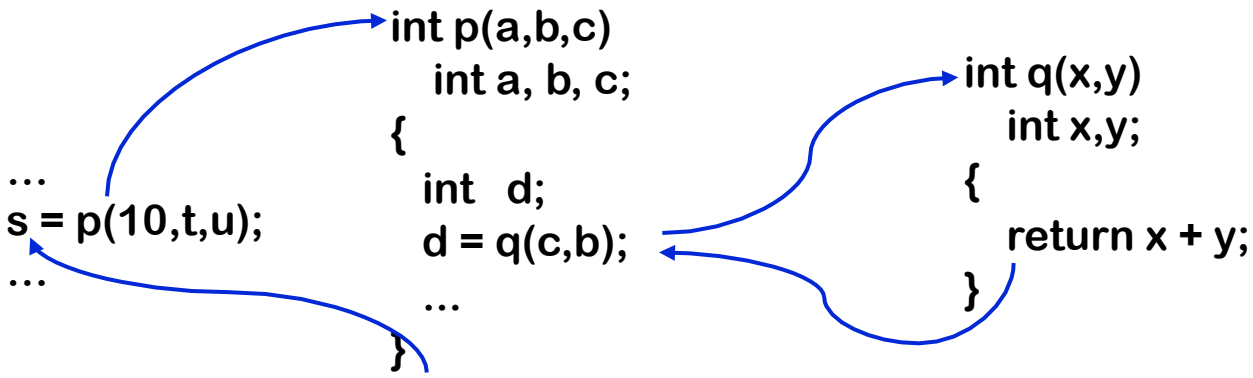
The Procedure Abstraction
Part II: Symbol Tables, Storage

Last Lecture

- Control Abstraction
  - → Well defined entries & exits
  - → Mechanism to return control to caller



The Procedure Abstractions: Today

- Name Space
- External Interface

The Procedure as a Name Space

Why introduce lexical scoping?

- A compile-time mechanism for binding variables
- Lets programmer introduce "local" names

How can compiler keep track of all those names?

```
procedure p {
    int a, b, c
            ....
    {
        int v, b, x, w
            ....
    }
}
```

The Procedure as a Name Space

The Problem

- At point *X*, which declaration of *b* is current?

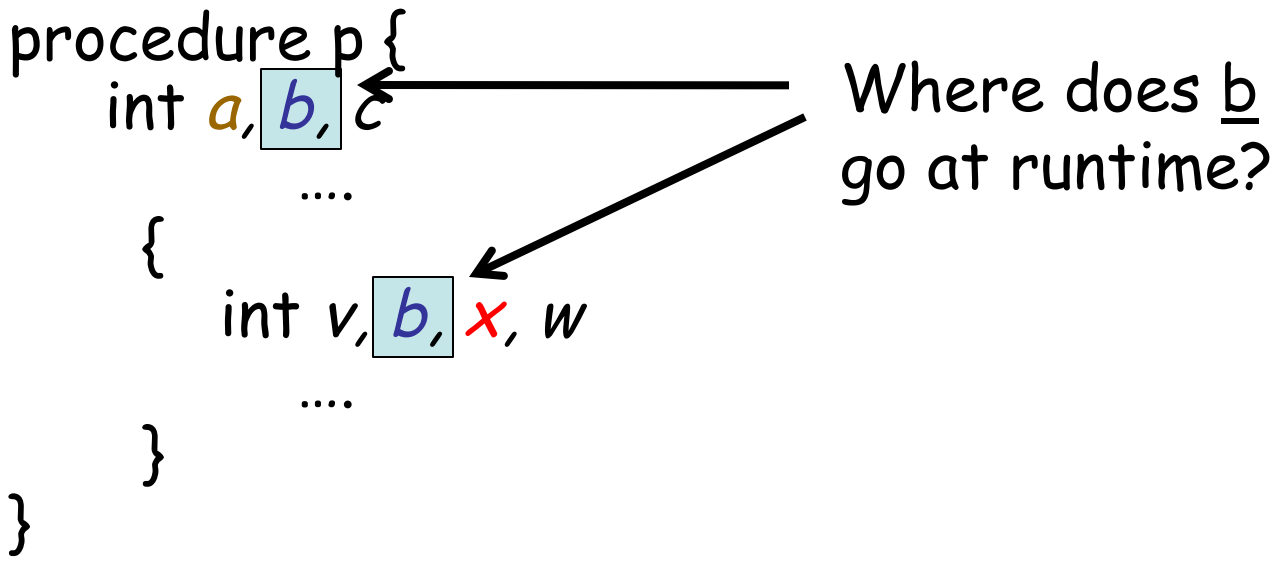
```
procedure p {
    int a, b, c
            ....
    {
        int v, b, x, w
            ....
    }
    ...
}
```

Different points where b can be accessed

The Procedure as a Name Space

The Problem

- At run-time, where is *b* found?



