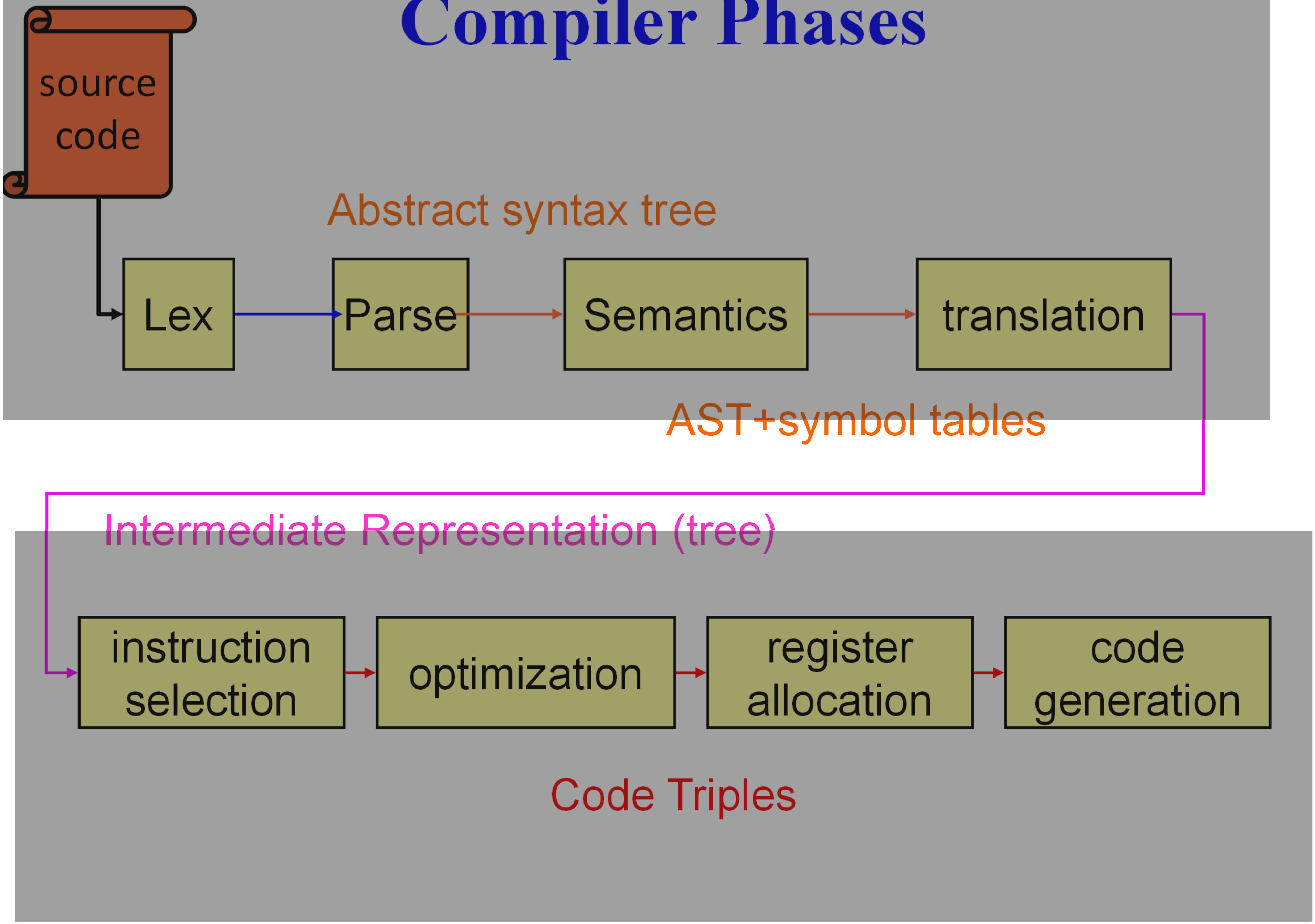
Functions:
Calling Conventions + Frames

15-411/15-611 Compiler Design

Seth Copen Goldstein

October 3, 2019

Compiler Phases



Today

- Calling Conventions
- Activation Frames
- IR for Function Calls
- Putting it all together

Lab3

- Function declarations and calls
- Typedefs
- assert

Understanding functions is the key.

What is the role of a Function?

- Provides an independent namespace
  - Parameters
  - Local variables
- Binds a name to an executable sequence
  Can be invoked with a call
- Provides illusion of custom instruction
  Control continues after call
- Interface to rest of world

- Job of compiler it create this abstraction from
  - A single PC
  - Byte addressed (single) memory space
  - Shared registers

Function as Contract

- Contract between
  - Architects
  - Compiler writers
  - Operating System
- Supports Interoperability
- Separate Compilation
- Plug-n-Play

Most Important part of the contract is between callers and callees.
The abstraction of the function is the key.

Benefits of “Function”

- Supports implementation and maintenance of large programs
  - Intellectual leverage (.e.g., decompose tasks)
  - Development efficiency (e.g., separate compilation)
- Supports cooperation of large independent systems (.e.g, O/S + Application)
- Supports Portability (e.g., libc)

What is the role of a Function?

- Provides an independent namespace
  - Parameters
  - Local variables
- Binds a name to an executable sequence
  Can be invoked with a call
- Provides illusion of custom instruction
  Control continues after call
- Interface to rest of world

- Job of compiler it create this abstraction from
  - A single PC
  - Byte addressed (single) memory space
  - Shared registers

Need to find code for bar

```
Foo: instr1
     instr2  x,y,z
     mov     z,a
     add     r3,r1,r2
     instr3
```

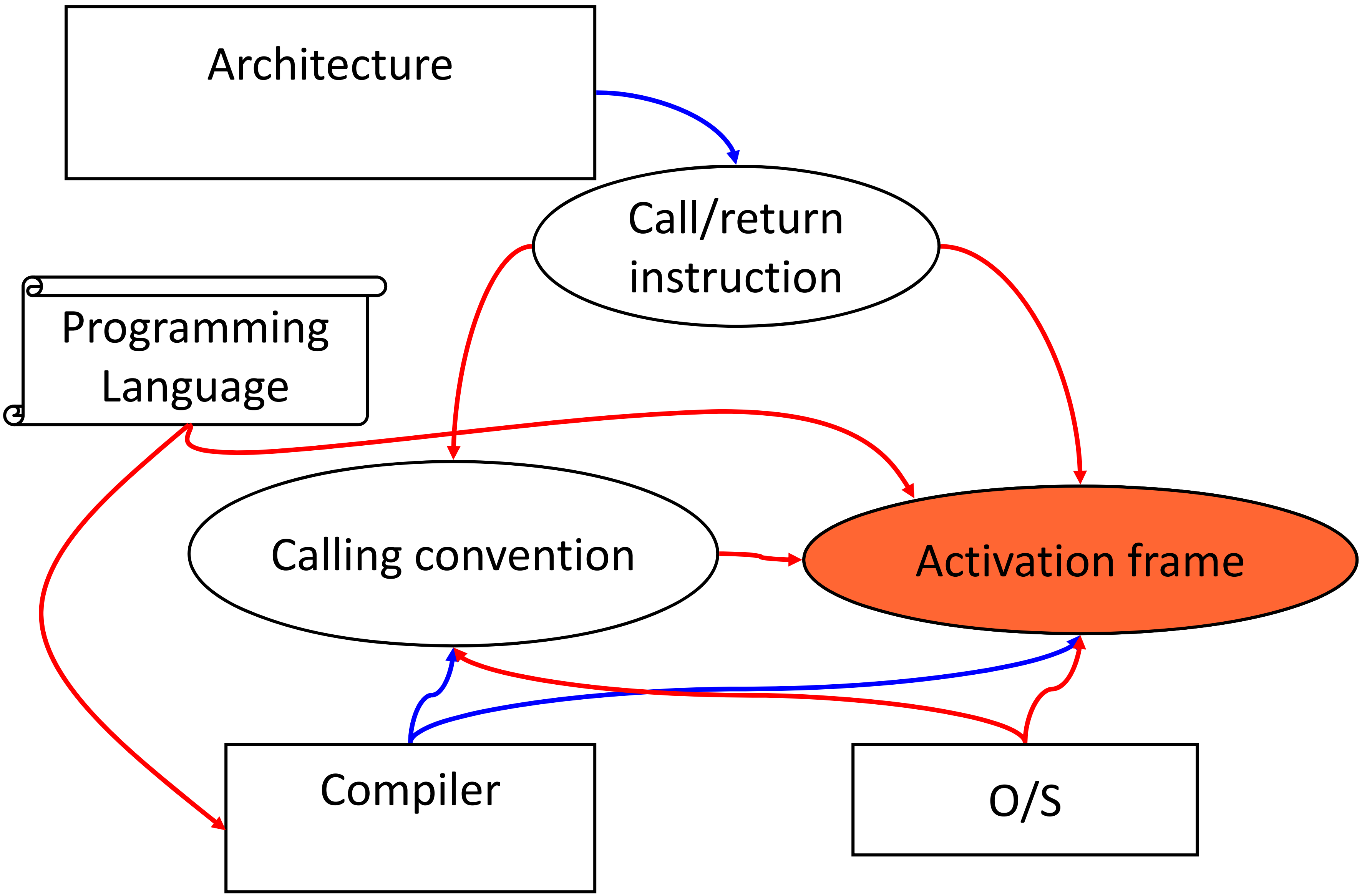
```
Bar: instr1  op1,op2
     instr2  x,y,z
     add     r3,r1,r2
     instr3
```

Abstraction supported by 3 mechanisms:

- Call instruction
- Activation Frame
- Calling Convention

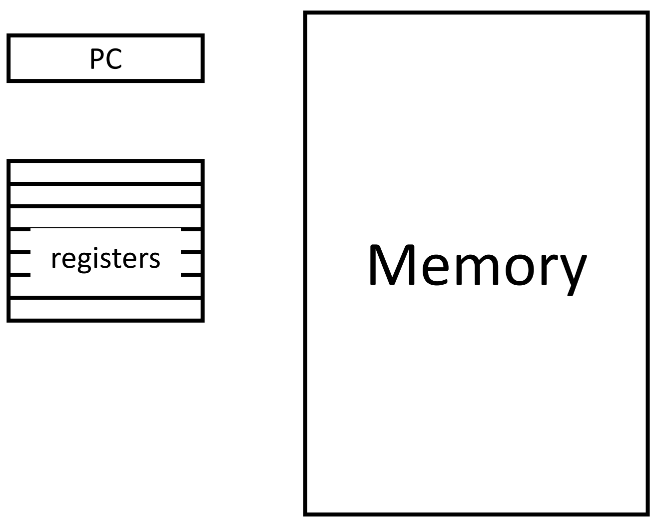
Need to resume Foo at correct place.

Implementing the Function



The Activation Frame

- Information to restore caller environment
  - Return address
  - registers
- Establishes local environment for function
  - Parameters
  - Locals
  - Temporaries
  - Dynamically allocated data?
- Support for non-locals?

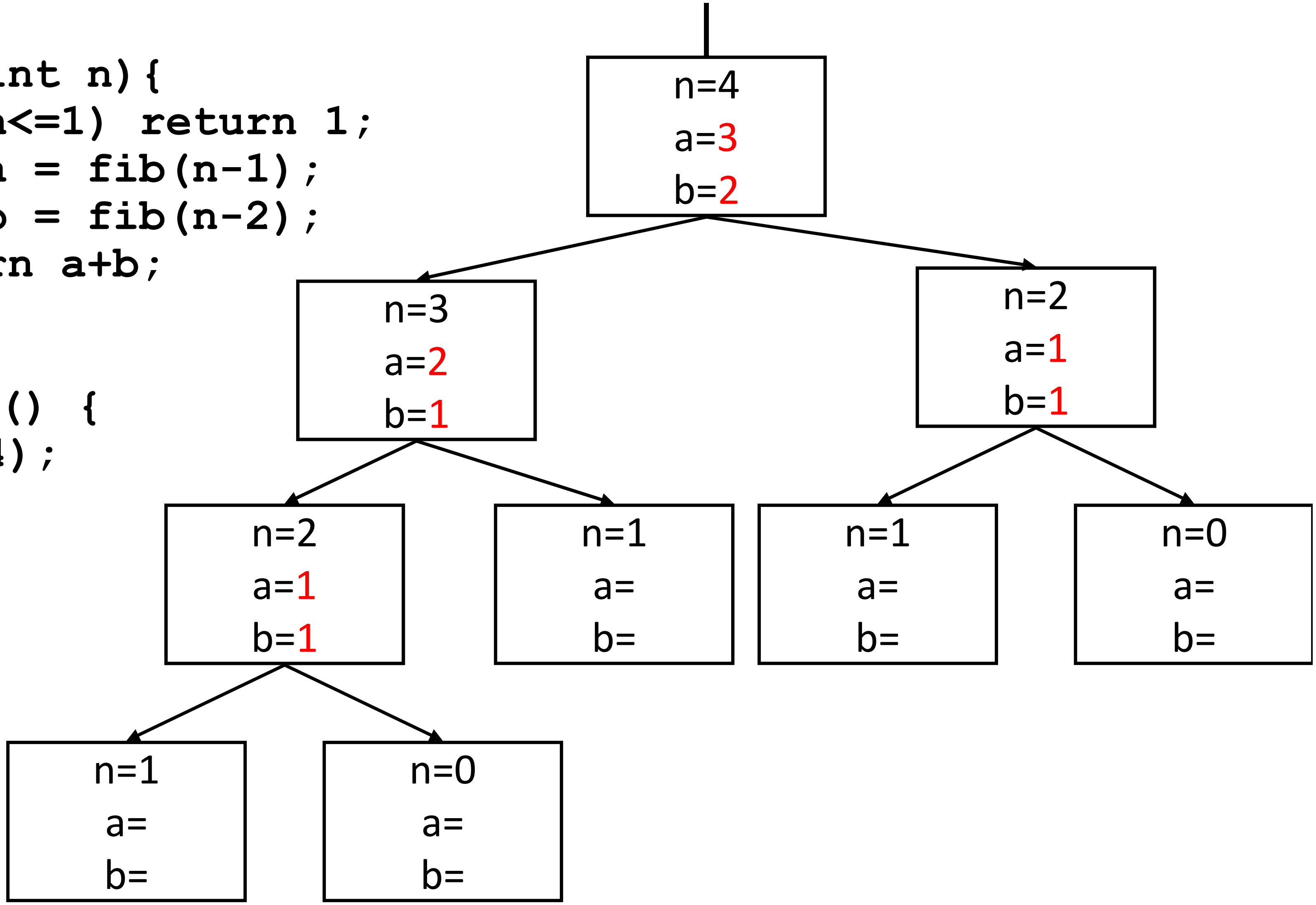


Programming Language Issues

- Can functions be recursive?
- What is Parameter passing mechanism?
  - Call-by-name
  - Call-by-value
  - Call-by-reference
- Can (and how) are non-local names referenced?
- What happens to local variables on return from function?
- Can storage be allocated locally and dynamically in a function?
- Are functions first-class objects?

