, …, a_n

Fields: f_1, …, f_p

Methods: m_1, m_2, …

Example

One possible code sequence for a call $\texttt{this.m}_2(\texttt{f}_1,42)$ is as follows (assuming this is at offset 16 from rbp):

```
subq $24, %rsp          # space for parameters
movq 16(%rbp), %rax     # get "this" in rax
movq %rax, (%rsp)       # save m2's "this" param
movq 8(%rax), %rdx      # get value of f1
movq %rdx, 8(%rsp)      # and save as arg 2
movq $42, 16(%rsp)      # set 42 as arg 3
movq (%rax), %rdx       # vtable address in rdx
movq 8(%rdx), %rdx      # address for m2
call *%rdx              # call the function
addq $12, %rsp          # remove parameters
```

Summary: Objects

- Objects can be implemented by contiguous blocks of storage, with fields referenced by their offsets from the start of the object
- Member functions can be implemented by passing an extra argument, called this, as an implicit parameter
- Virtual functions can be implemented using virtual function tables
- Extensibility is built in to the data structures

Higher-order / First-class functions

Higher-order / first-class functions

- Functions that take other functions as arguments or return other functions as results are sometimes called higher-order functions:

```
int(int) f(int x, int y) {
    int g(int z) { return x+z; }
    return g;
}
```

a type for functions that take an int argument and return an int result

free variable

- In this case, g may be called after f has returned
- … so g may need to access f's x parameter after f has returned
- … so we can't just dispose of stack frames as soon as a function exits.

Moving from the stack to the heap …

- A solution in this case is to allocate the "stack frame" for f on the heap, and not on the stack
  - Frames like this are sometimes referred to as "activation records"
  - Unused activation records can be recovered by garbage collection instead of by popping them off the stack
- Alternatively, if we don't keep the stack frames in the heap, then we will need to save a copy of x's value in the representation for g before f returns
- Variants of these schemes are used in many functional language implementations (and, increasingly now, also in other settings such as C++, Python, and Javascript implementations)

Lambda expressions

Lambda expressions (anonymous functions) provide a way to write functions without giving them a name.

Haskell	`\x -> x + 1`	Javascript	`function (x) x + 1`

LISP	`(lambda (x)`
	`(+ x 1))`
		C++ 11	`[] (int x) -> int`
			`{ return x + 1; }`

Python	`lambda x: x + 1`	Smalltalk	`[ :x | x + 1]`

Example

- The previous example:

```
int(int) f(int x, int y) {
    int g(int z) { return x+z; }
    return g;
}
```

- Rewritten using a lambda expression:

```
int(int) f(int x, int y) {
    return (\z -> x+z);
}
```

Using function values

- A general purpose "mapping" primitive:

```
void mapArray(int(int) f, int[] arr) {
    for (int i=0; i<arr.length; i++) {
        arr[i] = f(arr[i]);
    }
}
```

Note that this `f` is a parameter of `mapArray`, not a known function

- To increment every element in an array, `arr`:

```
mapArray(\x -> x + 1, arr);
```

- To double every element in an array, `arr`:

```
mapArray(\x -> x * 2, arr);
```

- Etc...

Composing function values

- A general purpose "composition" primitive:

```
(int)int compose(int(int) f, int(int) g) {
    return \x -> f (g x);
}
```

- Using `compose`, we can combine two separate mapping operations:

```
mapArray(g, arr);
mapArray(f, arr);
```

- Into a single iteration across the array:

```
mapArray(compose(f, g), arr);
```

- Etc...

Representing function values

- How should we represent values of type `int(int)`?
  - There are many different values, including: `(\z -> x+z)`, `(\x -> x+1)`, `(\x -> x*2)`, `(\x -> f(g(x)))`, ...
  - ... any of which could be passed as arguments to functions like `mapArray` or `compose`, ...
  - ... so we need a uniform, but flexible way to represent them
- A common answer is to represent functions like these by a pointer to a "closure", a heap allocated object that contains:
  - a code pointer (i.e., the code for the function)
  - the values of its free variables
- Because we're making copies of the free variables, we usually require them to be immutable (e.g., final/const)

Closures

- *Every* function of type `int(int)` will be represented using the same basic structure:

`codeptr`	`...`

- The code pointer and list of variables vary from one function value to the next:

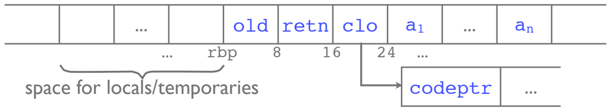
`(\z -> x+z)`	codeptr$_1$	`x`	
`(\x -> x+1)`	codeptr$_2$		
`(\x -> x*2)`	codeptr$_3$		
`(\x -> f(g(x)))`	codeptr$_4$	`f`	`g`

- To make a closure, allocate a suitably sized block of memory and save the required code pointer and variable values

Calling unknown functions

- If `f` is a known function, then we call it by pushing its arguments on the stack and jumping directly to its code
- What if `f` is an unknown function, represented by a variable that points to a closure structure instead?
- The System V ABI doesn't cover this case, but we can make up our own convention:
  - push the arguments
  - push a pointer to the closure (for access to free variables)
  - call the code that is pointed to at the start of the closure

Entering a closure



- Example: if `f` is an expression of type `int(int)`, then we can compile a call of the form `f(e)` using the following code:

```
Ers⟦e⟧(%rax)
pushq  %rax
Ers⟦f⟧(%rax)
pushq  %rax
movq   (%rax), %rax
call   *%rax
addq   $16, %rsp
```

evaluate and push argument

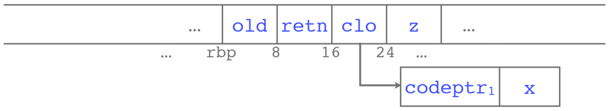
evaluate function

push closure

"enter" closure

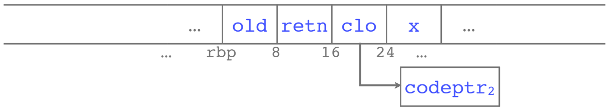
remove parameters

Implementation of `(\z -> x+z)`



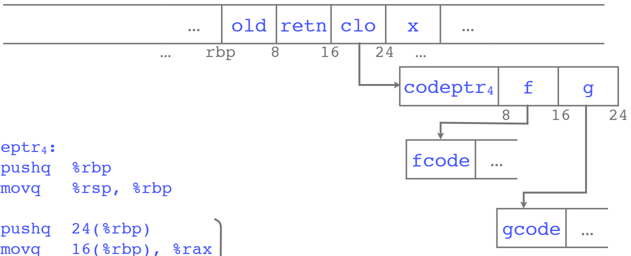
```
codeptr₁:
    pushq  %rbp
    movq   %rsp, %rbp

    movq   16(%rbp), %rax -- clo
    movq   8(%rax), %rax  -- x
    addq   24(%rbp), %rax -- z

    movq   %rbp, %rsp
    popq   %rbp
    ret
```

Implementation of `(\x -> x+1)`



```
codeptr₂:
    pushq  %rbp
    movq   %rsp, %rbp

    movq   24(%rbp), %rax
    incq   %rax

    movq   %rbp, %rsp
    popq   %rbp
    ret
```