The Procedure as a Name Space

The Problem

- As parser goes in & out of scopes, how does it delete *b*?

```
procedure p {
    int a, b, c
        ....
     {
        int v, b, x, w
        ....
     }
    ...
}
```

As parser goes in and out of scopes, need to delete b.

Answer: Lexically-scoped Symbol Tables

The problem

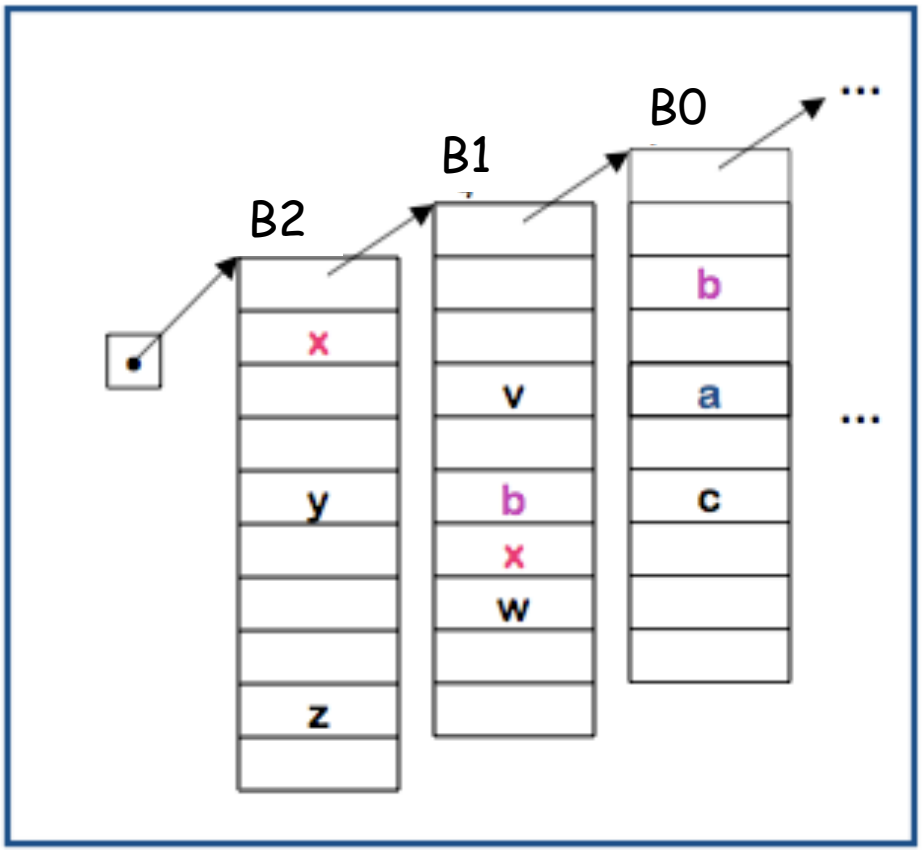
- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The interface

- insert(*name, level*) – creates record for *name* at *level*
- lookup(*name, level*) – returns pointer or index
- delete(*level*) – removes all names declared at *level*