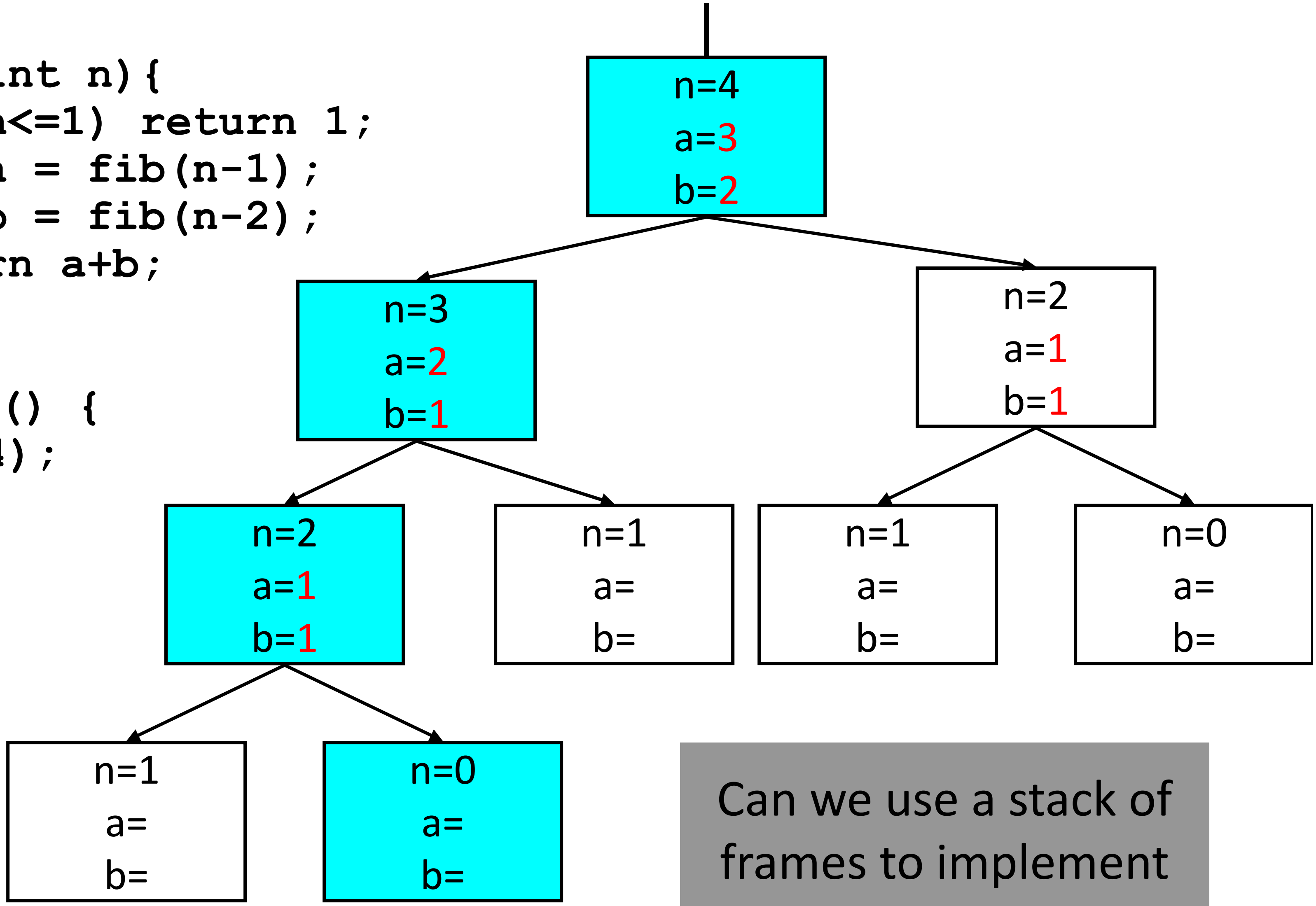
An Activation Tree

```
int fib(int n){
    if (n<=1) return 1;
    int a = fib(n-1);
    int b = fib(n-2);
    return a+b;
}

int main() {
    fib(4);
}
```



A Control Path

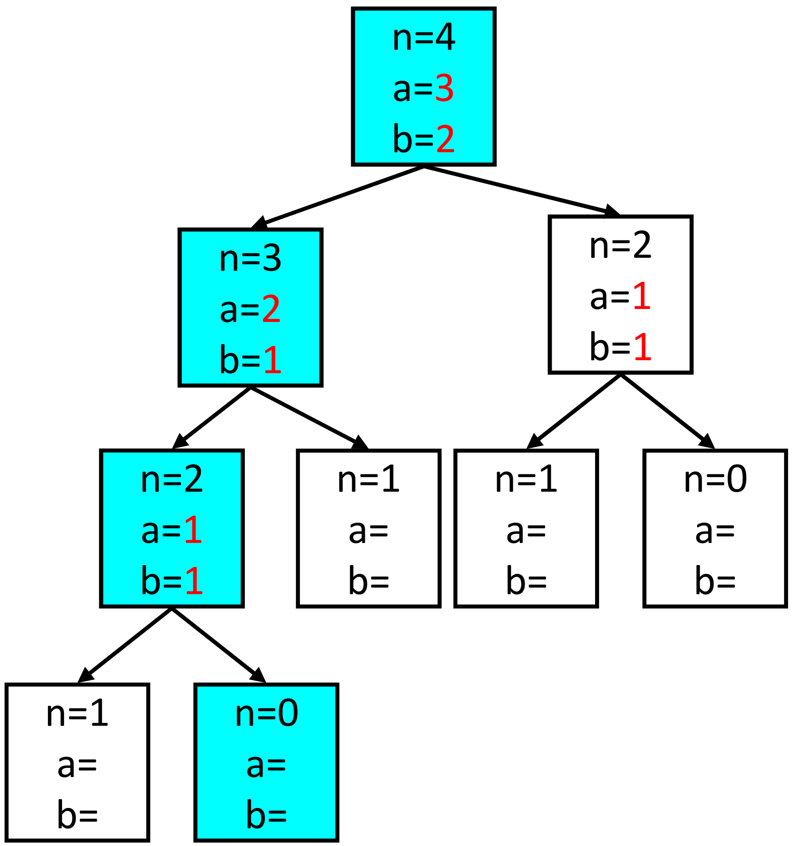
```
int fib(int n){
    if (n<=1) return 1;
    int a = fib(n-1);
    int b = fib(n-2);
    return a+b;
}

int main() {
    fib(4);
}
```



Can we use a stack of frames to implement this?

Collection of Frames

- Can functions be recursive? yes
- What is Parameter passing mechanism?
  - Call-by-name
  - Call-by-value
  - Call-by-reference
- Can (and how) are non-local names referenced?
- What happens to local variables on return from function? ?
- Can storage be allocated locally and dynamically in a function?
- Are functions first-class objects? ?



Returning references

- Can a function return a reference to a local variable?
- E.g.:

```
int* dangle() {
   int a;
   return &a;
}
```

- If so, can we use a stack of frames?

Returning Functions

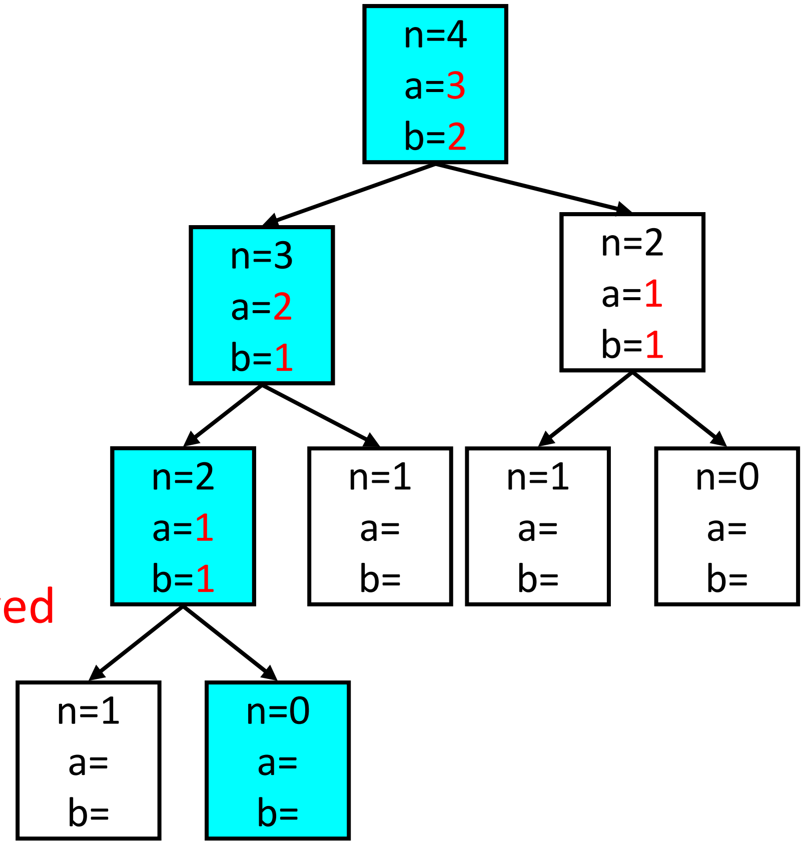
- Can a function return a function?
- E.g.:

```
typedef int (*p2f)(int);
p2f hof(void) {
   int add5(int b) {
      return 5+b;
   }
   return &add5;
}
```

- Can we use a stack of frames?

Collection of Frames

- Can functions be recursive? yes
- What is Parameter passing mechanism?
  - Call-by-name
  - Call-by-value
  - Call-by-reference
- Can (and how) are non-local names referenced? ?
- What happens to local variables on return from function? destroyed
- Can storage be allocated locally and dynamically in a function?
- Are functions first-class objects? ?



Non-local Access

- Can a function refer to variables in outer functions?
- E.g.:

```
int add2(int a, int c) {
   int add1(int b) {
      return a+b;
   }
   return add1(c);
}
```

- Stack of Frames ok?
- There are other issues however (deal with later)

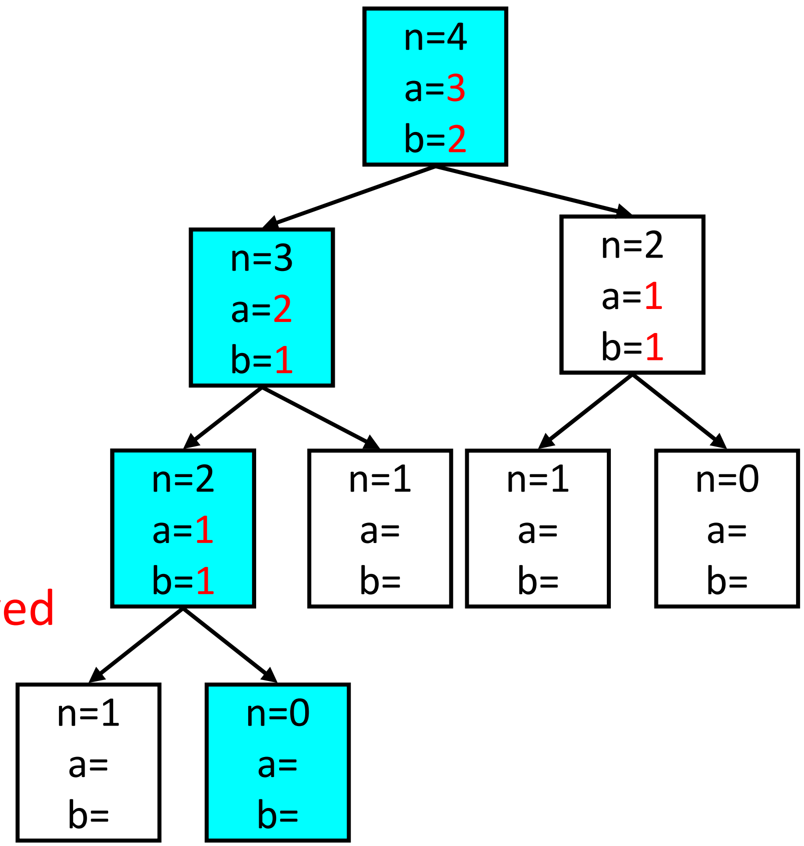
Non-local Access vs. Global Access

```
int add2(int a, int c) {
   int add1(int b) {
      return a+b;
   }
   return add1(c);
}
```

```
int a;
int add2(int c) {
   int add1(int b) {
      return a+b;
   }
   return add1(c);
}
```

Collection of Frames

- Can functions be recursive? yes
- What is Parameter passing mechanism?
  - Call-by-name
  - Call-by-value
  - Call-by-reference
- Can (and how) are non-local names referenced? no
- What happens to local variables on return from function? destroyed
- Can storage be allocated locally and dynamically in a function?
- Are functions first-class objects? ?



1st Class Functions&Non-local Access

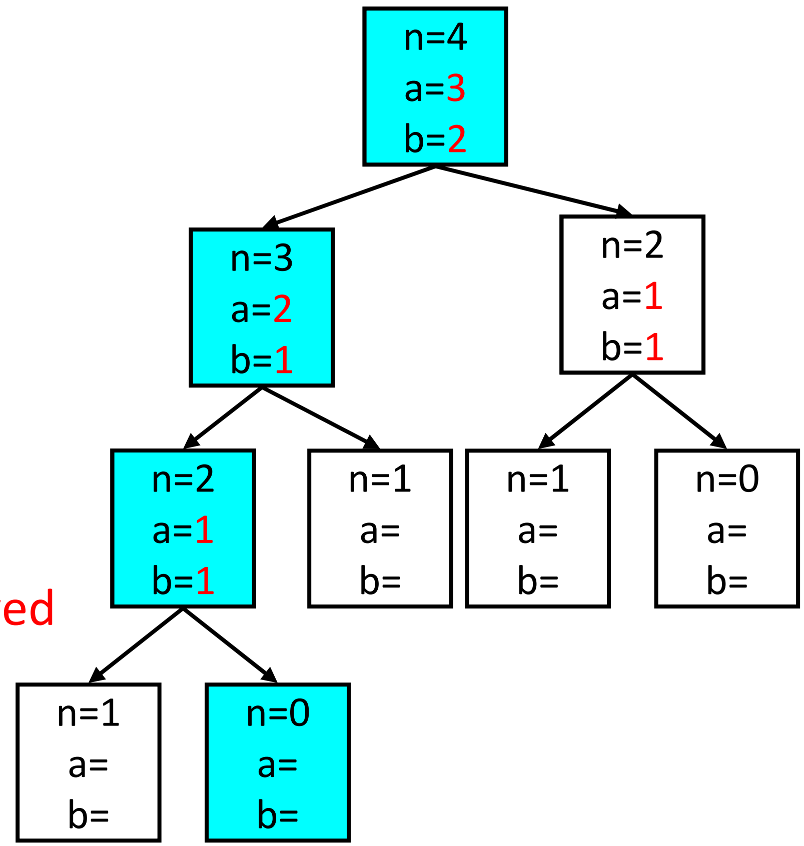
- Can a function return a function?
- E.g.:

```
typedef int (*p2f)(int);
p2f hof(int a) {
   int adda(int b) {
      return a+b;
   }
   return &adda;
}
```

- What is going on here?
- Combination of
  - non-local access &
  - first-class functions.

Collection of Frames

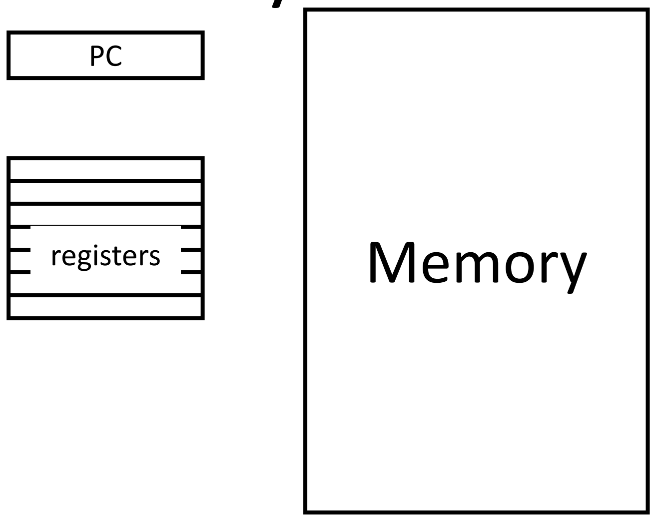
- Can functions be recursive? yes
- What is Parameter passing mechanism?
  - Call-by-name
  - Call-by-value
  - Call-by-reference
- Can (and how) are non-local names referenced? yes
- What happens to local variables on return from function? destroyed
- Can storage be allocated locally and dynamically in a function?
- Are functions first-class objects? no



Use a stack of activation frames.

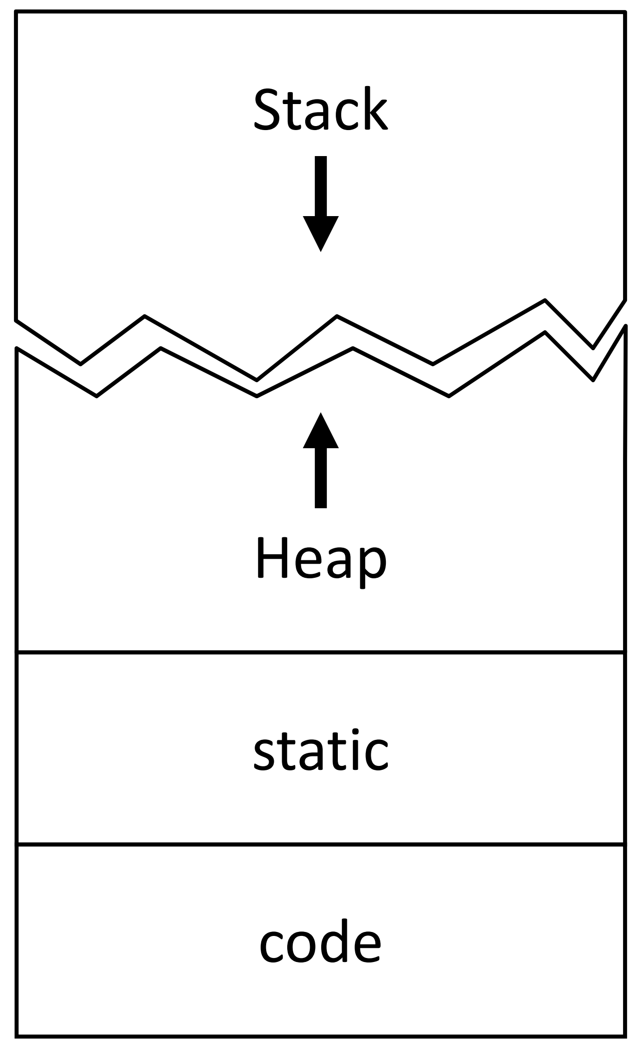
Memory Layout

- We went through this analysis to determine the interaction of frames
- We are assuming:
  - stack is good for storing frames
  - Allows “unlimited” recursion
- How does this interact with entire system?