Implementation of `(\x -> f(g(x)))`



```
codeptr₄:
    pushq  %rbp
    movq   %rsp, %rbp

    pushq  24(%rbp)
    movq   16(%rbp), %rax
    movq   16(%rax), %rax
    pushq  %rax
    call   *(%rax)
    addq   $16, %rsp
```

g(x)

```
    pushq  %rax
    movq   16(%rbp), %rax
    movq   8(%rax), %rax
    pushq  %rax
    call   *(%rax)

    movq   %rbp, %rsp
    popq   %rbp
    ret
```

f(g(x))

Closures vs. objects

- Invoking an unknown function through a closure is very similar to invoking a method of an object …
  - Method invocations pass the object itself as an implicit argument (just as we are pass the closure pointer here)
  - Closures are like objects with a single method
  - Free variables correspond to object fields
- In Java 8, lambda expressions are "just" a convenient way to write (local, anonymous) definitions of single-method classes
  - Very useful for GUI call-backs, aggregate operations, concurrency libraries, etc…

Simulating closures in Java

We can simulate the use of closures using Java classes:

```
abstract class IntToInt {               //   (int)int type
   abstract int of(int arg);
}

class PlusOne extends IntToInt {        //   \x -> x + 1
    int of(int arg) { return arg + 1; }
}

class TimesTwo extends IntToInt {       //   \x - > x * 2
    int of(int arg) { return arg * 2; }
}

class PlusX extends IntToInt {          //   \z -> x + z
    private int x;
    PlusX(int x)     { this.x = x; }
    int of(int arg) { return x + arg; }
}
```

Mapping over an array

- A general purpose "mapping" primitive:

```
static void mapArray(IntToInt f, int[] arr) {
    for (int i=0; i<arr.length; i++) {
        arr[i] = f.of(arr[i]);
    }
}
```

- To increment every element in an array:

```
mapArray(new PlusOne(), arr);
```

- Using an "inner" class:

```
mapArray(new IntToInt() {
              int of(int arg) { return arg+1; }
            }, arr);
```

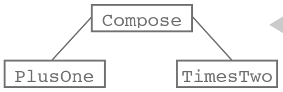
Composing functions

- A general purpose "composition" primitive:

```
class Compose extends IntToInt {
    private IntToInt f;
    private IntToInt g;
    Compose(IntToInt f, IntToInt g)
      { this.f = f; this.g = g; }
    int of(int arg) { return f.of(g.of(arg)); }
}
```

- To replace each value in an array with a corresponding odd number:

```
mapArray(new Compose(new PlusOne(), new TimesTwo()), arr);
```



abstract syntax, with an interpreter/eval function called "of"!

Summary: First-class functions

- More sophisticated forms of function can be supported by modifying or generalizing the ways that stack frames are used
- For a language with higher-order functions, we need to allocate activation records on the heap, or copy data to other heap-allocated objects, because a function value may have a longer lifetime than the function that created it
- Function values can be represented by closure objects that pair a code pointer with a list of variable values
- Invoking an unknown function through a closure is very similar to invoking a method of an object ...
- These techniques are key tools in the implementation of functional (and, increasingly, OO) programming languages

Nested functions

Nested functions

- Sometimes it is useful to allow function definitions to be nested, one inside another

```
int f(int x, int y) {
    int square(int z) { return z*z; }
    return square(x) + square(y);
}
```

- This might, for example, be used to introduce a local function, `square`, without making it more widely visible
- Java, (standard) C, C++ do not allow this, but ML, Pascal, Haskell, and many other languages do ...

A more complicated example: quicksort

```
void sort(File inp, File out) {
  int[] a;
  ... a ... readArray ... quicksort ... writeArray ...
}

void readArray(inp, a)  { ... inp ... a ... }
void writeArray(a, out) { ... a ... out ... }

void quicksort(a, lo, hi) {
  int pivot = ...;
  ... a ... pivot ... partition ... quicksort ...
}

void partition(a, pivot, lo, hi) {
   ... a ... pivot ... swap ...
}

void swap(a, i, j) { ... a[i] ... a[j] ...}
```

A more complicated example: quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray(inp, a)  { ... inp ... a ... }
  void writeArray(a, out) { ... a ... out ... }

  ... a ... readArray ... quicksort ... writeArray ...
}

void quicksort(a, lo, hi) {
  int pivot = ...;
  ... a ... pivot ... partition ... quicksort ...
}

void partition(a, pivot, lo, hi) {
   ... a ... pivot ... swap ...
}

void swap(a, i, j) { ... a[i] ... a[j] ...}
```

Move `readArray` and `writeArray` in to `sort`

A more complicated example: quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray()  { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  ... a ... readArray ... quicksort ... writeArray ...
}

void quicksort(a, lo, hi) {
  int pivot = ...;
  ... a ... pivot ... partition ... quicksort ...
}

void partition(a, pivot, lo, hi) {
   ... a ... pivot ... swap ...
}

void swap(a, i, j) { ... a[i] ... a[j] ...}
```

parameters are no longer required!

A more complicated example: quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray()  { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  ... a ... readArray ... quicksort ... writeArray ...
}

void quicksort(a, lo, hi) {
  int pivot = ...;
  void partition(a, pivot, lo, hi) {
     ... a ... pivot ... swap ...
  }
  void swap(a, i, j) { ... a[i] ... a[j] ...}

  ... a ... pivot ... partition ... quicksort ...
}
```

Move `partition` and `swap` in to `quicksort`

A more complicated example: quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray()  { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  ... a ... readArray ... quicksort ... writeArray ...
}

void quicksort(a, lo, hi) {
  int pivot = ...;
  void partition() {
     ... a ... pivot ... swap ...
  }
  void swap(i, j) { ... a[i] ... a[j] ...}

  ... a ... pivot ... partition ... quicksort ...
}
```

again, fewer parameters are required!

A more complicated example: quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray()  { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  ... a ... readArray ... quicksort ... writeArray ...
}

void quicksort(a, lo, hi) {
  int pivot = ...;
  void partition() {
     void swap(i, j) { ... a[i] ... a[j] ...}
     ... a ... pivot ... swap ...
  }

  ... a ... pivot ... partition ... quicksort ...
}
```

Move `swap` in to `partition`

A more complicated example: quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray()  { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  void quicksort(a, lo, hi) {
    int pivot = ...;
    void partition() {
       void swap(i, j) { ... a[i] ... a[j] ...}
       ... a ... pivot ... swap ...
    }

    ... a ... pivot ... partition ... quicksort ...
  }

  ... a ... readArray ... quicksort ... writeArray ...
}
```

Move quicksort in to sort

A more complicated example: quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray()  { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  void quicksort(lo, hi) {
    int pivot = ...;
    void partition() {
       void swap(i, j) { ... a[i] ... a[j] ...}
       ... a ... pivot ... swap ...
    }