Example

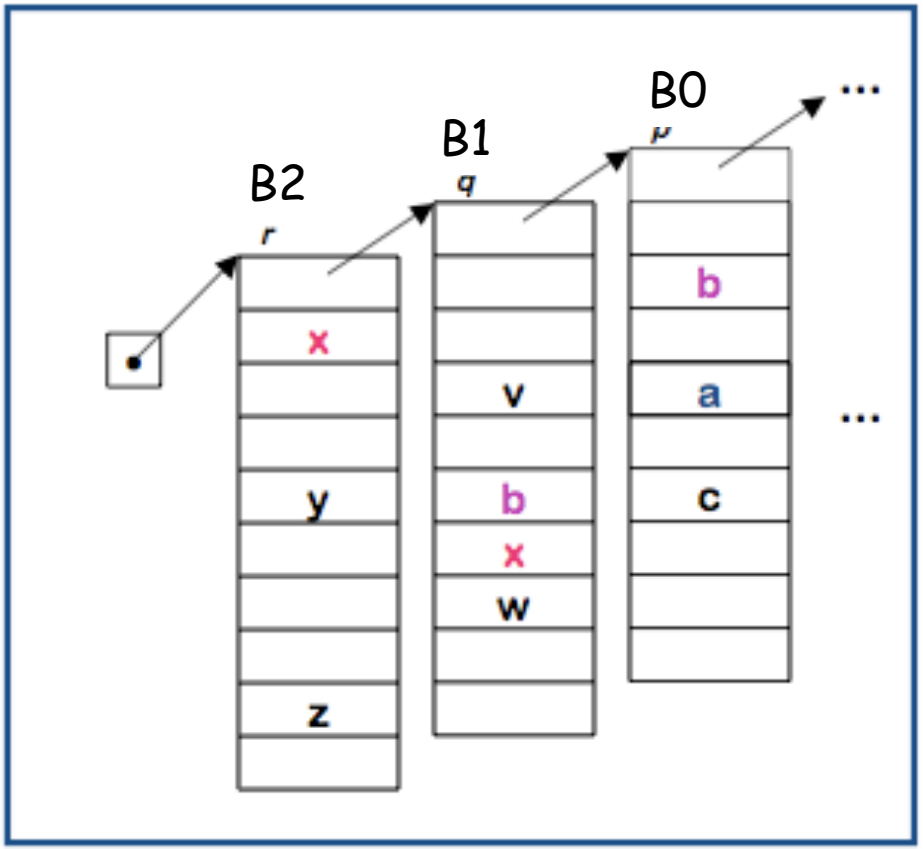
```
B0: procedure b {
      int a, b, c
B1:   {
            int v, b, x, w
B2:         {
              int x, y, z
              ....
            }

B3:         {
              int x, a, v
              ...
            }
            ...
      }
    ...
}
```