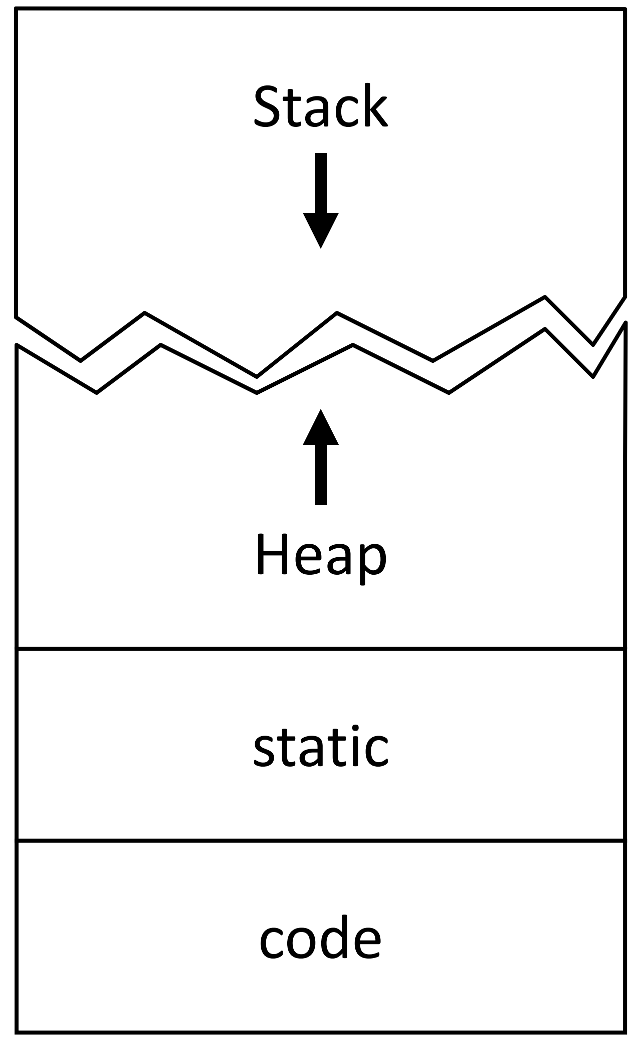
Memory Organization

- Instructions are (usually) static and go into code.
- Static data is allocated at compile time, resolved at link-time
- Stack grows down and holds activation frames
- Heap grows up

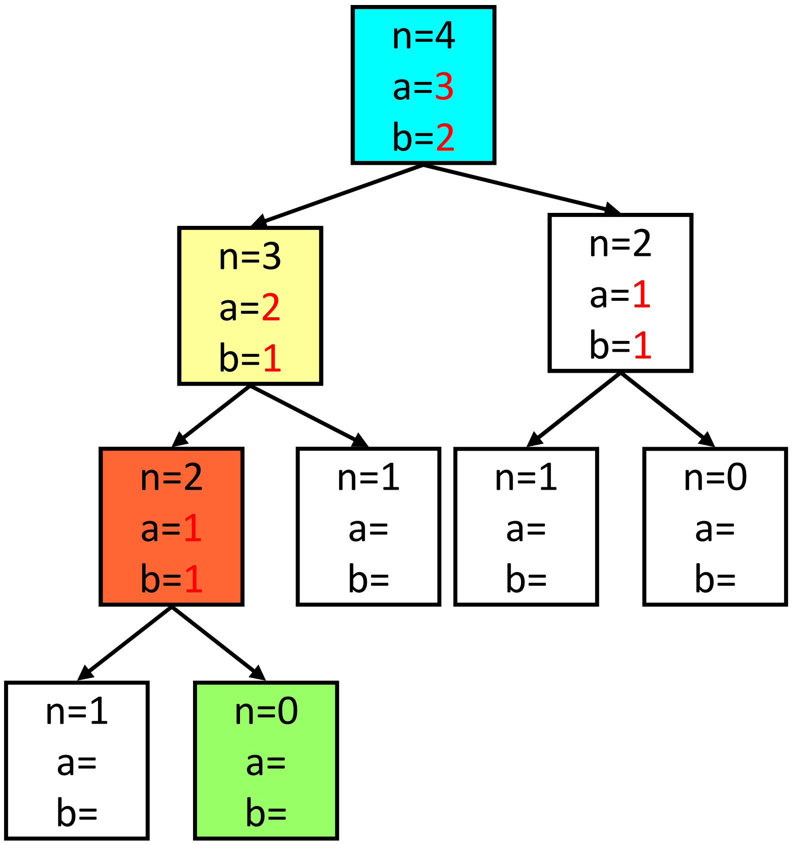


Memory Organization

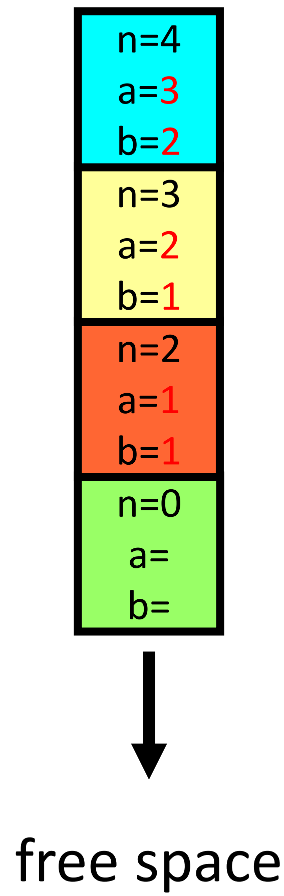
- Code and static contain fixed size statically allocated information
- Stack and data contain dynamically sized and dynamically allocated information
- Stack and heap compete for memory.
- Relates to storage classes



The Stack

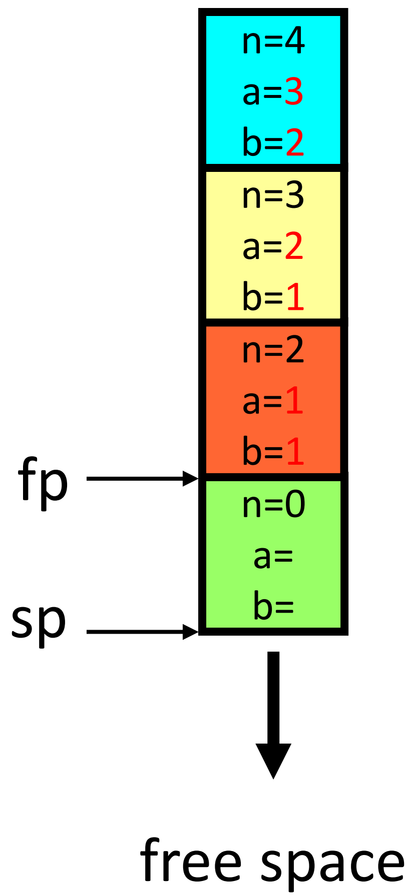


n=4
a=3
b=2
n=3
a=2
b=1
n=2
a=1
b=1
n=2
a=1
b=1
n=1
a=
b=
n=1
a=
b=
n=0
a=
b=
n=1
a=
b=
n=0
a=
b=



n=4
a=3
b=2
n=3
a=2
b=1
n=2
a=1
b=1
n=0
a=
b=
free space

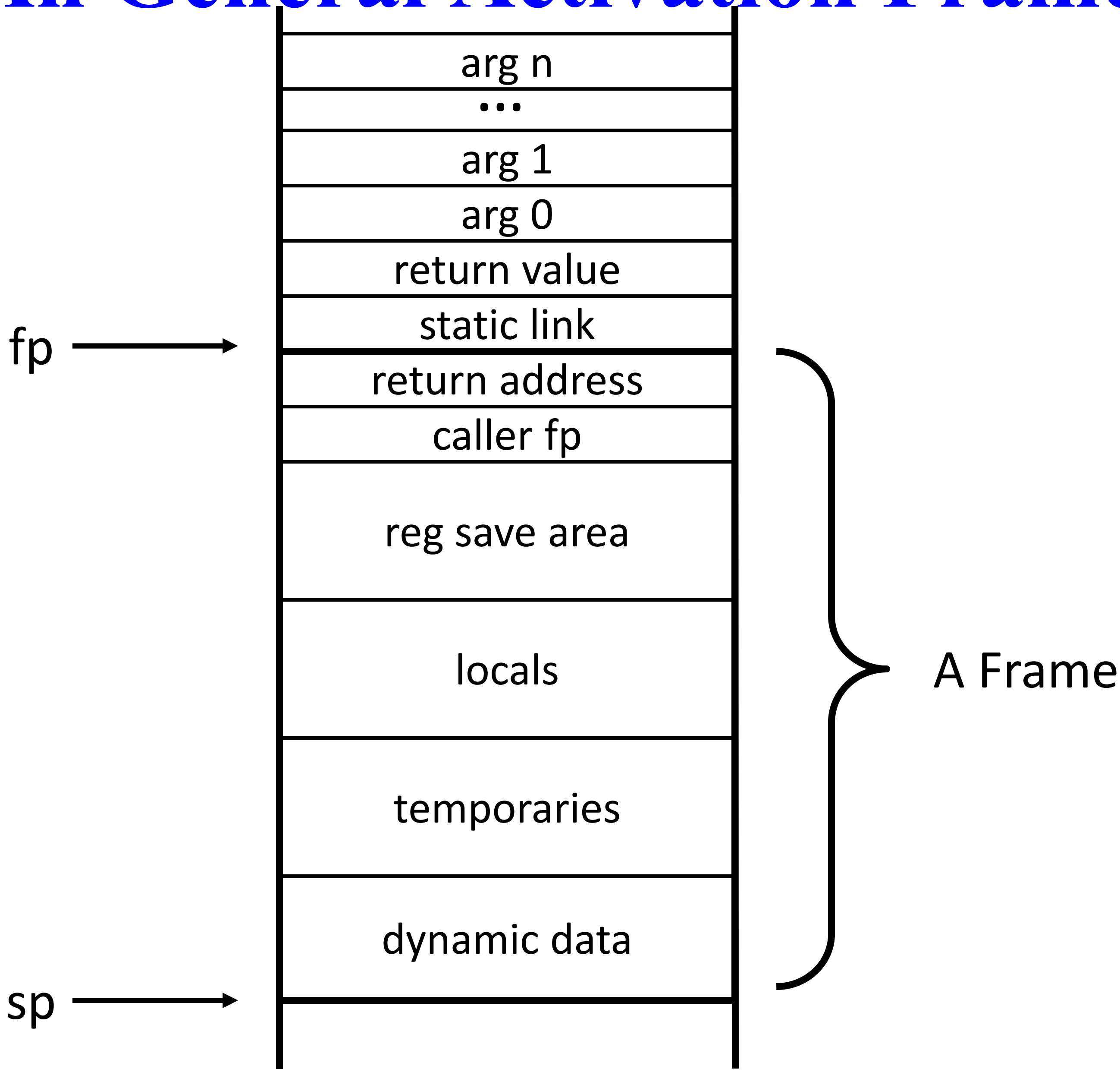
The Stack



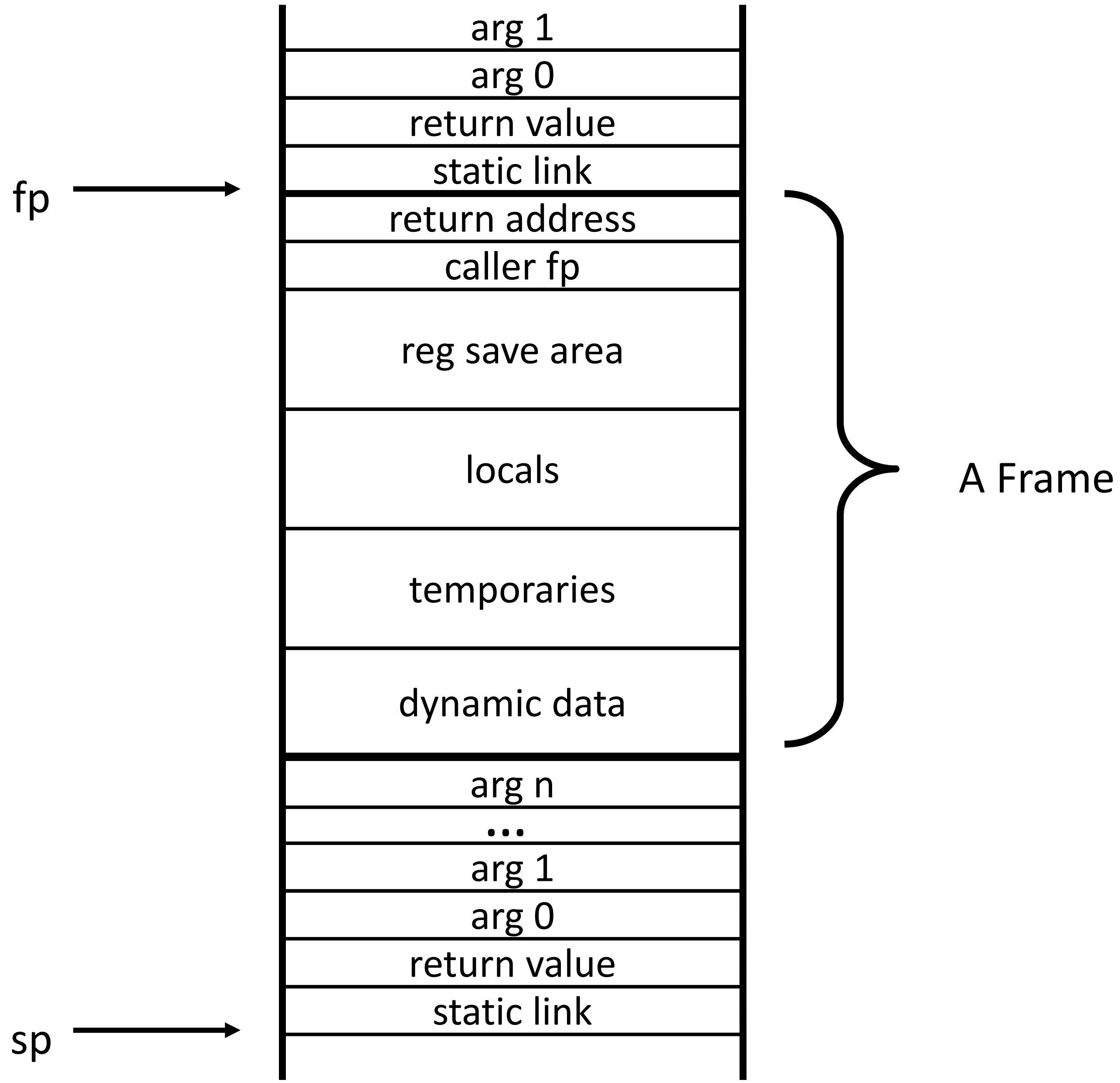
- How do we know where one frame starts and another stops?
- How do we track return address?
- How do we access local variables?
- How do we access non-local variables?
  - Frame Pointer (fp)
  - Stack Pointer (sp)

Why not just say **`%rbp`**?

An General Activation Frame



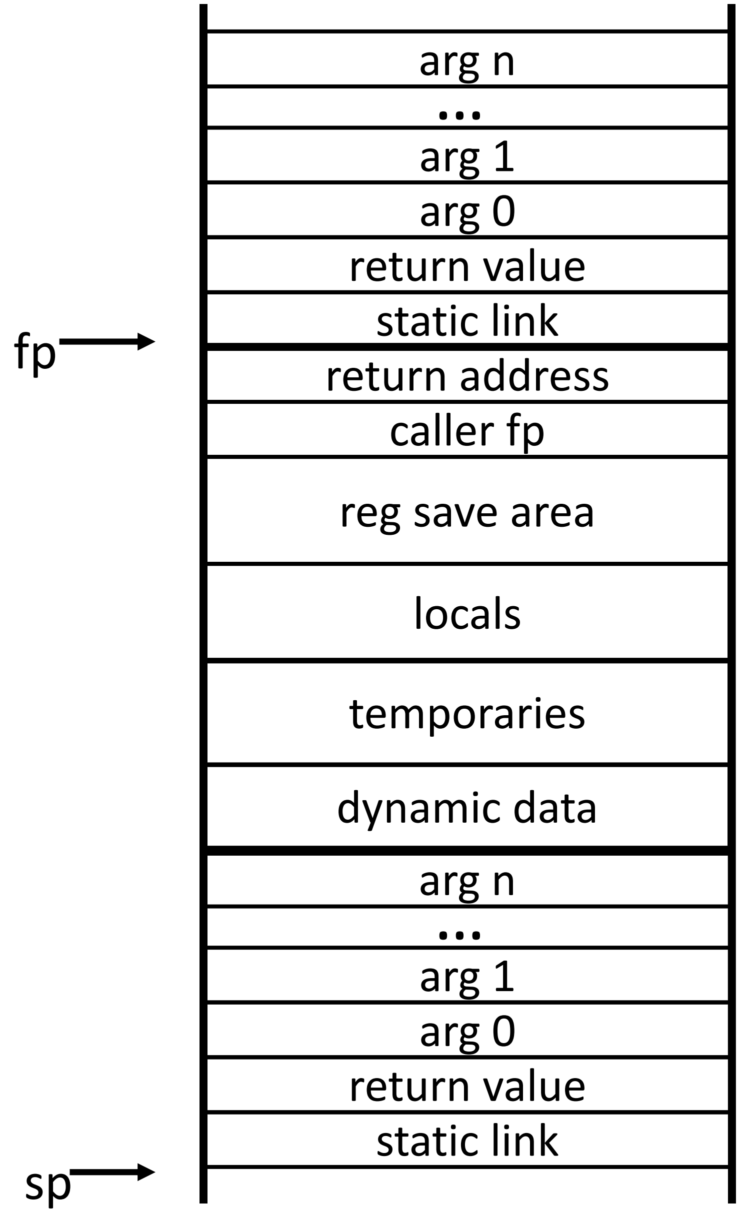
Right Before Next Call



Who does what?