    ... a ... pivot ... partition ... quicksort ...
  }

  ... a ... readArray ... quicksort ... writeArray ...
}
```

yet again, fewer parameters are required!

how can we compile code like this?

Free variables

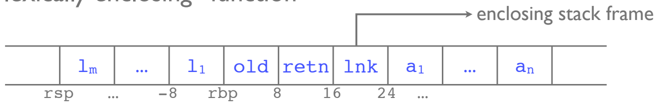
- The challenge here is in dealing with nested functions that access variables that are defined in enclosing functions

```
int f(int x, int y) {
     int g(int z) { return x+z; }
     return g(x+y);
}
```

- For example, x is said to be "bound" in the definition of f, but "free" in the definition of g
- The code for g refers to a variable that is not in its stack frame

Static links

- To accommodate languages like this, we will add an extra field to each stack frame that points to the stack frame of the "lexically enclosing" function



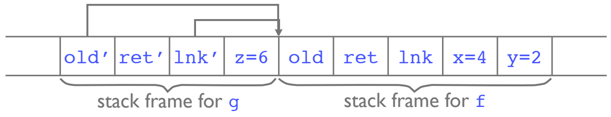
- This is known as a static link (or access link)
  - "**static**" because it points to stack frame of the enclosing function (which can be determined at compile time)
  - by comparison, the old base pointer provides a "**dynamic**" link: it points to the stack frame of the calling function (which is determined at run time)

Example

Given the earlier definition:

```
int f(int x, int y) {
     int g(int z) { return x+z; }
     return g(x+y);
}
```

and a call f(4,2), the stack might look something like the following during the call to g



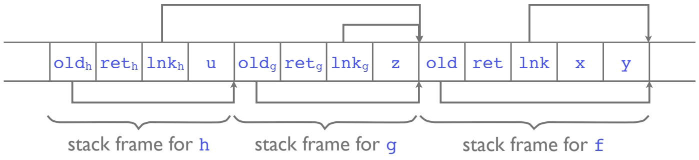
In particular, the frame containing the value for x can be found by following the static link, lnk'

Static link ≠ base pointer

- Suppose we have the definition:

```
int f(int x, int y) {
     int g(int z) { return h(x+h(z)); }
     int h(int u) { return y*u; }
     return g(x+y);
}
```

- The static link that g uses in calls to h points to the stack frame for f, not the stack frame for g



Lexical depth

- A function at the top level (i.e., with no enclosing function) has lexical depth 0 (and does not need a static link)
- A function that appears inside the definition of a function with depth n has depth n+1
- To access a variable/call a function at depth n, from a function at depth m (note that n≤m), then we have to follow the static link in the current frame (m-n) times

Lexical depths for quicksort

```
0 void sort(File inp, File out) {
    int[] a;
1   void readArray()  { ... inp ... a ... }
1   void writeArray() { ... a ... out ... }

1   void quicksort(lo, hi) {
      int pivot = ...;
2     void partition() {
3       void swap(i, j) { ... a[i] ... a[j] ...}
        ... a ... pivot ... swap ...
      }

      ... a ... pivot ... partition ... quicksort ...
    }

    ... a ... readArray ... quicksort ... writeArray ...
  }
```

Lexical depths for our simpler example: f

- Suppose we have the definition:

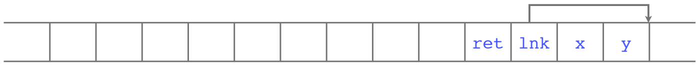
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

Lexical depths for our simpler example: f

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
f: pushq   %rbp
   movq    %rsp, %rbp

   movq    24(%rbp), %rax
   addq    32(%rbp), %rax
   pushq   %rax

   pushq   %rbp

   call    g
   addq    $16, %rsp

   movq    %rbp, %rsp
   popq    %rbp
   ret
```



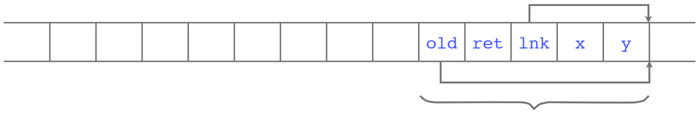
Lexical depths for our simpler example: f

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

save old base pointer

```
f: pushq   %rbp
   movq    %rsp, %rbp

   movq    24(%rbp), %rax
   addq    32(%rbp), %rax
   pushq   %rax

   pushq   %rbp

   call    g
   addq    $16, %rsp

   movq    %rbp, %rsp
   popq    %rbp
   ret
```



stack frame for f

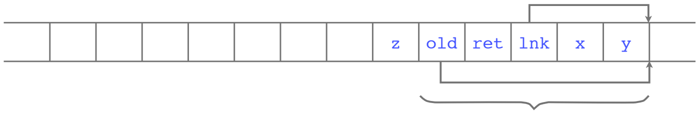
Lexical depths for our simpler example: f

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

calculate z=x+y

```
f: pushq   %rbp
   movq    %rsp, %rbp

   movq    24(%rbp), %rax
   addq    32(%rbp), %rax
   pushq   %rax

   pushq   %rbp

   call    g
   addq    $16, %rsp

   movq    %rbp, %rsp
   popq    %rbp
   ret
```



stack frame for f

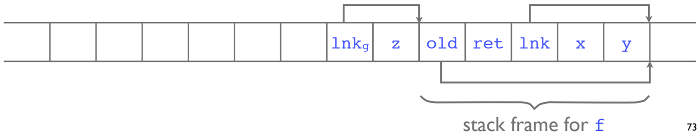
Lexical depths for our simpler example: f

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
f: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax
   addq   32(%rbp), %rax
   pushq  %rax

   pushq  %rbp

   call   g
   addq   $16, %rsp

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

static link



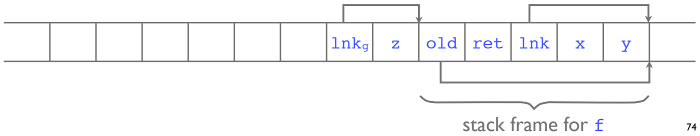
Lexical depths for our simpler example: f

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
f: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax
   addq   32(%rbp), %rax
   pushq  %rax

   pushq  %rbp

   call   g
   addq   $16, %rsp

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

call g with this frame as static link



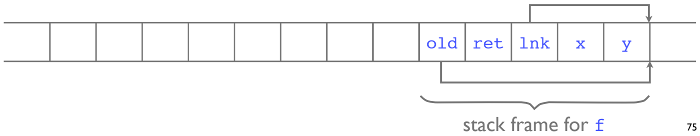
Lexical depths for our simpler example: f

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
f: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax
   addq   32(%rbp), %rax
   pushq  %rax

   pushq  %rbp

   call   g
   addq   $16, %rsp

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

remove parameters from stack



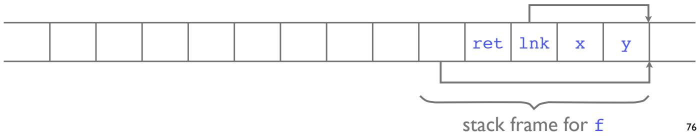
Lexical depths for our simpler example: f