Lexically-scoped Symbol Tables

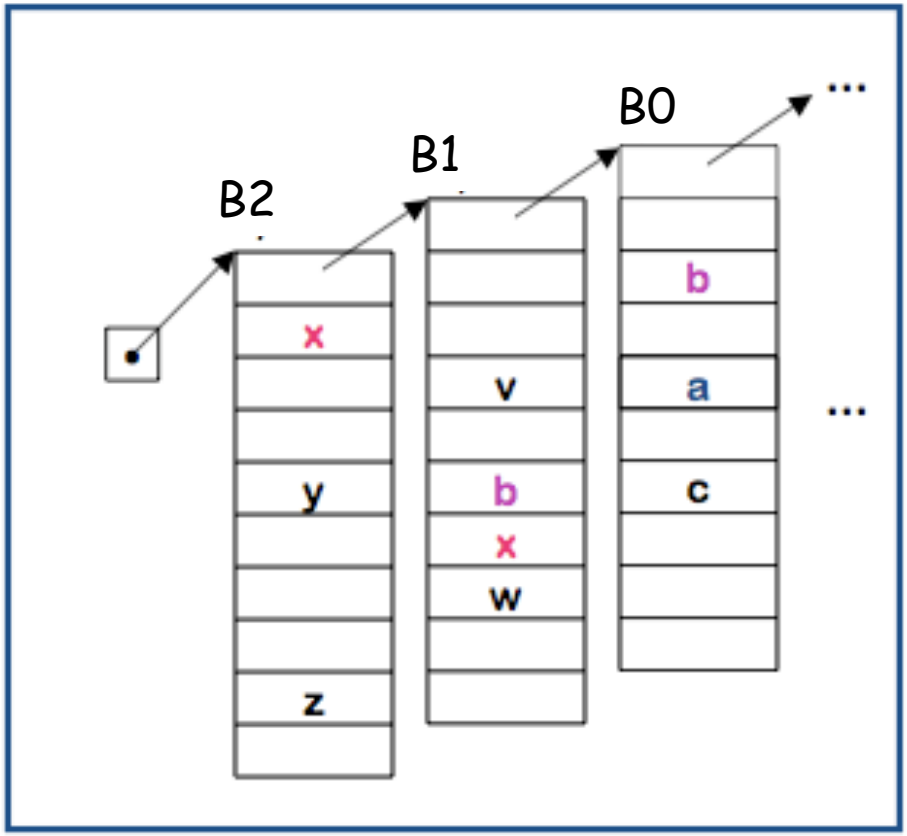
High-level idea

- Create a new table for each scope
- Chain them together for lookup



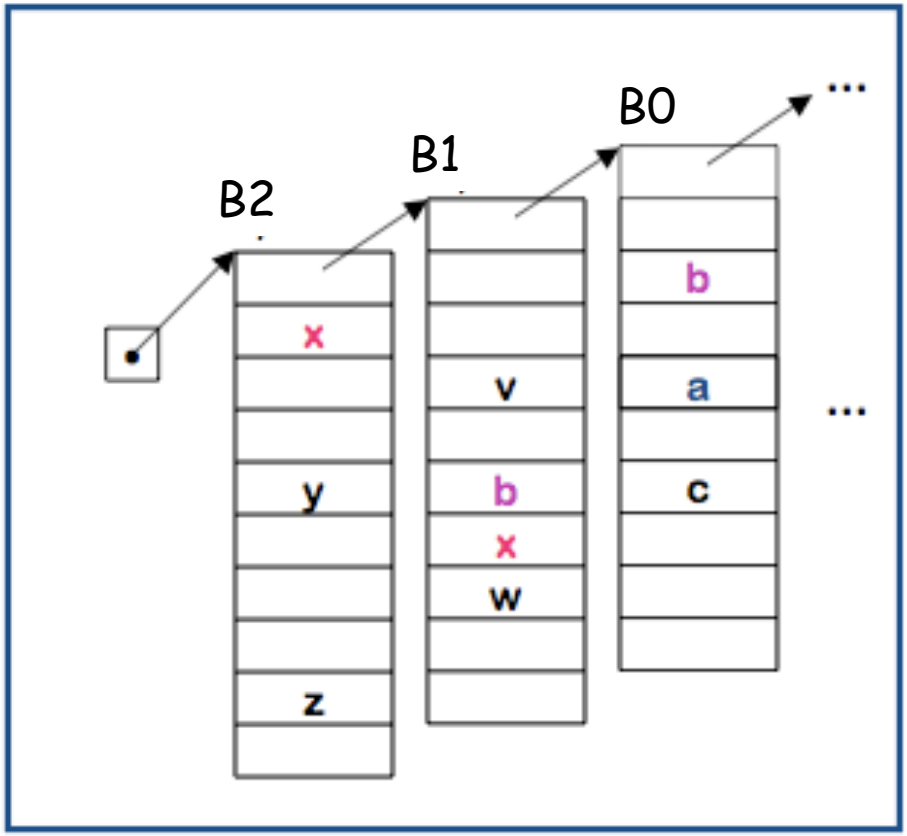
Symbol Table Operations: Insert()

- ***insert()*** **may need to create table**
  it always inserts at current level



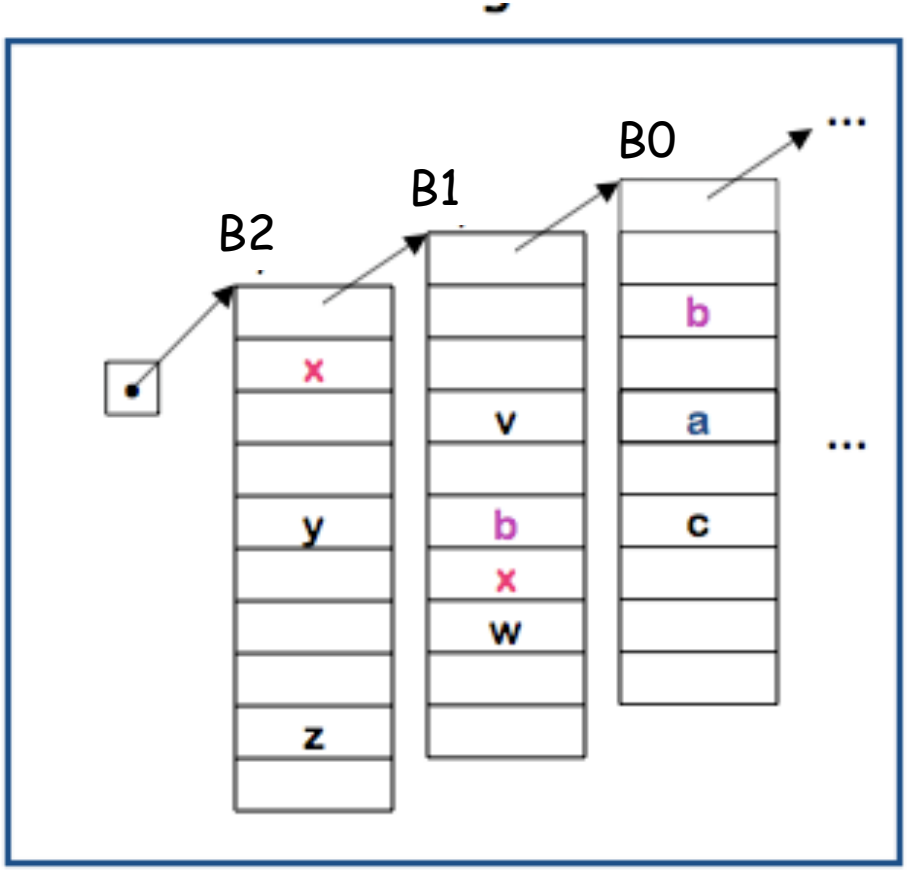
Symbol Table Operations: Lookup()