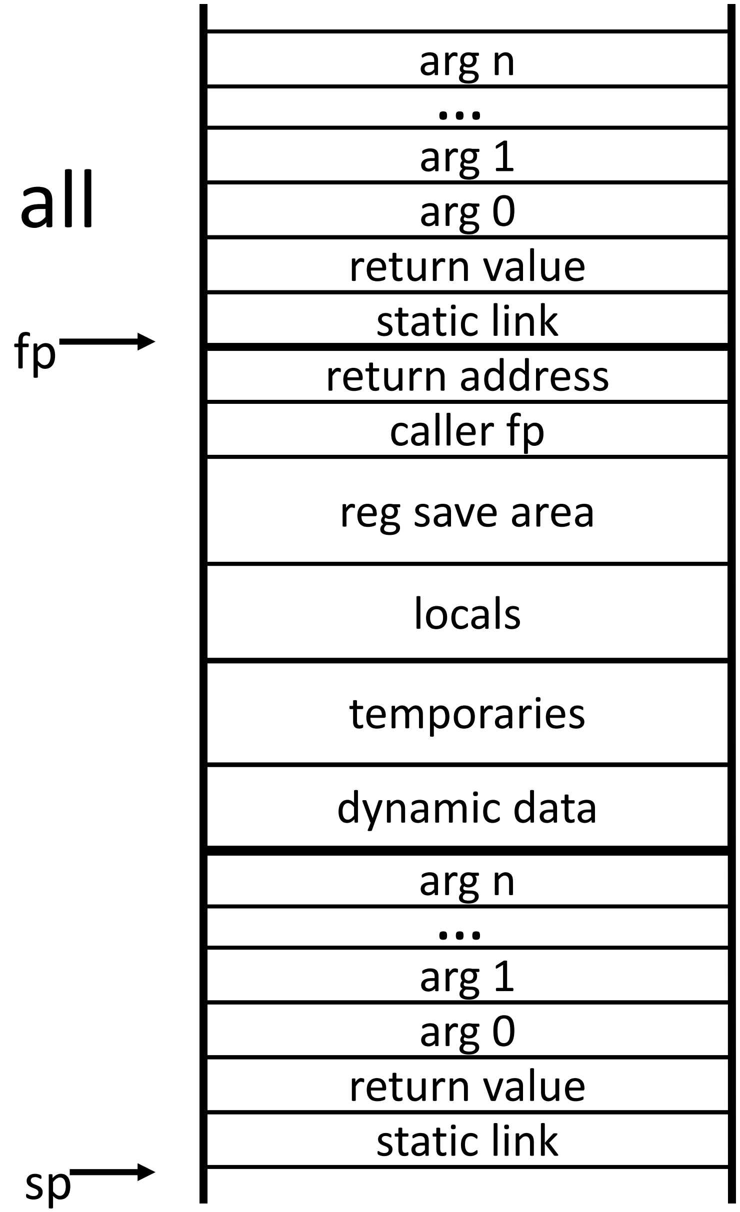
```
Foo: Prolog
     instr1    op1,op2
     instr2    x,y,z
     mov       z,a
     add          r3,r1,r2
     setup for call
     call bar(a,b,c,d)
     recover from call
     instr1    op1,op2
     instr2    x,y,z
     mov       z,a
     add          r3,r1,r2
     Epilog
```

The answer is: it depends!



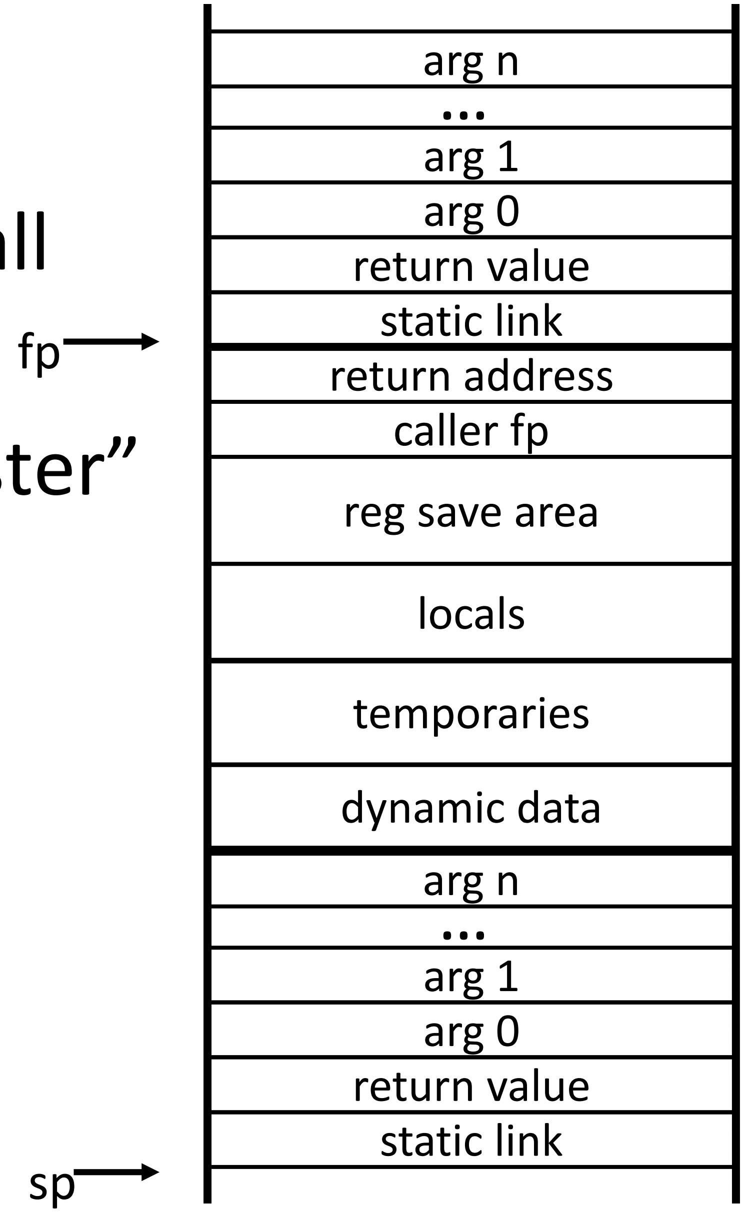
Frame Pointer

- Used as base for accessing all elements of frame.
- In Prolog:
  - [sp+x] = fp; save caller's fp
  - fp = sp
  - sp -= frameSize
- In Epilog
  - sp = fp
  - fp = [sp+x]
- Do we always need fp?



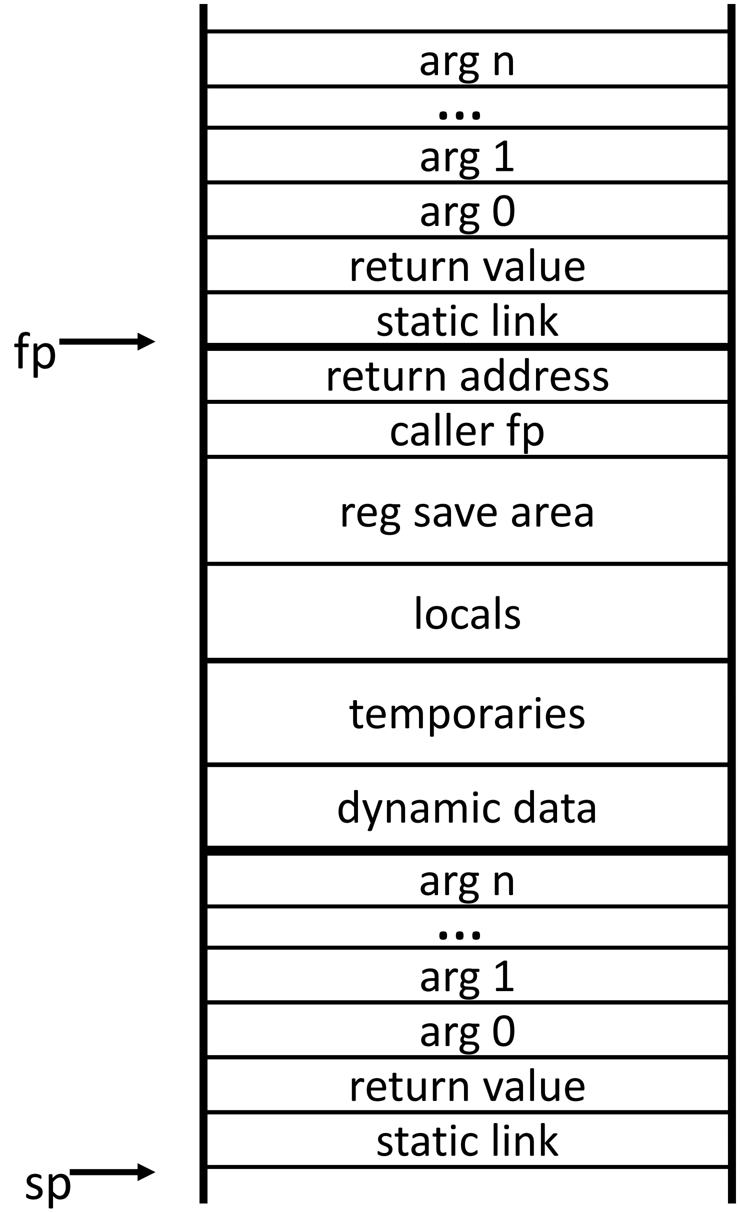
Frame Pointer

- Used as base for accessing all elements of frame.
- Many times a "fictional register"
- On Call
  - sp -= frameSize
  - fp = sp + frameSize
- On Return
  - sp += frameSize



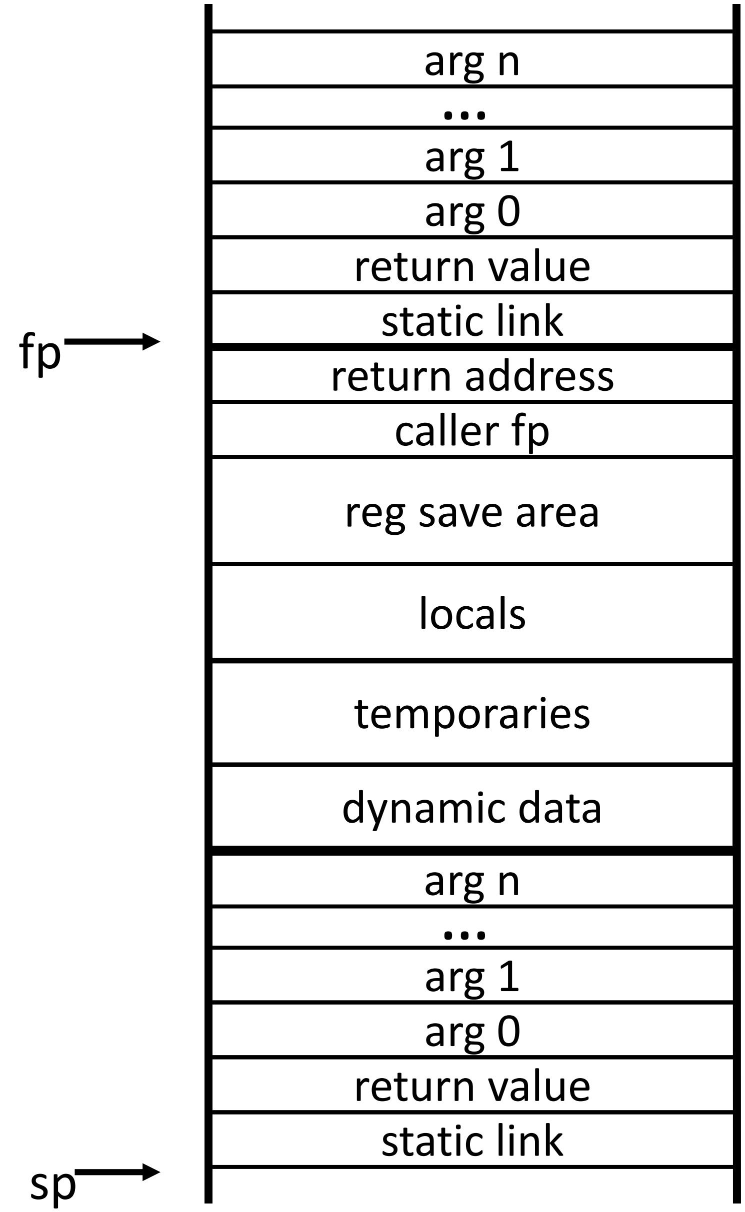
Parameter Passing

- Caller puts parameters into stack starting at current sp
- Save space for return value
- Invoke Callee
- Actually we can do better!
  - Caller reserves space for first k params and return value
  - Why is this better?
  - Why bother to reserve space at all?



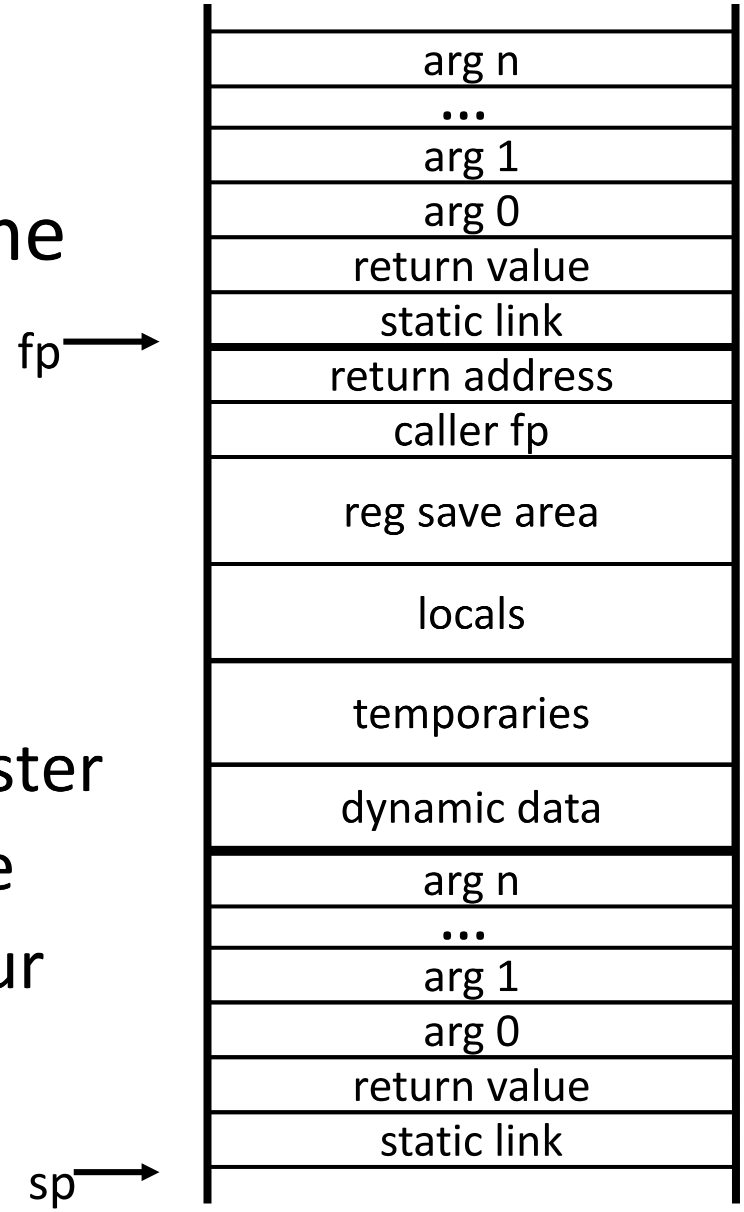
Registers

- One set of registers
- Callee might want to use same register as caller
- Caller can save all registers
- Callee can save all registers
- Which is better?
- Issues:



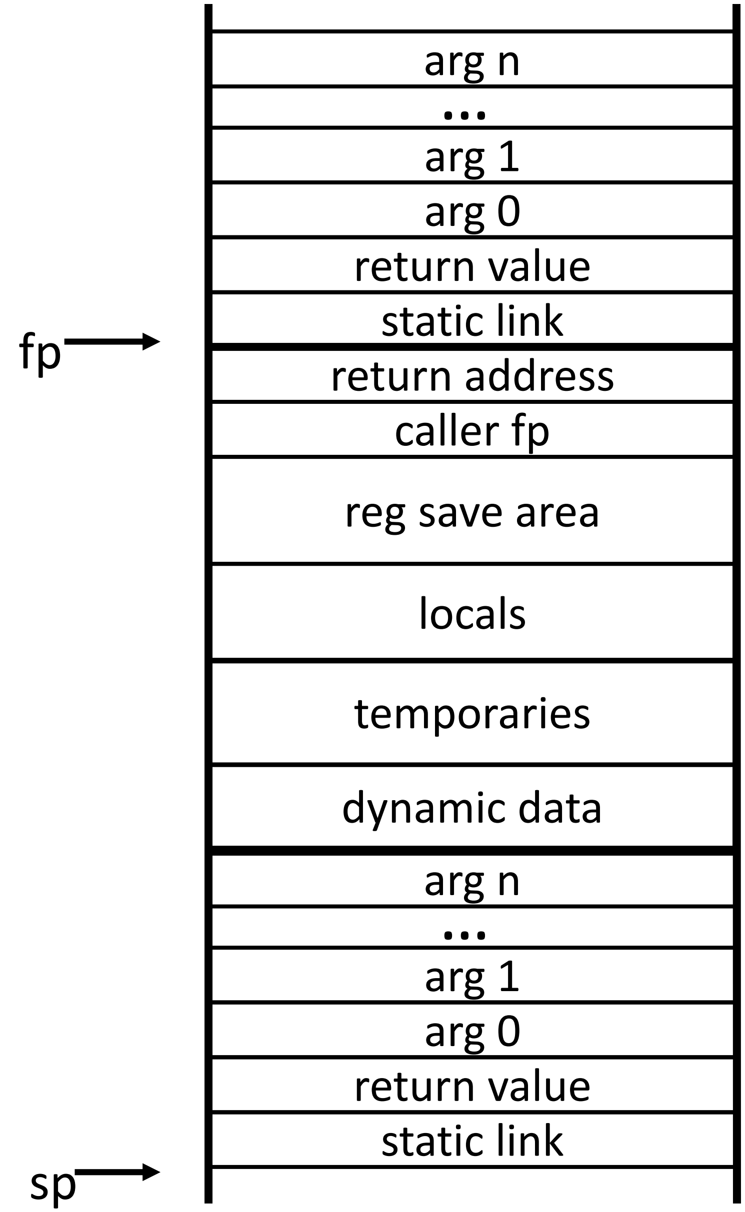
Registers

- One set of registers
- Callee might want to use same register as caller
- Caller can save all registers
- Callee can save all registers
- Issues:
  - callee might not use your register
  - Extreme case is leaf procedure
  - caller might not have used your register



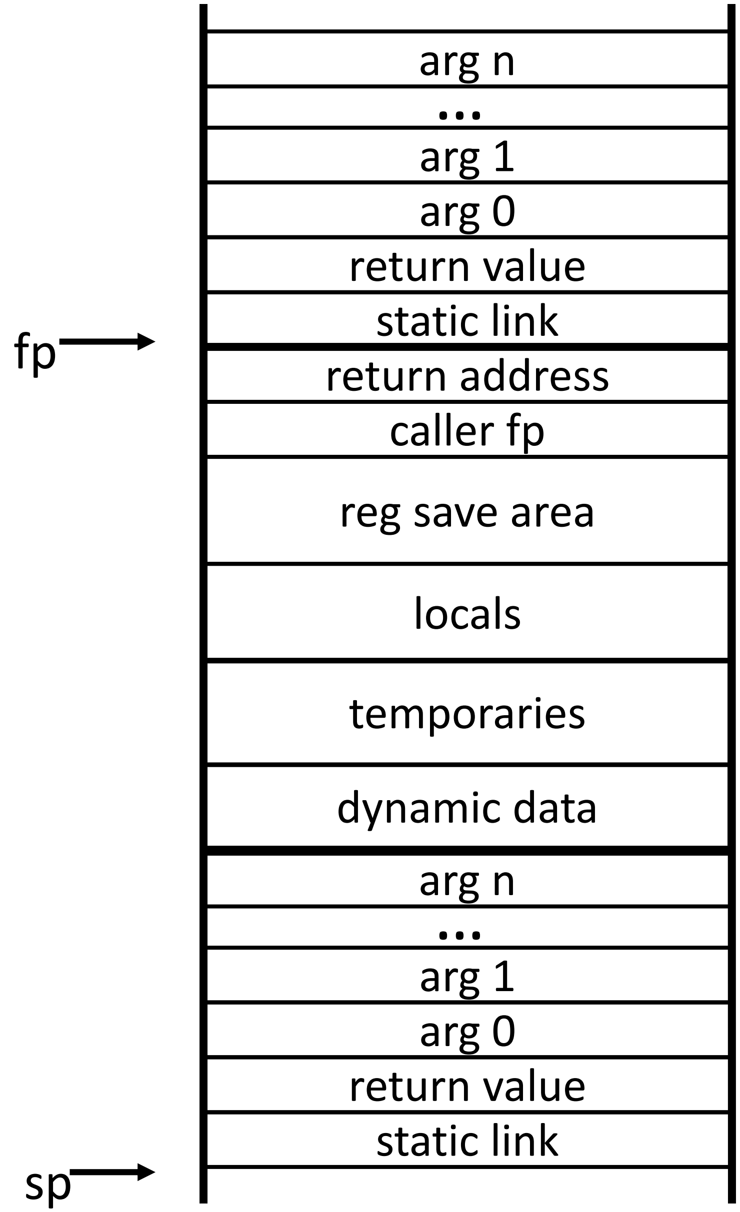
Registers

- One set of registers
- Callee might want to use same register as caller
- Caller can save all registers
- Callee can save all registers
- Make some registers
  - caller save
  - callee save
- Or, register windows?



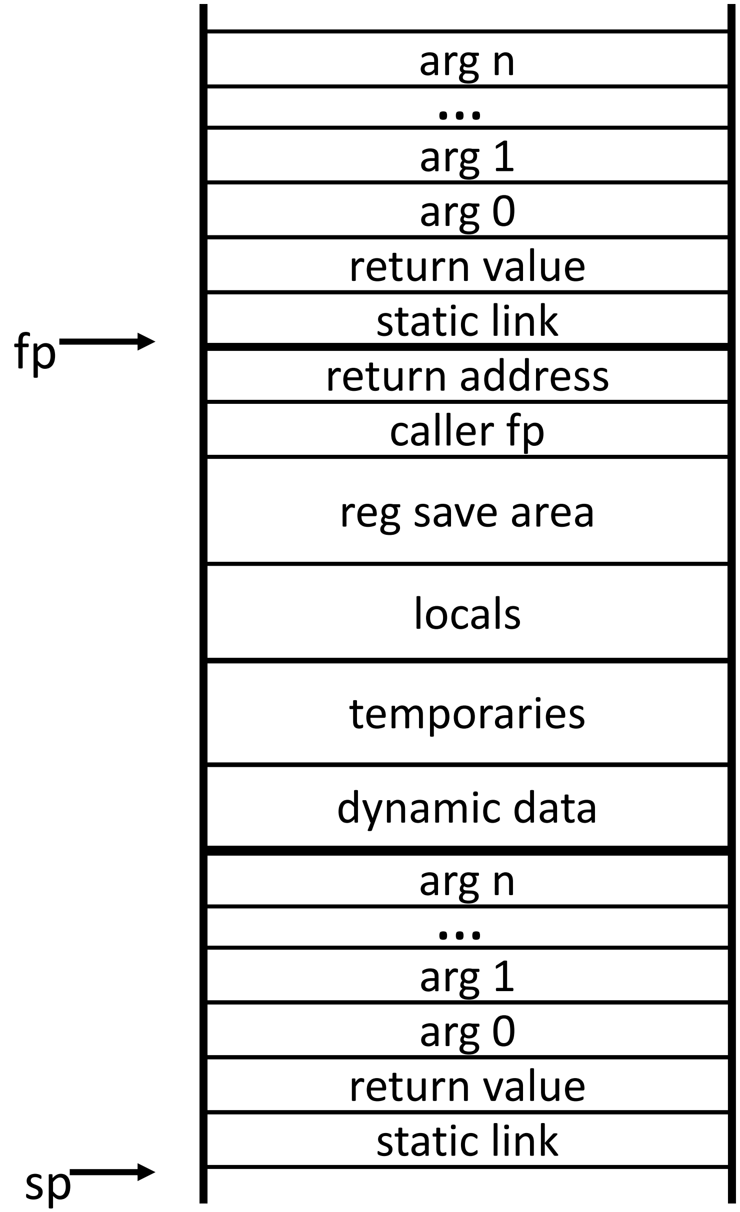
Return Address

- Who should save it?
- Should it be saved?



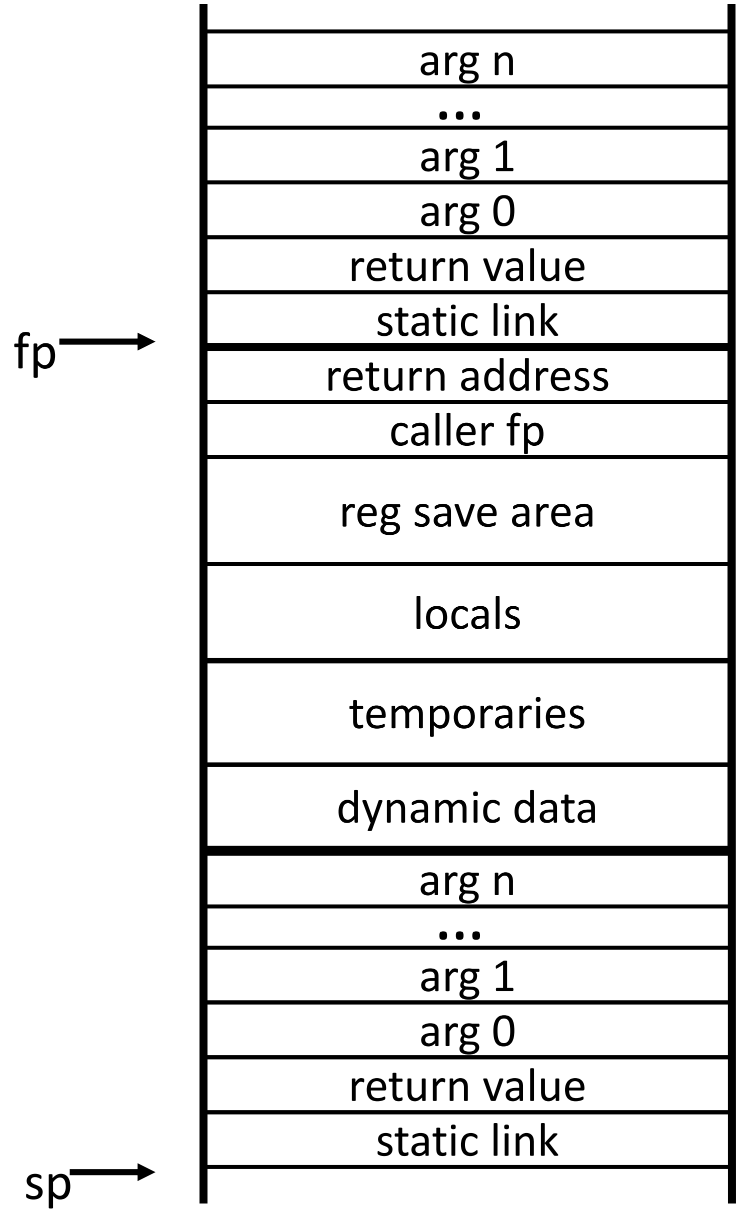
Locals/Temps/Dynamic

- Allocated by callee
- Dynamic data requires fp and sp



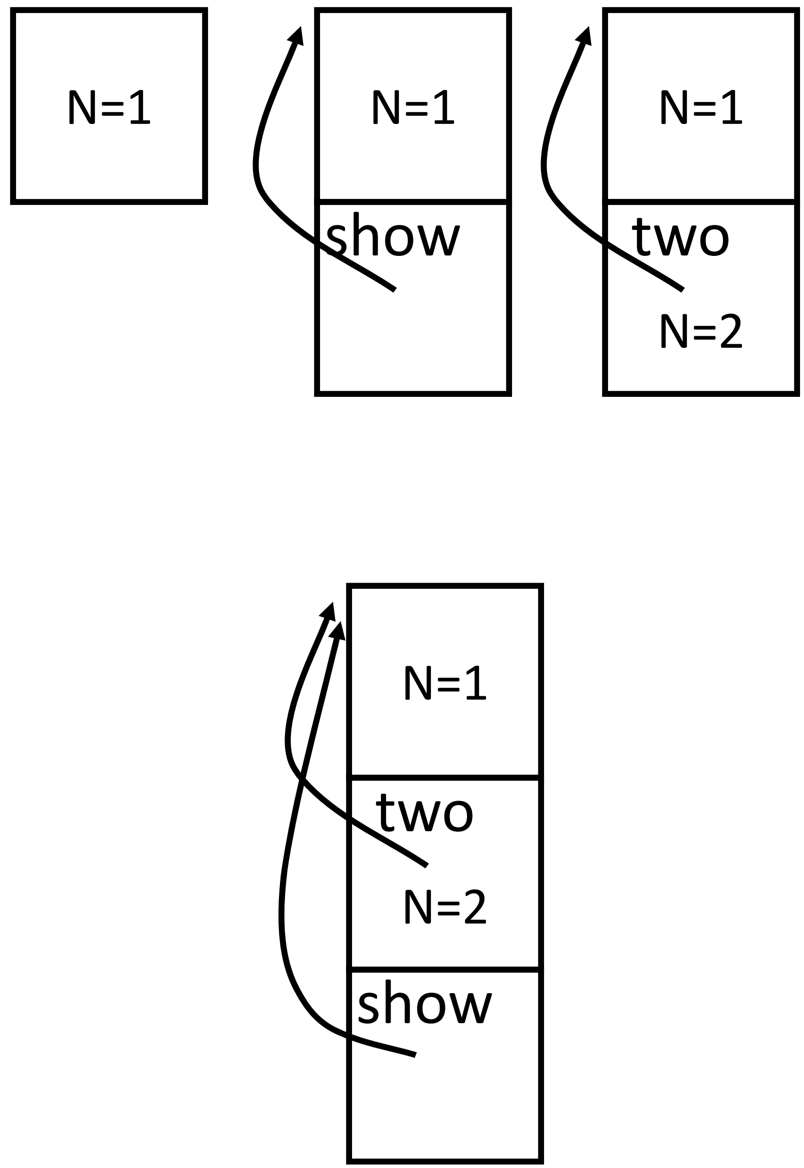
The Static Link

- Static link used to access non-local variables for nested procedures.



Nested Functions*

```
void outer(void) {
    int N = 1;
    void show(void) {
      print(N);
      print(" ");
    }
    void two(void) {
      int N = 2;
      show();
    }
    show(); two();
    show(); two())
}
```



*Lexically scoped

Implementing Nested Functions

- Non-local names are referenced by their level and offset.
  - level is lexical nesting depth
  - offset is offset into activation frame
- During compilation names must be translated into <level,offset> pair.
  - Use block structured symbol tables
  - Track difference between current function's nesting depth and referenced names nesting depth
- At runtime, either
  - static links
  - displays

Static Links

- Keep a link list which follows the lexical nesting depth (NOT THE SAME AS PARENT FP!)
- Can follow chain to find frame at level k
- On call/return setup and teardown chain
- Caller passes pointer to lexically enclosing frame of callee.
- Maintenance cost: store (on call)
- Access cost from frame at level l to one at level k: (k-l) extra loads

Display

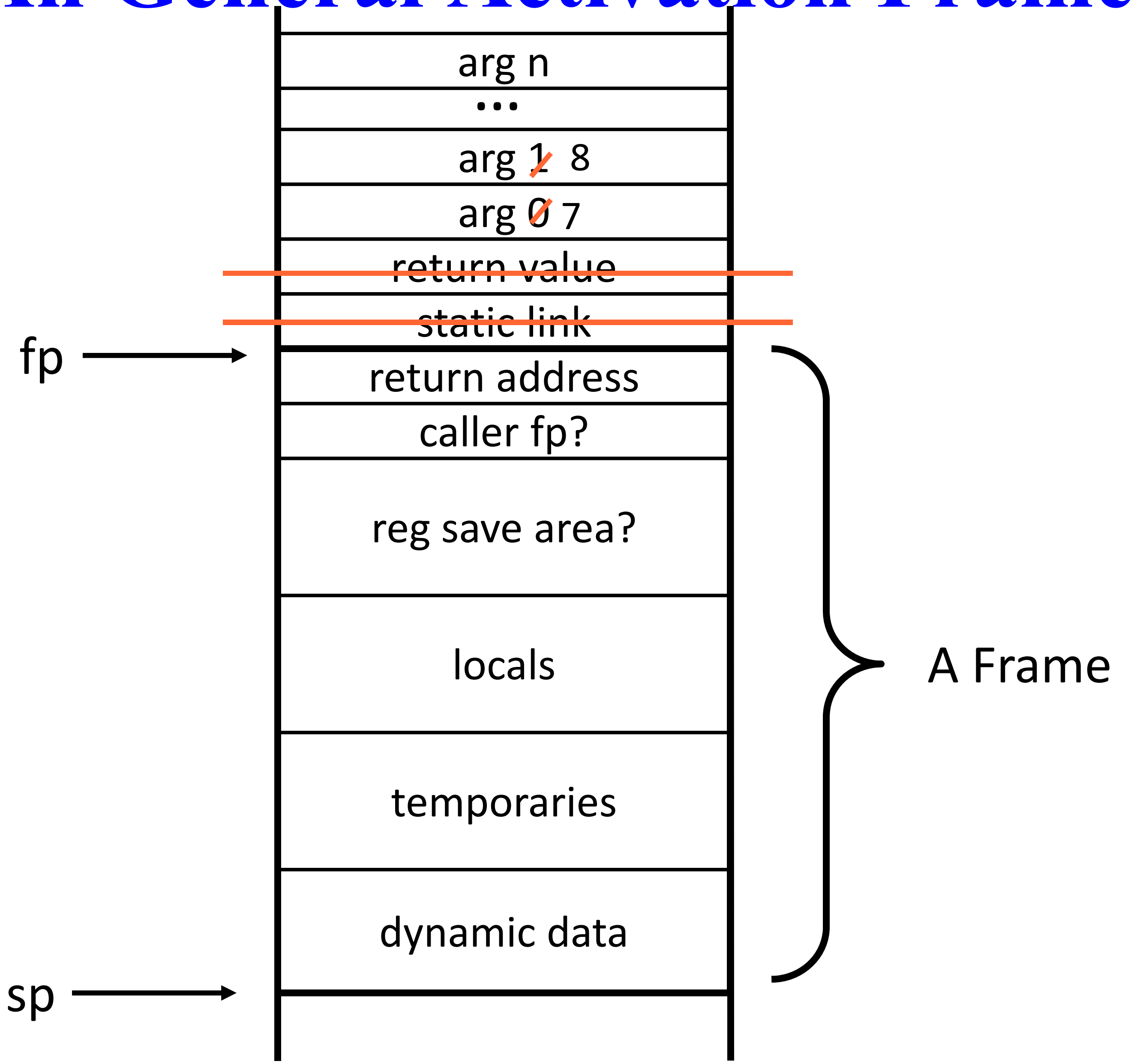
- Maintain global table with size = maximum lexical nesting in program
- In prolog:
  - save k^{th} entry in display for call to function at level k.
  - Store FP in k^{th} entry in display
- In epilog:
  - restore display
- On access: one load from display to get proper frame.