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
f: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax
   addq   32(%rbp), %rax
   pushq  %rax

   pushq  %rbp

   call   g
   addq   $16, %rsp

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

restore old base pointer



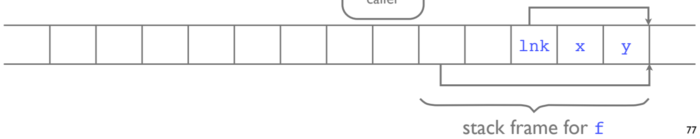
Lexical depths for our simpler example: f

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
f: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax
   addq   32(%rbp), %rax
   pushq  %rax

   pushq  %rbp

   call   g
   addq   $16, %rsp

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

return to caller

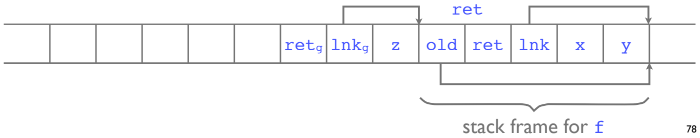


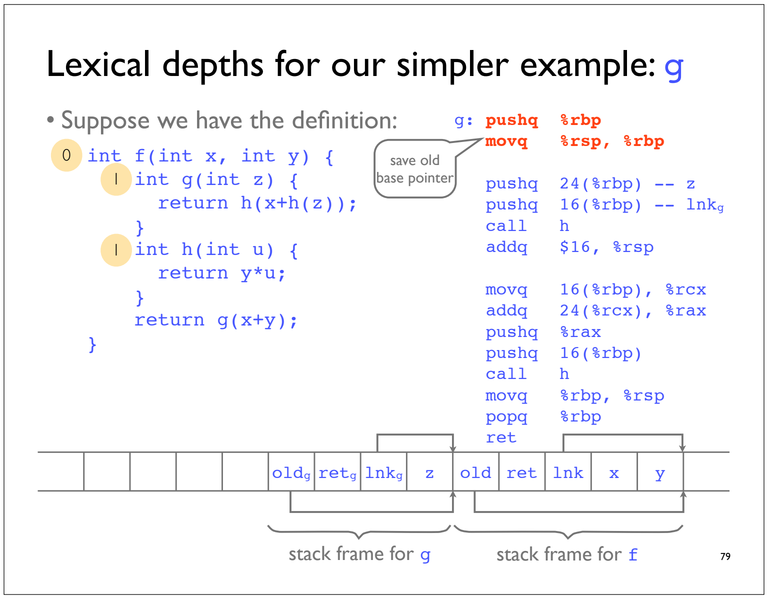
Lexical depths for our simpler example: g

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp)  -- z
   pushq  16(%rbp)  -- lnk_g
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```





Lexical depths for our simpler example: g

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

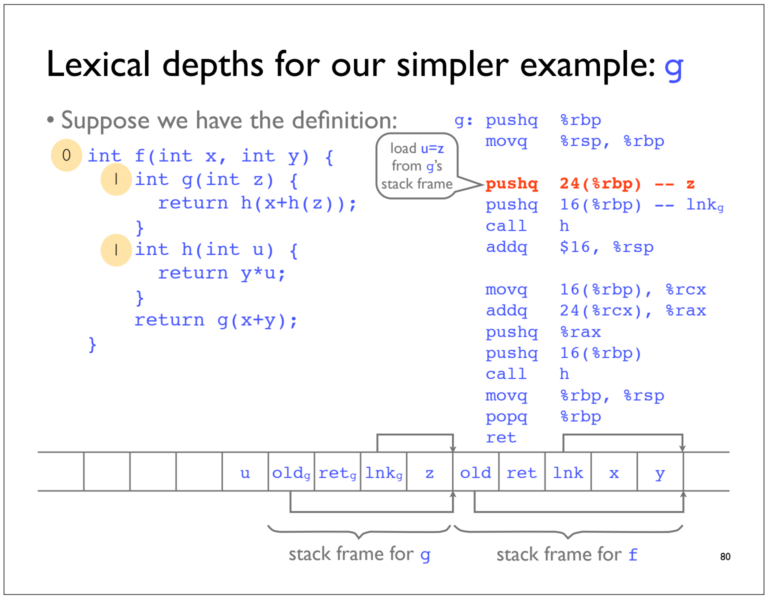
```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnkg
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

save old base pointer

					old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for g   stack frame for f

79



Lexical depths for our simpler example: g

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

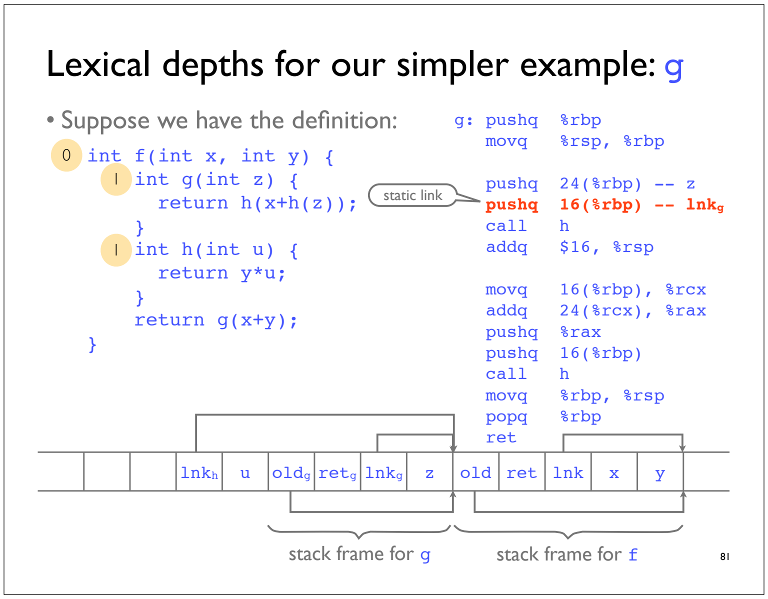
```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnkg
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

load u=z from g's stack frame

				u	old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for g   stack frame for f