- ***lookup()*** **walks chain of tables & returns first occurrence of name**



Symbol Table Operations: Delete()

- ***delete()*** **throws away table for level *B0*, if it is top table in the chain**



The Procedure as an External Interface

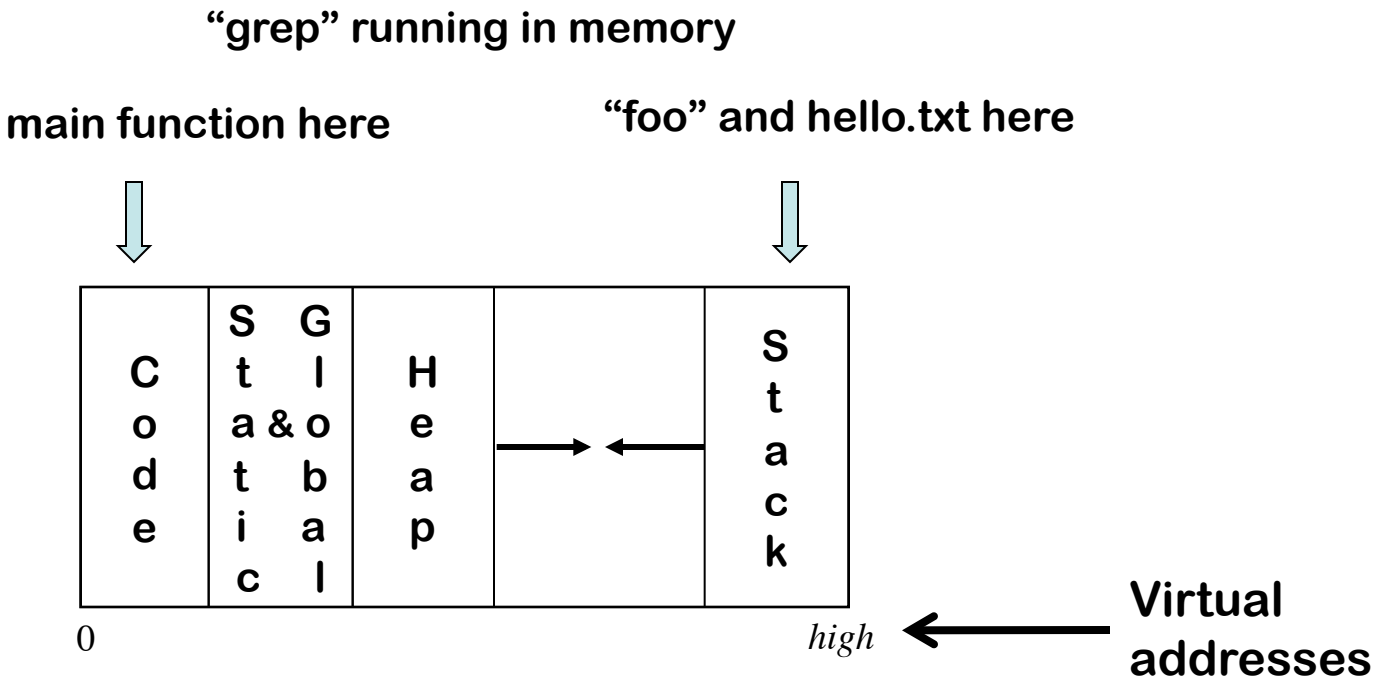
OS needs a way to start the program's execution

- When user invokes "grep" at a command line
  - → OS finds the executable
  - → OS creates a process and arranges for it to run "grep"
  - → "grep" is code from the compiler, linked with run-time system
    - ♦ Starts the run-time environment & calls "main"
    - ♦ After main, it shuts down run-time environment & returns
- When "grep" needs system services
  - → It makes a system call, such as fopen()