Activation Frame C0

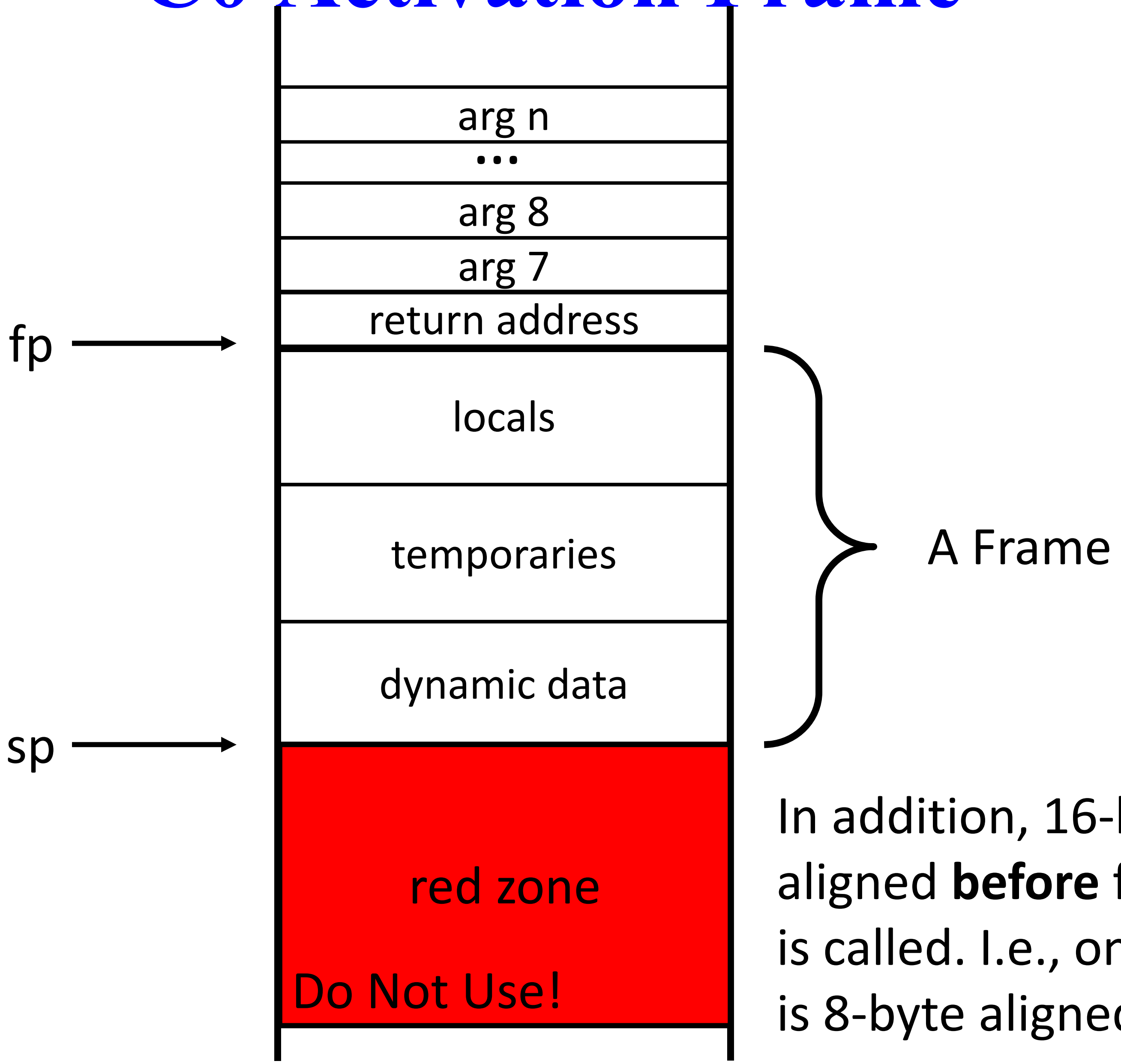
- No nested functions, so no static link
- return value is not stored on stack:
  `%rax`
- First 6 arguments stored in registers:
  `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
- Divides registers into caller save:
  `%r10, %r11`
- And, callee save:
  `%rbx, %rbp, %r12, %r13, %r14, %r15`

This is a part of C's calling convention

An General Activation Frame

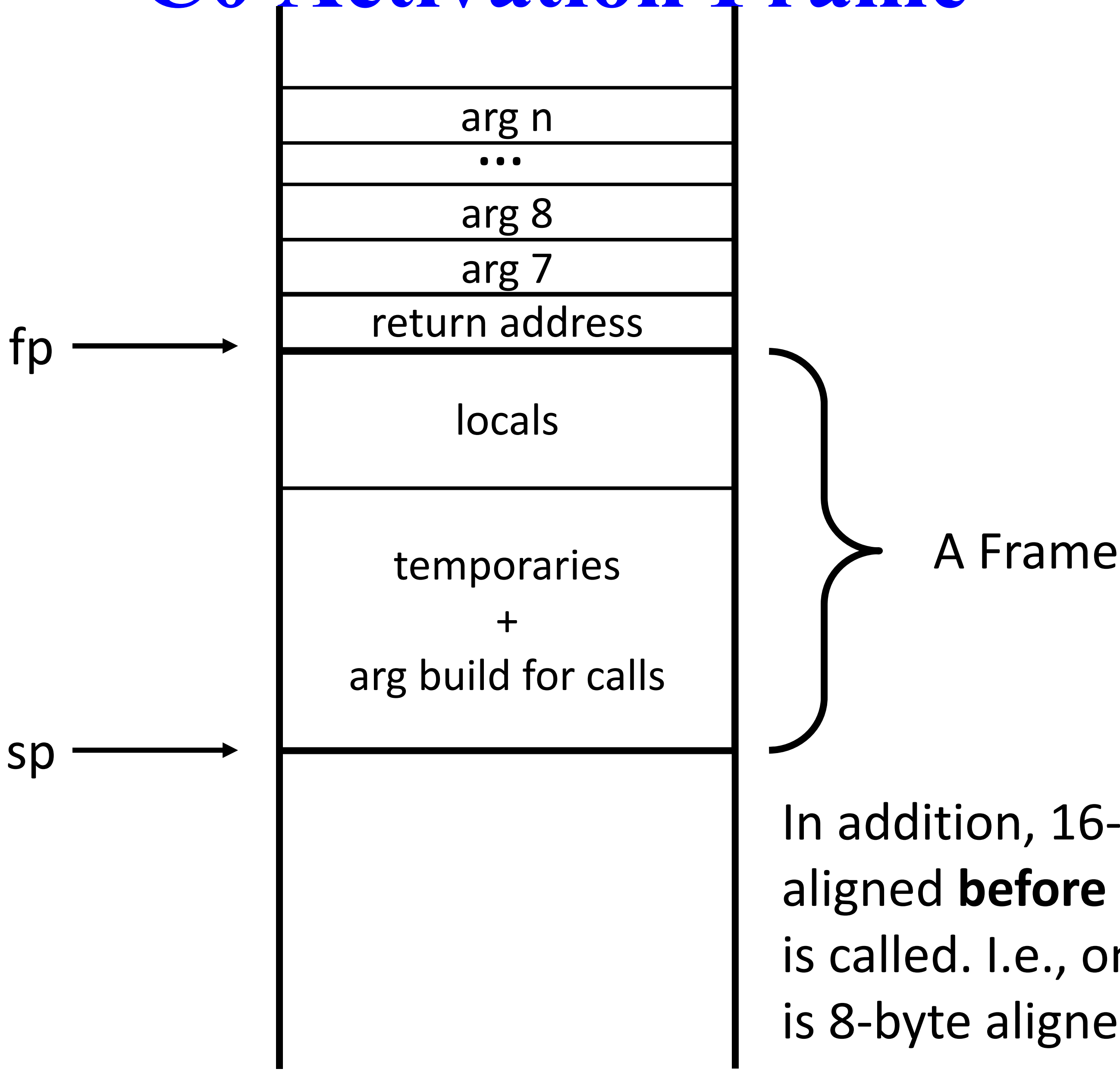


C0 Activation Frame



In addition, 16-byte aligned **before** function is called. I.e., on entry it is 8-byte aligned.

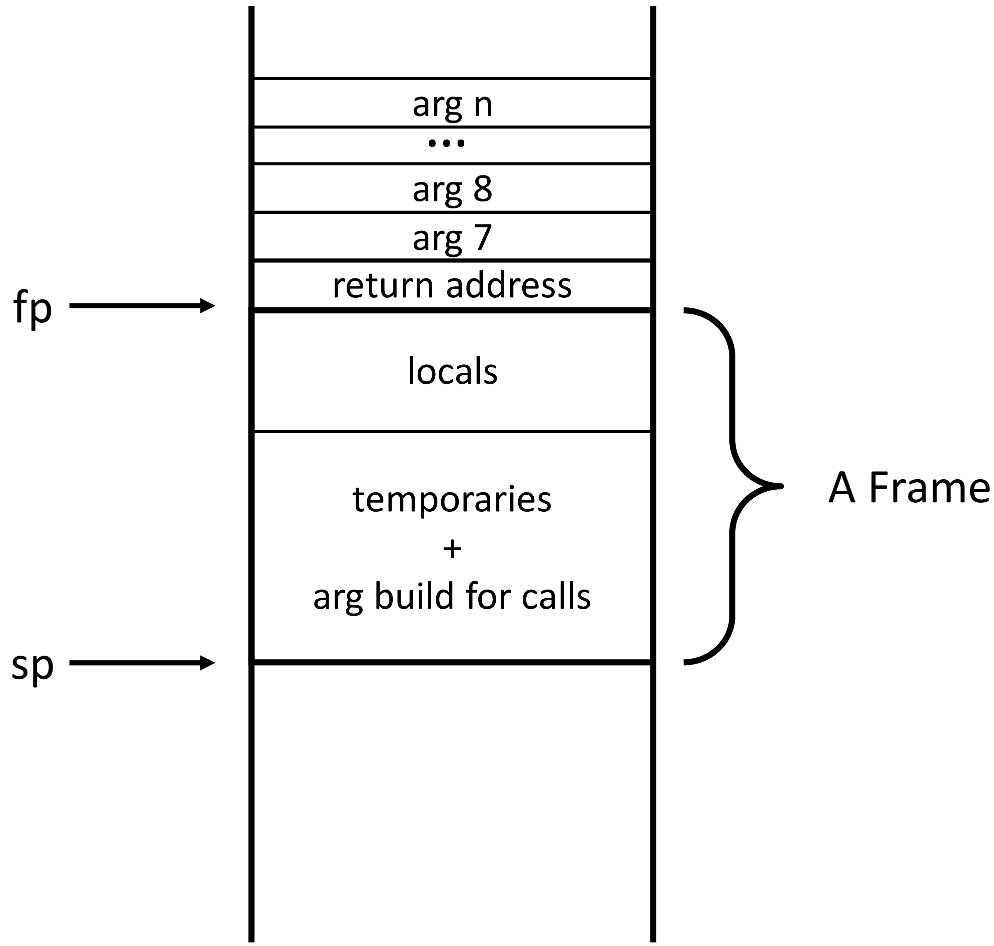
C0 Activation Frame



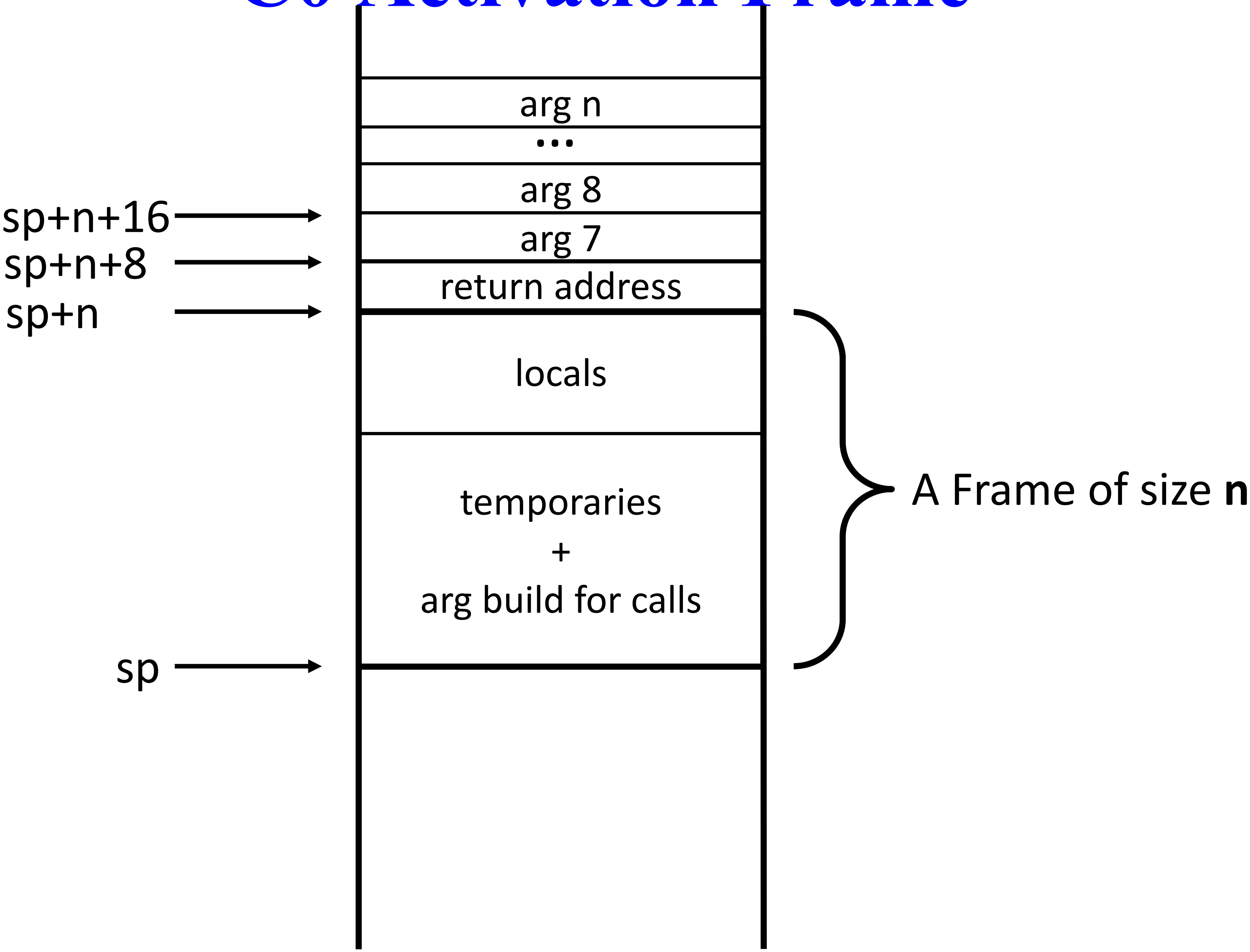
In addition, 16-byte aligned **before** function is called. I.e., on entry it is 8-byte aligned.

to fp or not to fp?