80



Lexical depths for our simpler example: g

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

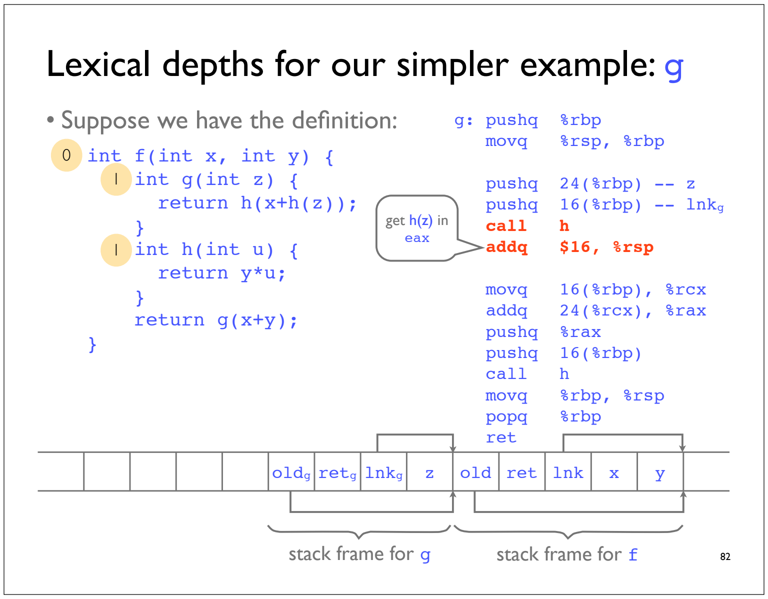
```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnkg
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

static link

			lnk$_h$	u	old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for g   stack frame for f

81



Lexical depths for our simpler example: g

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

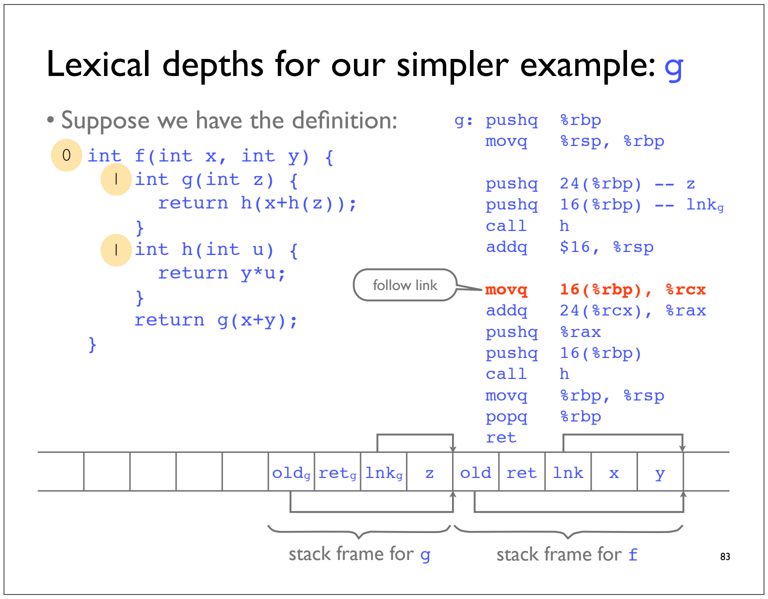
```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnkg
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

get h(z) in eax

					old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for g   stack frame for f

82



Lexical depths for our simpler example: g

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

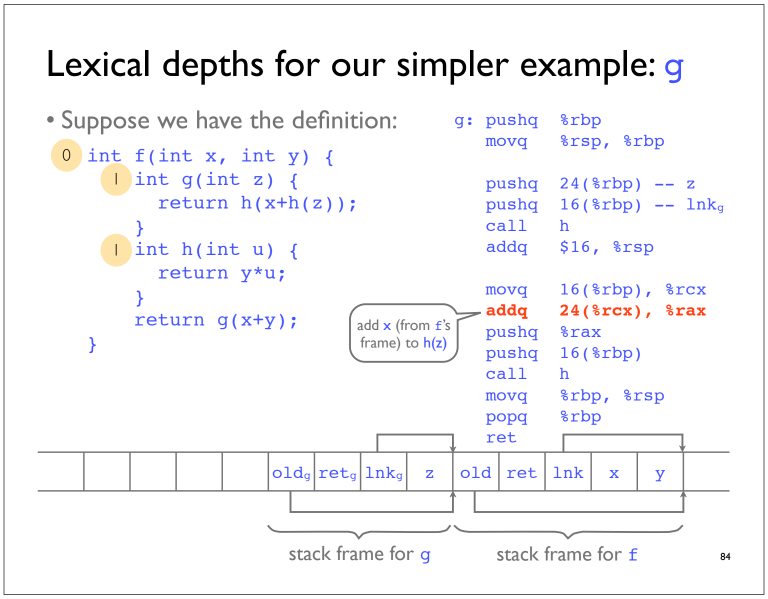
```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnkg
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

follow link

					old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for g   stack frame for f

83



Lexical depths for our simpler example: g

- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

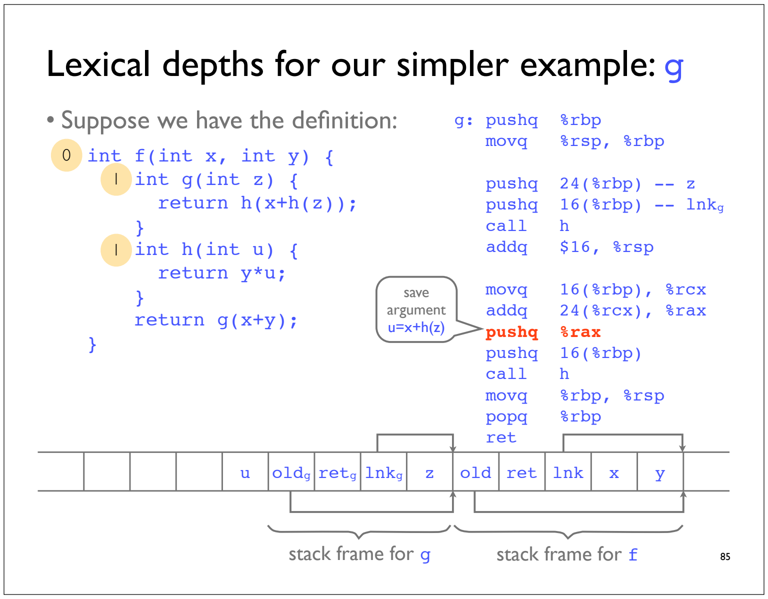
```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnkg
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

add x (from f's frame) to h(z)

					old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for g   stack frame for f

84



Lexical depths for our simpler example: g

- Suppose we have the definition:

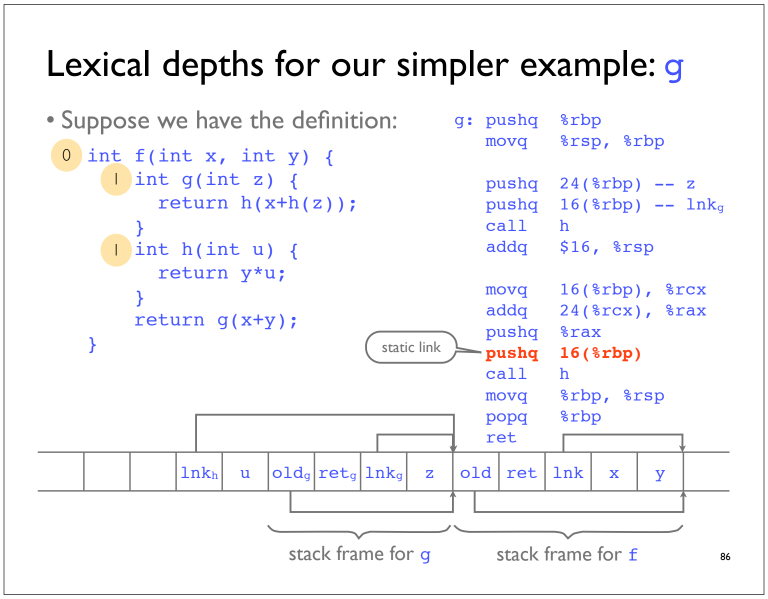
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnk_g
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

save argument u=x+h(z)

				u	old_g	ret_g	lnk_g	z	old	ret	lnk	x	y

stack frame for g   stack frame for f



Lexical depths for our simpler example: g

- Suppose we have the definition:

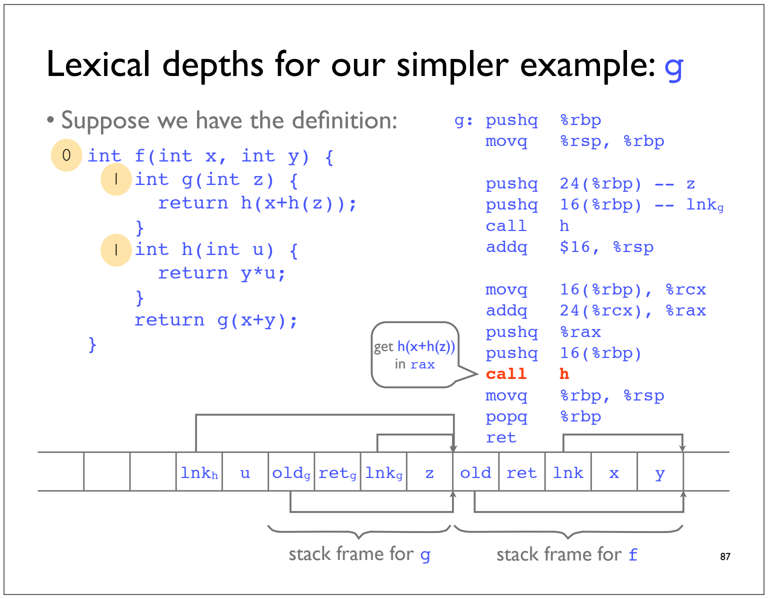
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnk_g
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

static link

			lnk_h	u	old_g	ret_g	lnk_g	z	old	ret	lnk	x	y

stack frame for g   stack frame for f



Lexical depths for our simpler example: g

- Suppose we have the definition:

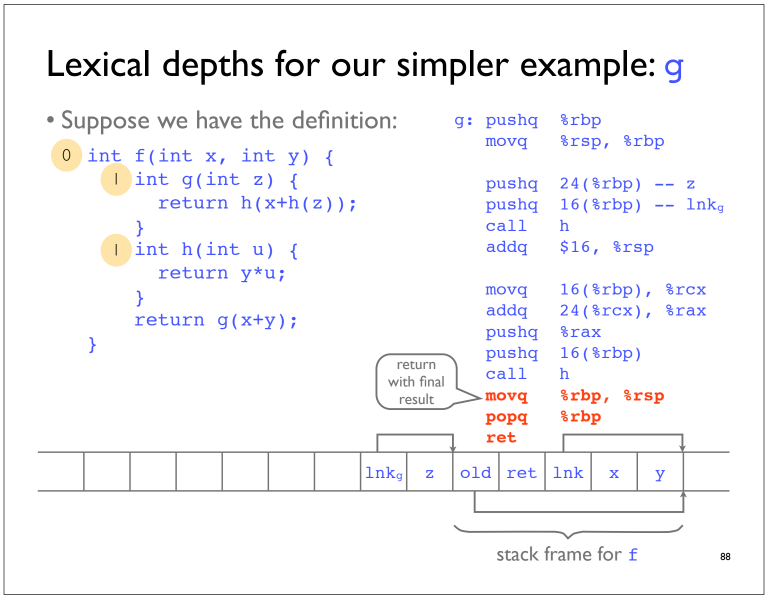
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnk_g
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

get h(x+h(z)) in rax

			lnk_h	u	old_g	ret_g	lnk_g	z	old	ret	lnk	x	y

stack frame for g   stack frame for f



Lexical depths for our simpler example: g

- Suppose we have the definition:

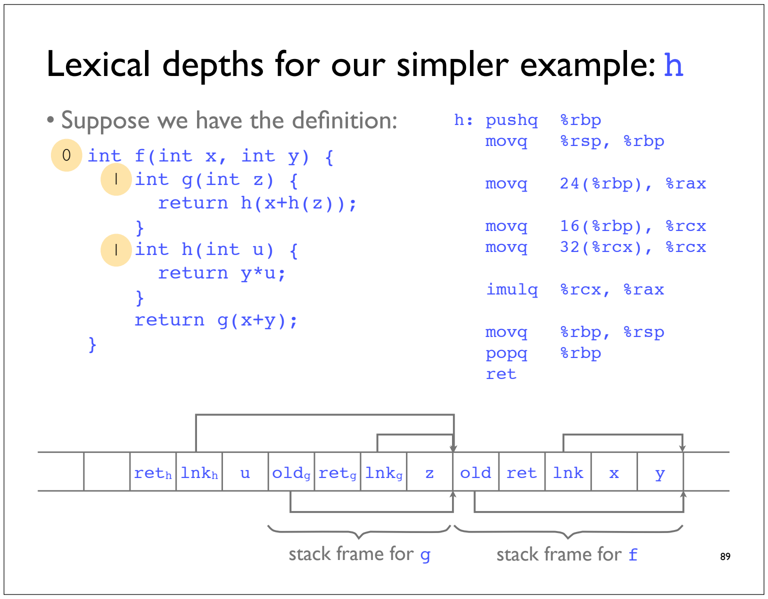
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
g: pushq  %rbp
   movq   %rsp, %rbp

   pushq  24(%rbp) -- z
   pushq  16(%rbp) -- lnk_g
   call   h
   addq   $16, %rsp

   movq   16(%rbp), %rcx
   addq   24(%rcx), %rax
   pushq  %rax
   pushq  16(%rbp)
   call   h
   movq   %rbp, %rsp
   popq   %rbp
   ret
```

return with final result

							lnk_g	z	old	ret	lnk	x	y

stack frame for f



Lexical depths for our simpler example: h

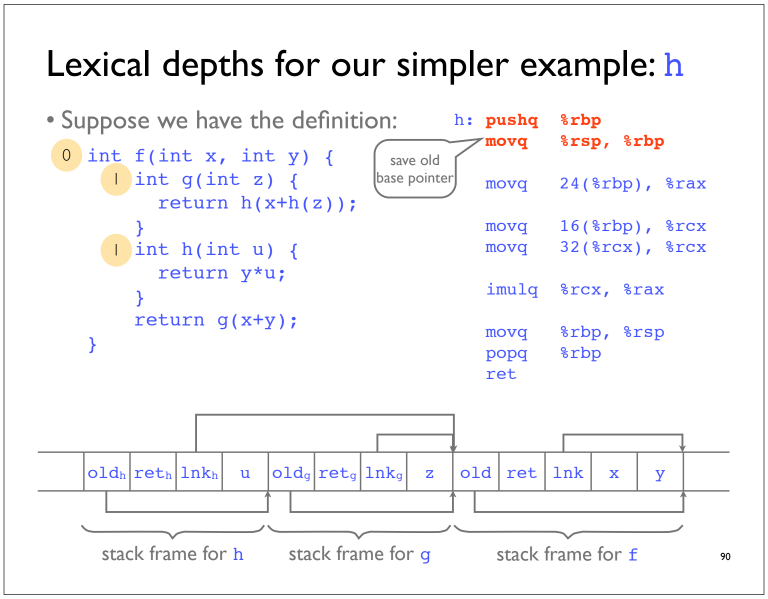
- Suppose we have the definition:

```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
h: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax

   movq   16(%rbp), %rcx
   movq   32(%rcx), %rcx

   imulq  %rcx, %rax

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

		ret_h	lnk_h	u	old_g	ret_g	lnk_g	z	old	ret	lnk	x	y

stack frame for g   stack frame for f



Lexical depths for our simpler example: h

- Suppose we have the definition:

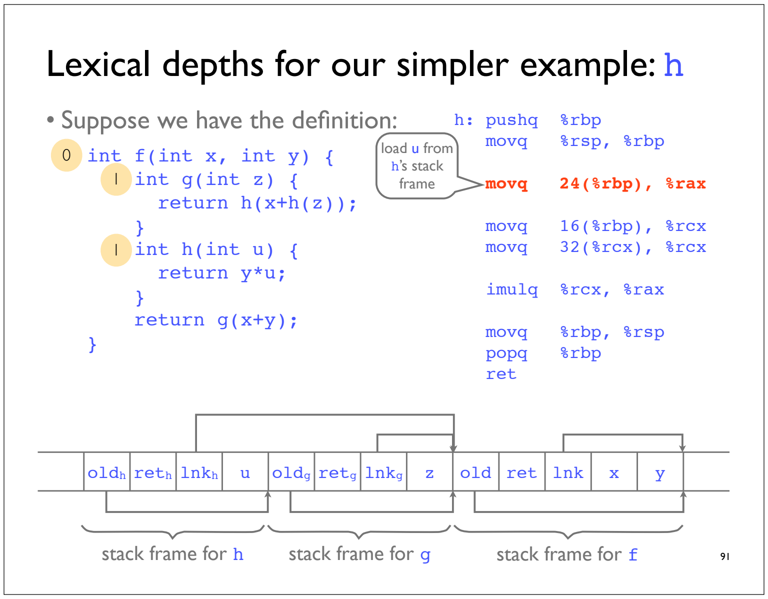
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
h: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax

   movq   16(%rbp), %rcx
   movq   32(%rcx), %rcx

   imulq  %rcx, %rax

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

save old base pointer

	old_h	ret_h	lnk_h	u	old_g	ret_g	lnk_g	z	old	ret	lnk	x	y

stack frame for h   stack frame for g   stack frame for f



Lexical depths for our simpler example: h

- Suppose we have the definition:

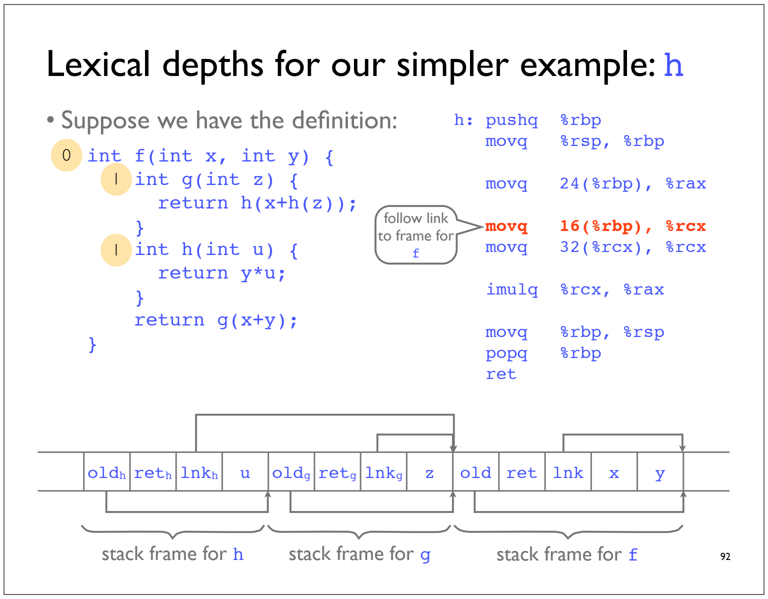
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
h: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax

   movq   16(%rbp), %rcx
   movq   32(%rcx), %rcx

   imulq  %rcx, %rax

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

load u from h's stack frame

old$_h$	ret$_h$	lnk$_h$	u	old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for h   stack frame for g   stack frame for f



Lexical depths for our simpler example: h

- Suppose we have the definition:

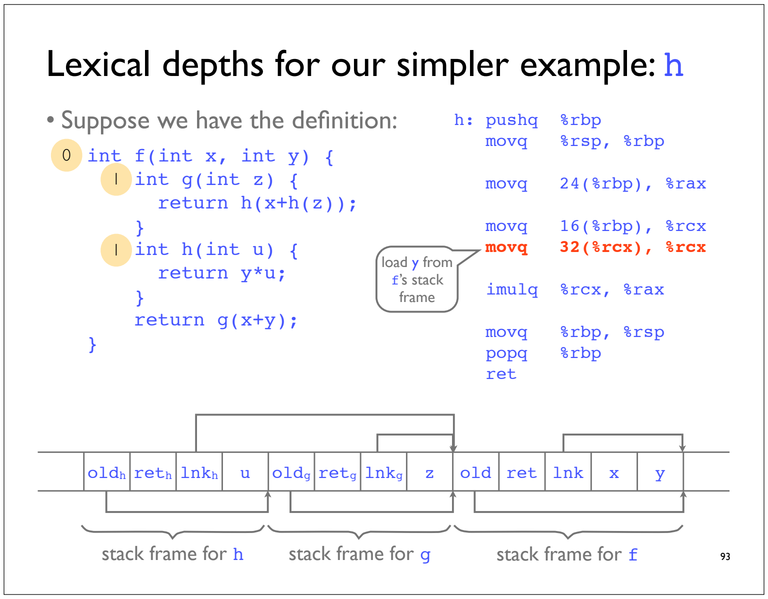
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
h: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax

   movq   16(%rbp), %rcx
   movq   32(%rcx), %rcx

   imulq  %rcx, %rax

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

follow link to frame for f

old$_h$	ret$_h$	lnk$_h$	u	old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for h   stack frame for g   stack frame for f



Lexical depths for our simpler example: h

- Suppose we have the definition:

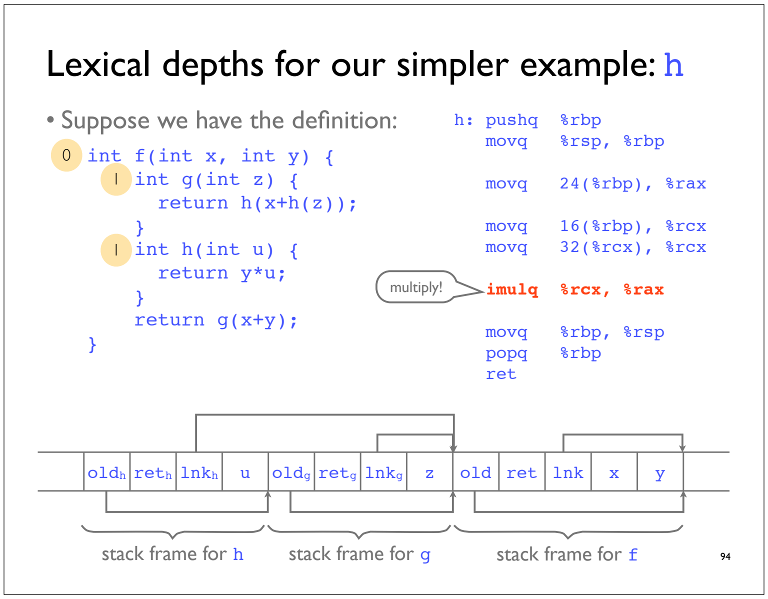
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
h: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax

   movq   16(%rbp), %rcx
   movq   32(%rcx), %rcx

   imulq  %rcx, %rax

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

load y from f's stack frame

old$_h$	ret$_h$	lnk$_h$	u	old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for h   stack frame for g   stack frame for f



Lexical depths for our simpler example: h

- Suppose we have the definition:

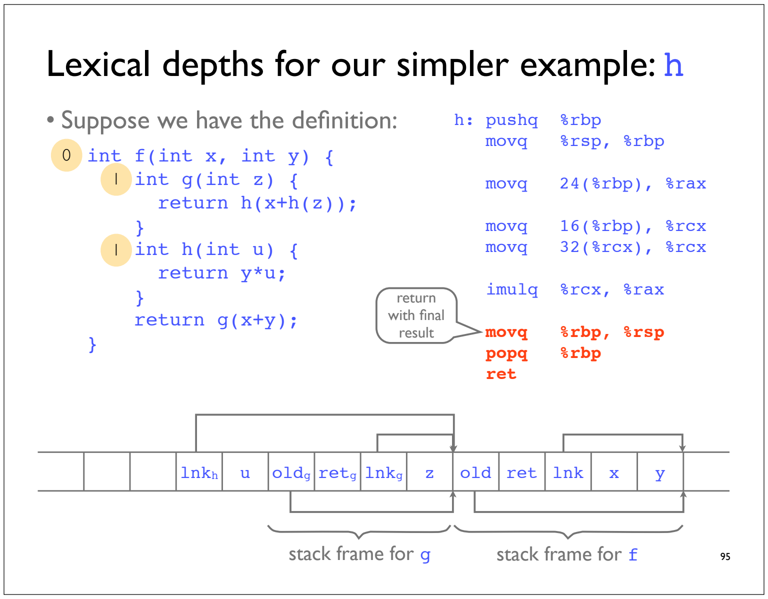
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
h: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax

   movq   16(%rbp), %rcx
   movq   32(%rcx), %rcx

   imulq  %rcx, %rax

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

multiply!

old$_h$	ret$_h$	lnk$_h$	u	old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for h   stack frame for g   stack frame for f



Lexical depths for our simpler example: h

- Suppose we have the definition:

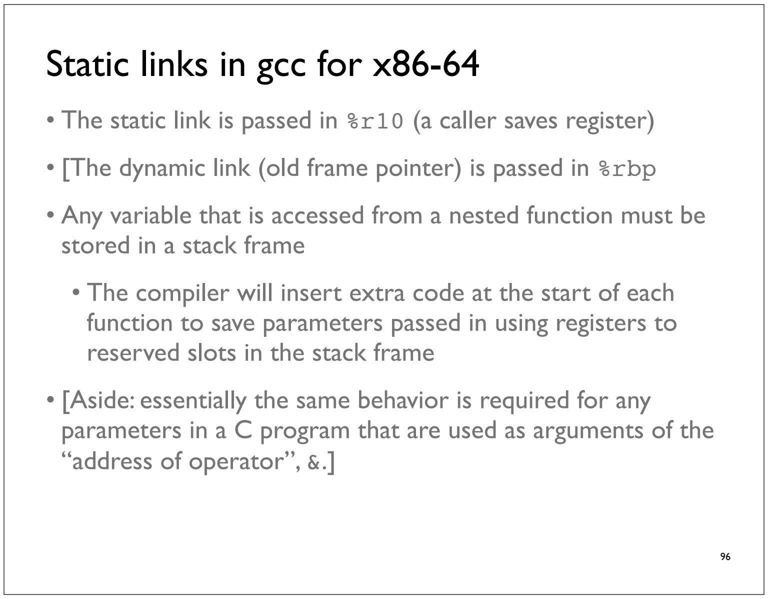
```
0 int f(int x, int y) {
1   int g(int z) {
      return h(x+h(z));
    }
1   int h(int u) {
      return y*u;
    }
    return g(x+y);
  }
```

```
h: pushq  %rbp
   movq   %rsp, %rbp

   movq   24(%rbp), %rax

   movq   16(%rbp), %rcx
   movq   32(%rcx), %rcx

   imulq  %rcx, %rax

   movq   %rbp, %rsp
   popq   %rbp
   ret
```

return with final result

		lnk$_h$	u	old$_g$	ret$_g$	lnk$_g$	z	old	ret	lnk	x	y

stack frame for g   stack frame for f



Static links in gcc for x86-64

- The static link is passed in `%r10` (a caller saves register)
- [The dynamic link (old frame pointer) is passed in `%rbp`
- Any variable that is accessed from a nested function must be stored in a stack frame
  - The compiler will insert extra code at the start of each function to save parameters passed in using registers to reserved slots in the stack frame
- [Aside: essentially the same behavior is required for any parameters in a C program that are used as arguments of the "address of operator", &.]

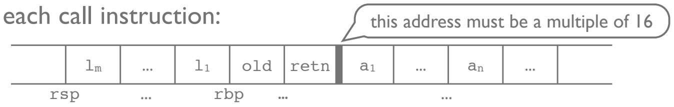
Summary: Nested functions

- We can support nested function definitions by adding an extra field (a "static link") to the stack frame that points to the frame of the enclosing function
- Variables defined in enclosing functions can be accessed by following (one or more) static links
- (provided that those variables are stored at known locations within the corresponding stack frames)

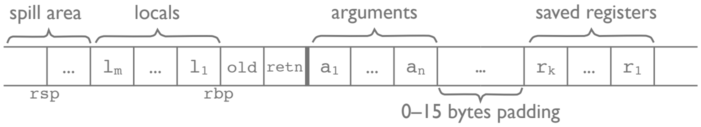
Alignment

Stack Alignment on Mac OS X

- Mac OS X requires that `%rsp` is 16 byte aligned just before each call instruction:



- Apple documentation makes the requirement clear but does not provide an explanation. Speculation seems to be that it has something to do with portability, or with the performance of multimedia instructions …
- This can be accomplished by padding the stack frame:


spill area
locals
arguments
saved registers
…
l_m
…
l_1
old
retn
a_1
…
a_n
…
r_k
…
r_1
rsp
rbp
0–15 bytes padding