The Procedure as an External Interface

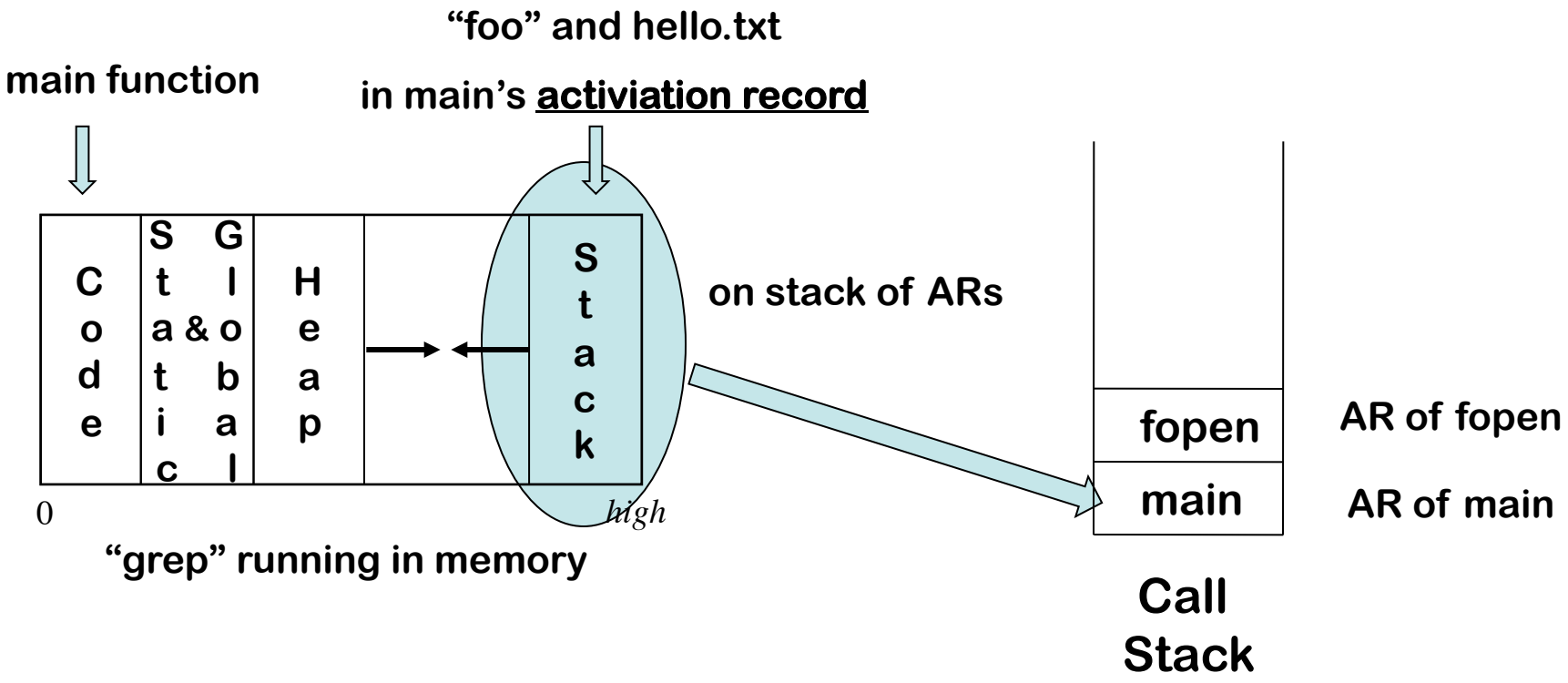
OS needs a way to start the program's execution