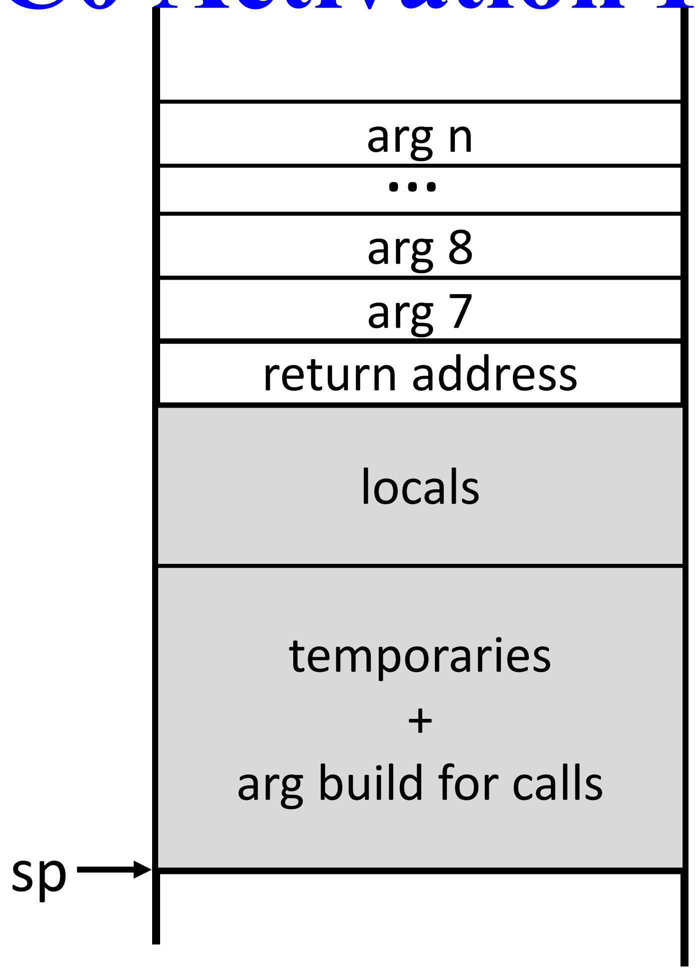
- Do we need a frame pointer?



C0 Activation Frame

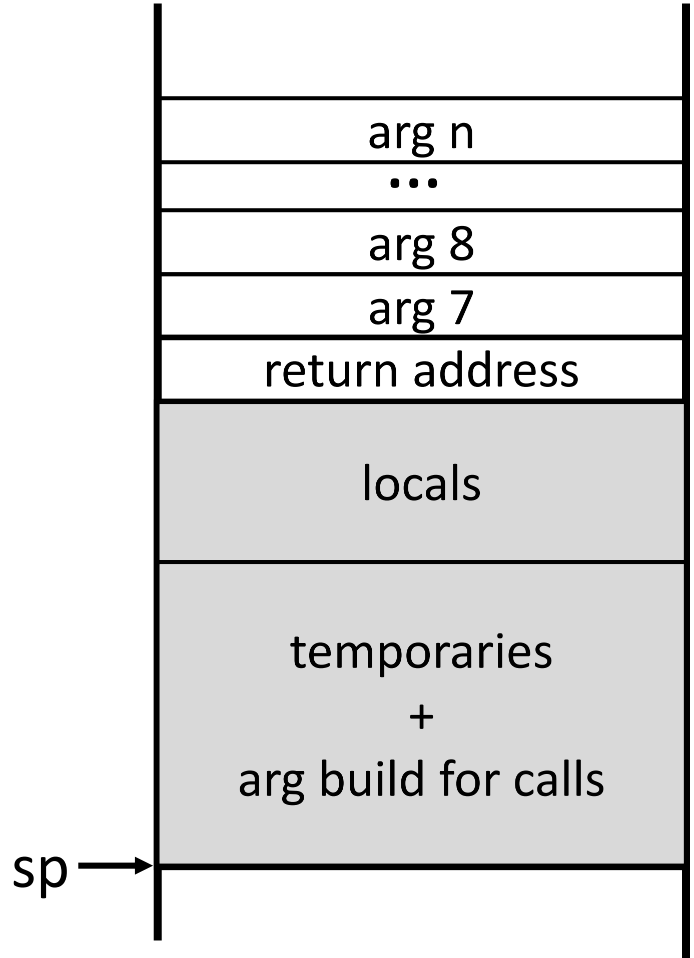


C0 Activation Frame



Who does what?

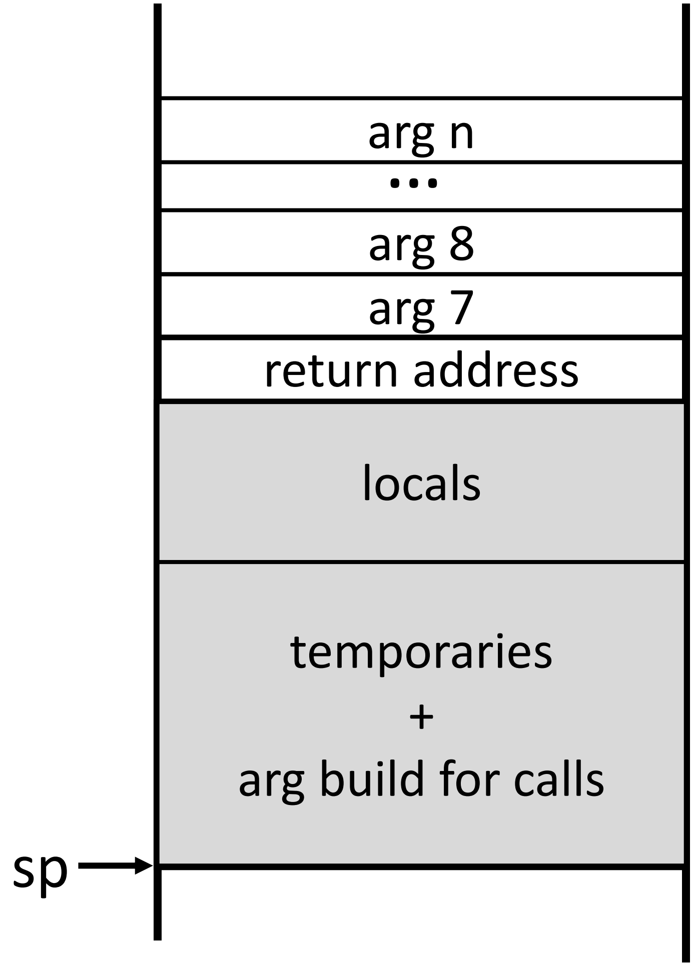
```
Foo: Prolog
     instr1    op1,op2
     instr2    x,y,z
     mov       z,a
     add          r3,r1,r2
     setup for call
     call bar(a,b,c,d)
     recover from call
     instr1    op1,op2
     instr2    x,y,z
     mov       z,a
     add          r3,r1,r2
     Epilog
```



The answer is: it depends!

Prolog

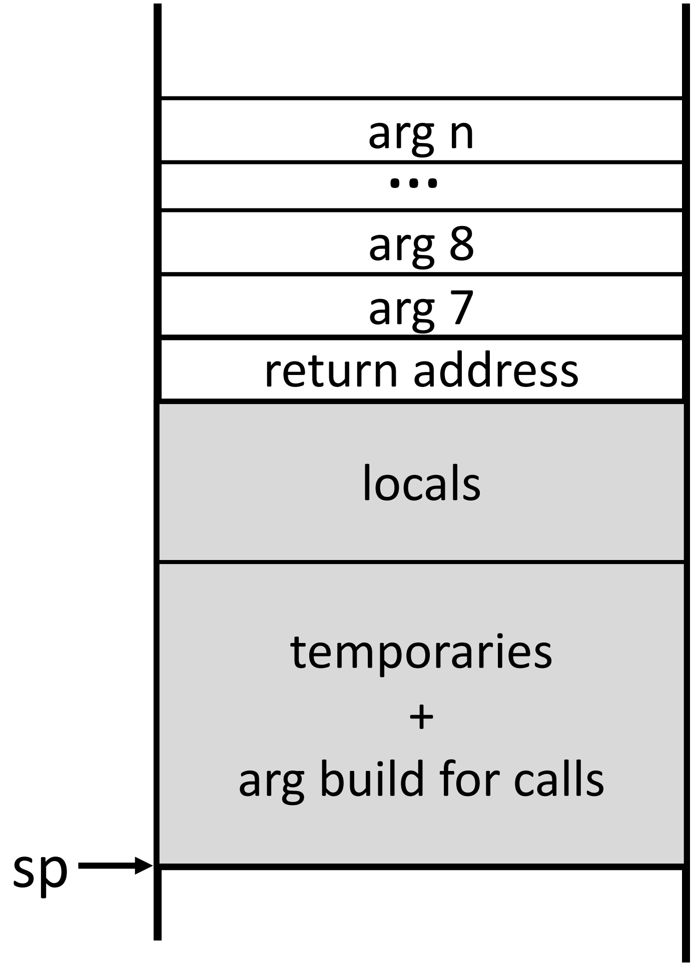
```
Foo: Prolog
     instr1    op1,op2
     instr2    x,y,z
     mov       z,a
     add          r3,r1,r2
     setup for call
     call bar(a,b,c,d)
     recover from call
     instr1    op1,op2
     instr2    x,y,z
     mov       z,a
     add          r3,r1,r2
     Epilog
```



Prolog: adjust sp
save any necessary callee-save registers

Epilog

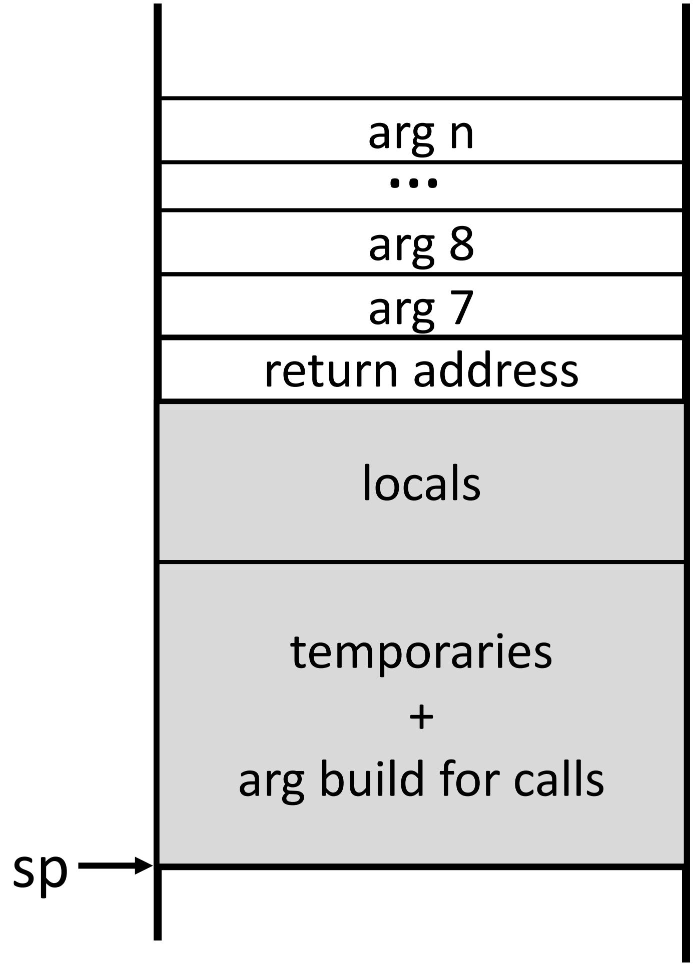
```
Foo: Prolog
       instr1    op1,op2
       instr2    x,y,z
       mov       z,a
       add          r3,r1,r2
       setup for call
       call bar(a,b,c,d)
       recover from call
       instr1    op1,op2
       instr2    x,y,z
       mov       z,a
       add          r3,r1,r2
       Epilog
```



epilog: re-adjust sp
restore any saved callee-save registers

setup for call

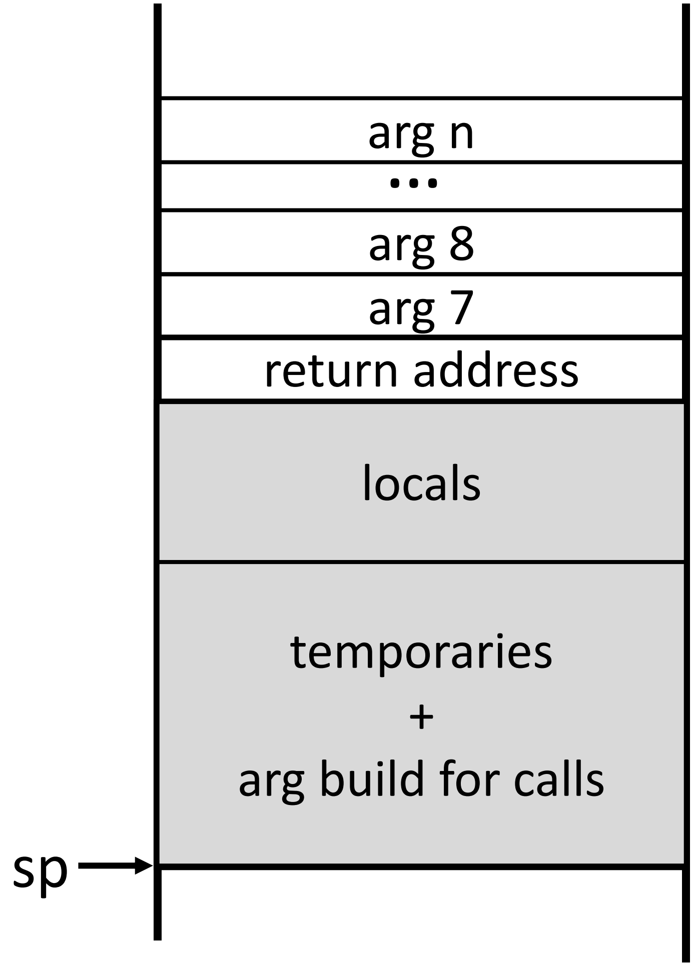
```
Foo: Prolog
       instr1    op1,op2
       instr2    x,y,z
       mov       z,a
       add          r3,r1,r2
       setup for call
       call bar(a,b,c,d)
       recover from call
       instr1    op1,op2
       instr2    x,y,z
       mov       z,a
       add          r3,r1,r2
       Epilog
```



before call: save any necessary caller-save registers
setup arg registers
possibly store 7th, …., nth arg on stack

recover from call

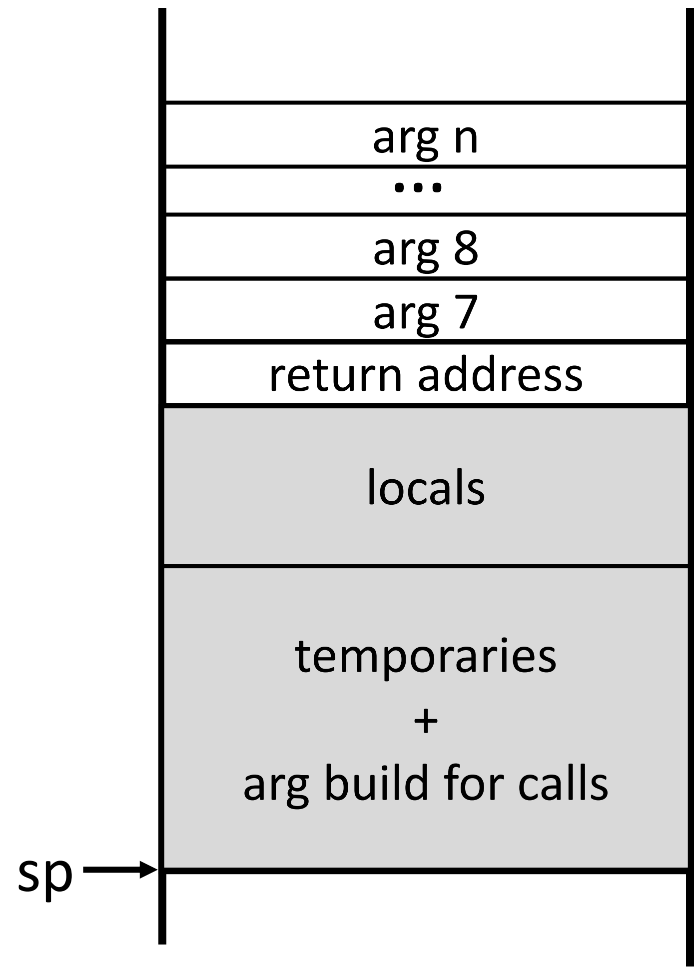
```
Foo: Prolog
     instr1    op1,op2
     instr2    x,y,z
     mov       z,a
     add          r3,r1,r2
     setup for call
     call bar(a,b,c,d)
     recover from call
     instr1    op1,op2
     instr2    x,y,z
     mov       z,a
     add          r3,r1,r2
     Epilog
```



after call: restore any saved caller-save registers

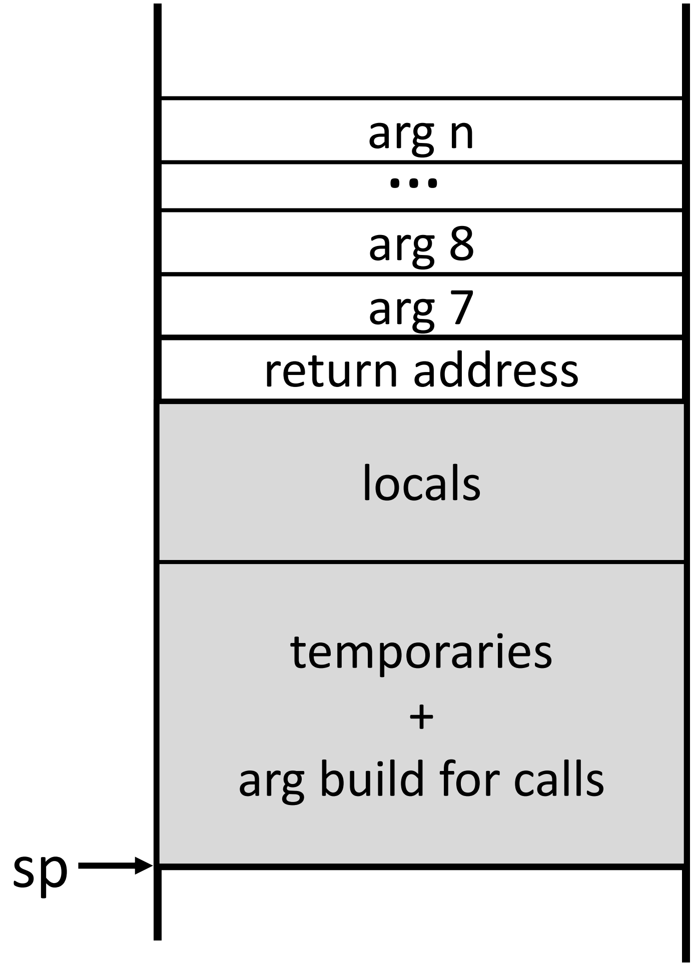
What are “locals” and “temps”?

- What gets saved in the frame?



What are "locals" and "temps"?

- What gets saved in the frame?
  - spilled automatic variables
  - escaping variables
- Escaping variables:
  - referenced in inner function
  - address taken
  - passed by reference
  - Can determine at semantic analysis time with recursive walk of AST
- When do we know?



How to Represent locations

- Various kinds of locations
  - automatic variables: temp
  - parameters: temp
  - hard registers: register
  - spill: frame slot
  - global: memory?
  - static: memory?
- When do we know size of frame?
- When do we generate loads and stores?
- How do we simplify our compiler?

Variables

- Three kinds of variables
  - globals & statics
  - local variables
  - formal parameters
- Issues:
  - where are they stored
  - How much space do th
  - How are they accessed
- This is both
  - machine dependent
  - and, language dependent

- Use an abstract Access type to represent all variables.
- It will end up being:
  - a Temp
  - a HardReg
  - a Slot
  - a MemoryLocation

Today

- Calling Conventions
- Activation Frames
- IR for Function Calls
- Putting it all together

Translating Function Calls

- function call is an expression in grammer, e.g.,

  int a = foo(bar(1), 2)+4;

- Translation?
- From last time,

  $$\text{tr}(f(e_1,\ldots,e_n)) = \text{<}(\check{e}_1;\ldots;\check{e}_n;\ t \leftarrow f(\hat{e}_1,\ldots,\hat{e}_n)),\ t\text{>}$$

- Evaluate all arguments first so we can use pure expressions in call.
- treat call itself as a “statement” and assign (if needed) return value to a fresh temp

IR for a function call

- Choices:

  $d \leftarrow f(s_1,\ldots,s_n)$

  call f

  %rax $\leftarrow$ call f

- The latter two assume that $s_1,\ldots,s_n$ have either been moved to appropriate arg register or put in proper place on stack.
- Side note on SSA and precolored registers:
  - Explicitly representing **`%rax`** will mean not in SSA form. So, **`call f`** may be preferred, in which case, set def set appropriately.

defs and uses

- Each triple has a potential ‘dest’ and ‘src’s
- It also will have a set of uses and defs (which will include the `dest’ and `src’s)
- For `**`call f`**’
  - defines %rax
  - uses all arg registers needed for the call, e.g., $s_1,\ldots,s_n$
  - ?

defs and uses at call site

- Each triple has a potential ‘dest’ and ‘src’s
- It also will have a set of uses and defs (which will include the `dest’ and `src’s)
- For `**`call f`**’
  - defines %rax
  - uses all arg registers needed for the call, e.g., $s_1,\ldots,s_n$
  - It also defines all caller-save registers!
  - So, call defines: %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11

Register

Abstract form	x86-64 Register	Usage	Preserved accross function calls
res_0	%rax	return value*	No
arg_1	%rdi	argument 1	No
arg_2	%rsi	argument 2	No
arg_3	%rdx	argument 3	No
arg_4	%rcx	argument 4	No
arg_5	%r8	argument 5	No
arg_6	%r9	argument 6	No
ler_7	%r10	caller-saved	No
ler_8	%r11	caller-saved	No
lee_9	%rbx	callee-saved	Yes
lee_{10}	%rbp	callee-saved*	Yes
lee_{11}	%r12	callee-saved	Yes
lee_{12}	%r13	callee-saved	Yes
lee_{13}	%r14	callee-saved	Yes
lee_{14}	%r15	callee-saved	Yes
	%rsp	stack pointer	Yes

What about callee?

- Function must preserve callee-save registers
- Could just save them all in prolog, restore them all at epilog

What about callee?

- Function must preserve callee-save registers
- Could just save them all in prolog, restore them all at epilog
- Wasted work for leaf functions, etc.
- Instead use power of register allocator (i.e., spilling and coalescing)
  - if they are not used, they become nops
  - If there is register pressure, then they will be spilled. (assuming spilling cost is calculated right.)

$$
\begin{aligned}
f:\ & t1 \leftarrow lee_9 \\
& t2 \leftarrow lee_{10} \\
& \ldots \\
& lee_{10} \leftarrow t2 \\
& lee_9 \leftarrow t1
\end{aligned}
$$

What about callee?

- Function must preserve callee-save registers
- Could just save them all in prolog, restore them all at epilog
- Wasted work for leaf functions, etc.
- Instead use power of register allocator (i.e., spilling and coalescing)
- What this means for `ret'?

What about callee?

- Function must preserve callee-save registers
- Could just save them all in prolog, restore them all at epilog
- Wasted work for leaf functions, etc.
- Instead use power of register allocator (i.e., spilling and coalescing)
- What this means for `ret’: All callee-registers are considered used by ret.

Coloring Order?

Abstract form	x86-64 Register	Usage	Preserved accross function calls
res_0	%rax	return value*	No
arg_1	%rdi	argument 1	No
arg_2	%rsi	argument 2	No
arg_3	%rdx	argument 3	No
arg_4	%rcx	argument 4	No
arg_5	%r8	argument 5	No
arg_6	%r9	argument 6	No
ler_7	%r10	caller-saved	No
ler_8	%r11	caller-saved	No
lee_9	%rbx	callee-saved	Yes
lee_{10}	%rbp	callee-saved*	Yes
lee_{11}	%r12	callee-saved	Yes
lee_{12}	%r13	callee-saved	Yes
lee_{13}	%r14	callee-saved	Yes
lee_{14}	%r15	callee-saved	Yes
	%rsp	stack pointer	Yes

%rax? %eax? %al

- So, far 32-bits in %eax
- Spilling callee-save registers, however, requires saving %rax.

Coloring Order?

Abstract form	x86-64 Register	Usage	Preserved accross function calls
res_0	%rax	return value*	No
arg_1	%rdi	argument 1	No
arg_2	%rsi	argument 2	No
arg_3	%rdx	argument 3	No
arg_4	%rcx	argument 4	No
arg_5	%r8	argument 5	No
arg_6	%r9	argument 6	No
ler_7	%r10	caller-saved	No
ler_8	%r11	caller-saved	No
lee_9	%rbx	callee-saved	Yes
lee_{10}	%rbp	callee-saved*	Yes
lee_{11}	%r12	callee-saved	Yes
lee_{12}	%r13	callee-saved	Yes
lee_{13}	%r14	callee-saved	Yes
lee_{14}	%r15	callee-saved	Yes
	%rsp	stack pointer	Yes

Today

- Calling Conventions
- Activation Frames
- IR for Function Calls
- Putting it all together

The power function

```
int pow(int b, int e)
//@requires e >= 0;
{
  if (e == 0)
    return 1;
  else
    return b * pow(b, e-1);
}
```

```
pow(b,e):
  if (e == 0) then done else recurse
done:
  ret 1
recurse:
  t0 <- e - 1
  t1 <- pow(b, t0)
  t2 <- b * t1
  ret t2
```

Liveness Information

program	def	use
pow(b, e) :	b,e	
if ($e ==$ 0) then done else recurse		e
done :		
ret 1		
recurse :		
$t_0 \leftarrow e - 1$	t_0	e b,
$t_1 \leftarrow$ pow(b, t_0)	t_1	t_0
$t_2 \leftarrow b * t_1$	t_2	b,t_1
ret t_2		t_2

Initial Translation with def/use

program	def	use
pow(b, e) :	b, e	
if $(e == 0)$ then done else recurse		e
done :		
ret 1		
recurse :		
$t_0 \leftarrow e - 1$	t_0	e
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1
ret t_2		t_2

Calculating liveness

program	def	use	live-in
$\mathsf{pow}(b, e)$:	b, e		
if $(e == 0)$ then done else recurse		e	
done :			
ret 1			
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	
$t_1 \leftarrow \mathsf{pow}(b, t_0)$	t_1	b, t_0	
$t_2 \leftarrow b * t_1$	t_2	b, t_1	
ret t_2		t_2	t_2

Calculating liveness

program	def	use	live-in
pow(b, e) :	b, e		
if $(e == 0)$ then done else recurse		e	
done :			
ret 1			
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0	
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Calculating liveness

program	def	use	live-in
pow(b, e) :	b, e		
if $(e == 0)$ then done else recurse		e	
done :			
ret 1			
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Calculating liveness

program	def	use	live-in
pow(b, e) :	b, e		
if $(e == 0)$ then done else recurse		e	
done :			
ret 1			
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	b, e
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Calculating liveness

program	def	use	live-in
pow(b, e) :	b, e		
if $(e == 0)$ then done else recurse		e	b, e
done :			
ret 1			
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Next: Arguments & retval explicit

program	def	use	live-in
pow(b, e) :	b, e		
if $(e == 0)$ then done else recurse		e	b, e
done :			
ret 1			
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Making argument's Explicit

program	def	use
pow :	arg_1, arg_2	
$b \leftarrow arg_1$	b	arg_1
$e \leftarrow arg_2$	e	arg_2
if $(e == 0)$ then done else recurse		
done :		
$res_0 \leftarrow 1$	res_0	
ret		res_0
recurse :		
$t_0 \leftarrow e - 1$	t_0	e
$arg_2 \leftarrow t_0$	arg_2	t_0
$arg_1 \leftarrow b$	arg_1	b
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2
$t_1 \leftarrow res_0$	t_1	res_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1
$res_0 \leftarrow t_2$	res_0	t_2
ret		res_0

Missing a def

Where are callee save regs?

Liveness

program	def	use	live-in
pow :	arg_1, arg_2		
$b \leftarrow arg_1$	b	arg_1	
$e \leftarrow arg_2$	e	arg_2	
if $(e == 0)$ then done else recurse			
done :			
$res_0 \leftarrow 1$	res_0		
ret		res_0	
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	
$arg_2 \leftarrow t_0$	arg_2	t_0	
$arg_1 \leftarrow b$	arg_1	b	
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2	
$t_1 \leftarrow res_0$	t_1	res_0	
$t_2 \leftarrow b * t_1$	t_2	b, t_1	
$res_0 \leftarrow t_2$	res_0	t_2	
ret		res_0	res_0

Liveness

program	def	use	live-in
pow :	arg_1, arg_2		
$b \leftarrow arg_1$	b	arg_1	arg_1, arg_2
$e \leftarrow arg_2$	e	arg_2	b, arg_2
if $(e == 0)$ then done else recurse			b, e
done :			
$res_0 \leftarrow 1$	res_0		
ret		res_0	res_0
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e
$arg_2 \leftarrow t_0$	arg_2	t_0	b, t_0
$arg_1 \leftarrow b$	arg_1	b	b, arg_2
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2	b, arg_1, arg_2
$t_1 \leftarrow res_0$	t_1	res_0	b, res_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
$res_0 \leftarrow t_2$	res_0	t_2	t_2
ret		res_0	res_0

Calculating Interference Graph

program	def	use	live-in
pow :	arg_1, arg_2		
$b \leftarrow arg_1$	b	arg_1	arg_1, arg_2
$e \leftarrow arg_2$	e	arg_2	b, arg_2
if $(e == 0)$ then done else recurse			b, e
done :			
$res_0 \leftarrow 1$	res_0		
ret		res_0	res_0
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e
$arg_2 \leftarrow t_0$	arg_2	t_0	b, t_0
$arg_1 \leftarrow b$	arg_1	b	b, arg_2
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2	b, arg_1, arg_2
$t_1 \leftarrow res_0$	t_1	res_0	b, res_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
$res_0 \leftarrow t_2$	res_0	t_2	t_2
ret		res_0	res_0

b—arg_2

b—e

b—res_0

b—t_0

b—arg_1

b—arg_3—arg_4—arg_5—arg_6—ler_7—ler_8

b—t_1

temp	interfering with
b	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8, e, t_0, t_1$
e	b
t_0	b
t_1	b
t_2	

Where to put b?

program	def	use	live-in
pow :	arg_1, arg_2		
$b \leftarrow arg_1$	b	arg_1	arg_1, arg_2
$e \leftarrow arg_2$	e	arg_2	b, arg_2
if $(e == 0)$ then done else recurse			b, e
done :			
$res_0 \leftarrow 1$	res_0		
ret		res_0	res_0
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e
$arg_2 \leftarrow t_0$	arg_2	t_0	b, t_0
$arg_1 \leftarrow b$	arg_1	b	b, arg_2
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2	b, arg_1, arg_2
$t_1 \leftarrow res_0$	t_1	res_0	b, res_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
$res_0 \leftarrow t_2$	res_0	t_2	t_2
ret		res_0	res_0

temp	interfering with
b	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8, e, t_0, t_1$
e	b
t_0	b
t_1	b
t_2	

Where to put b?

program	live-in
pow :	arg_1, arg_2, lee_9
push lee_9	arg_1, arg_2, lee_9
$b \leftarrow arg_1$	arg_1, arg_2
$e \leftarrow arg_2$	b, arg_2
if $(e == 0)$ then done else recurse	b, e
done :	
$res_0 \leftarrow 1$	
goto exitpow	res_0
recurse :	b, e
$t_0 \leftarrow e - 1$	b, e
$arg_2 \leftarrow t_0$	b, t_0
$arg_1 \leftarrow b$	b, arg_2
call pow	b, arg_1, arg_2
$t_1 \leftarrow res_0$	b, res_0
$t_2 \leftarrow b * t_1$	b, t_1
$res_0 \leftarrow t_2$	t_2
goto exitpow	res_0
exitpow :	res_0
pop lee_9	res_0
ret	lee_9, res_0

- We added epilog
- save and restore lee_9
- Make all returns goto epilog

Post coloring

pow :
  push lee_9
  $lee_9 \leftarrow arg_1$
  $res_0 \leftarrow arg_2$
  if $(res_0 == 0)$ then done else recurse
done :
  $res_0 \leftarrow 1$
  goto exitpow
recurse :
  $res_0 \leftarrow res_0 - 1$
  $arg_2 \leftarrow res_0$
  $arg_1 \leftarrow lee_9$ (redundant)
  call pow
  $res_0 \leftarrow res_0$ (redundant)
  $res_0 \leftarrow lee_9 * res_0$
  $res_0 \leftarrow res_0$ (redundant)
  goto exitpow
exitpow :
  pop lee_9
  ret

Final

pow :
  push lee_9
  $lee_9 \leftarrow arg_1$
  $res_0 \leftarrow arg_2$
  if $(res_0 == 0)$ then done else recurse
done :
  $res_0 \leftarrow 1$
  goto exitpow
recurse :
  $res_0 \leftarrow res_0 - 1$
  $arg_2 \leftarrow res_0$
  $arg_1 \leftarrow lee_9$
  call pow
  $res_0 \leftarrow res_0$
  $res_0 \leftarrow lee_9 * res_0$
  $res_0 \leftarrow res_0$
  goto exitpow
exitpow :
  pop lee_9
  ret

```
pow:    pushq   %rbx
        movl    %edi, %ebx
        movl    %esi, %eax
        cmpl    $0, %eax
        jne L1
        movl    $1, %eax
        goto L2
L1:     subl    $1, %eax
        movl    %eax, %esi
        call    pow
        imull   %ebx, %eax
L2:     popq %rbx
        ret
```

See you on Tuesday