> grep "foo" hello.txt

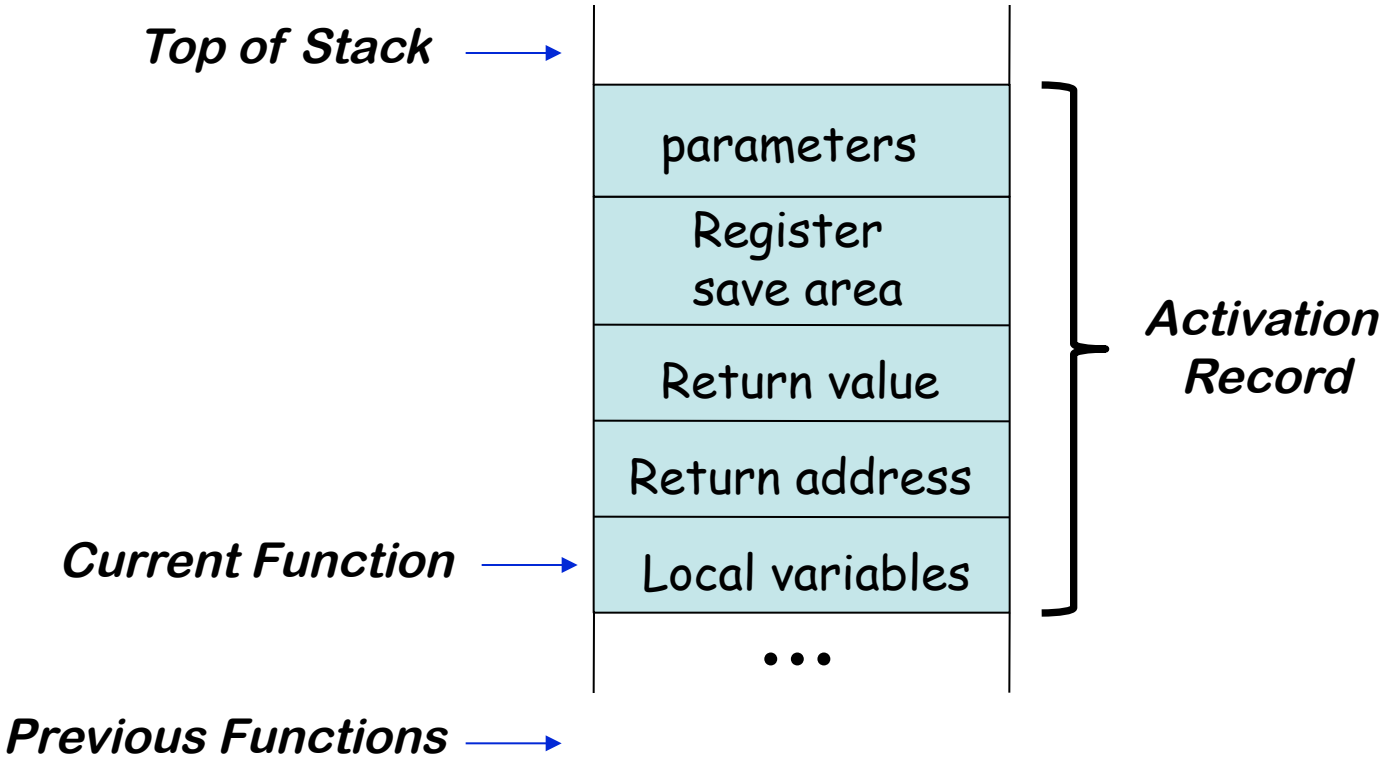


The Procedure as an External Interface

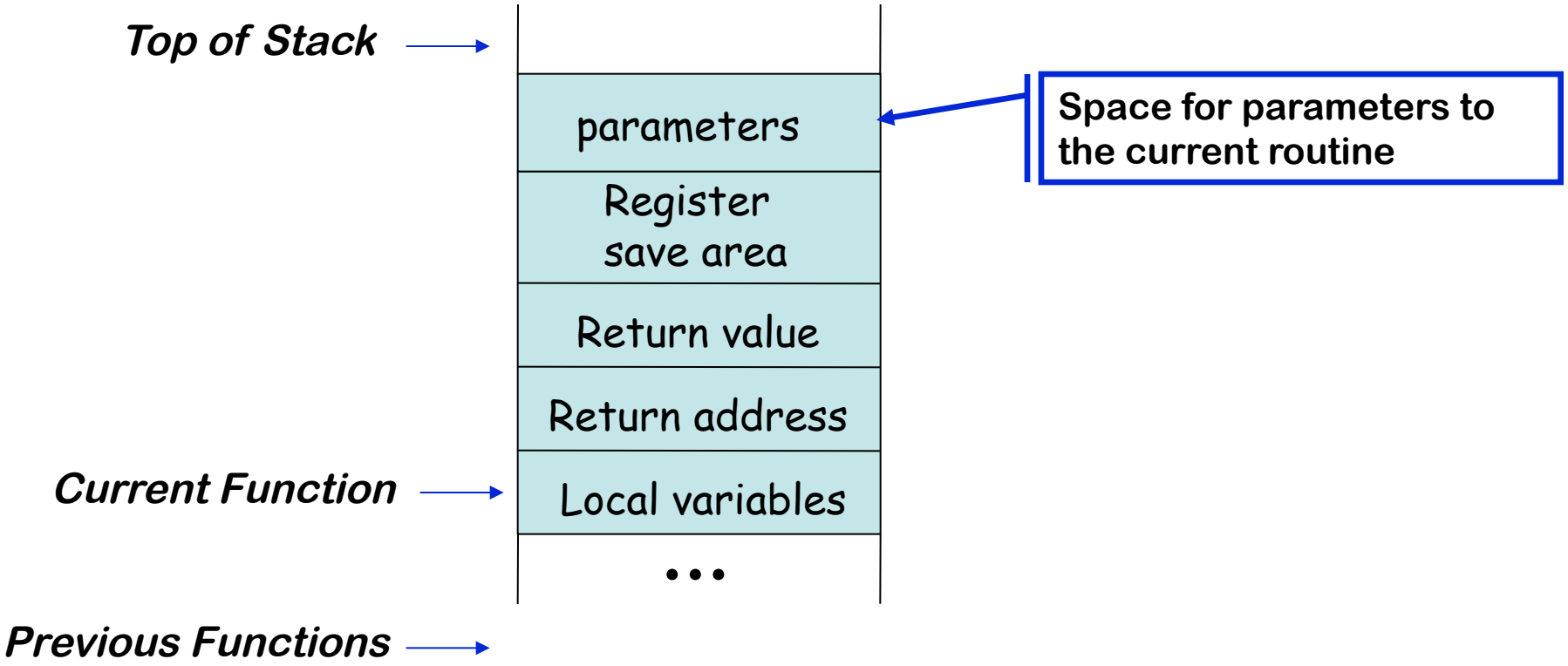
Grep may call fopen with "hello.txt"



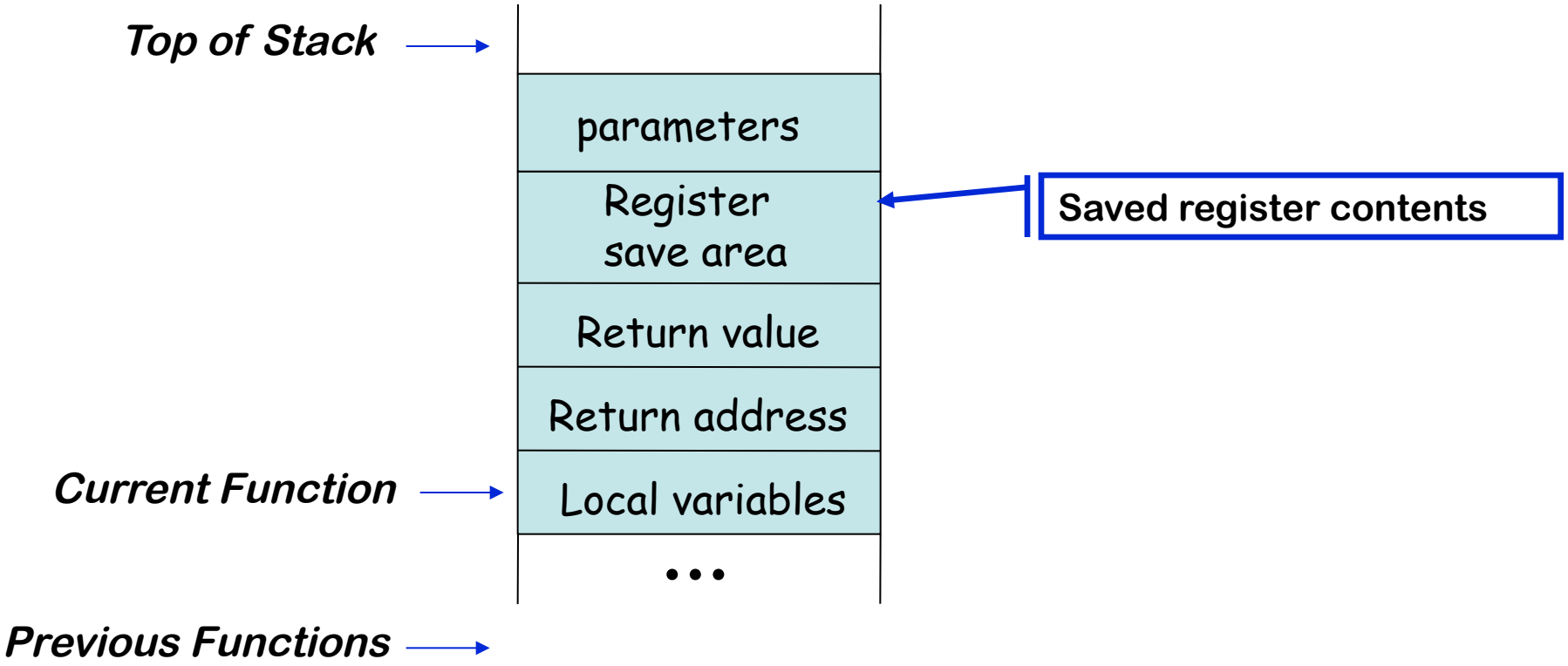
Activation Record



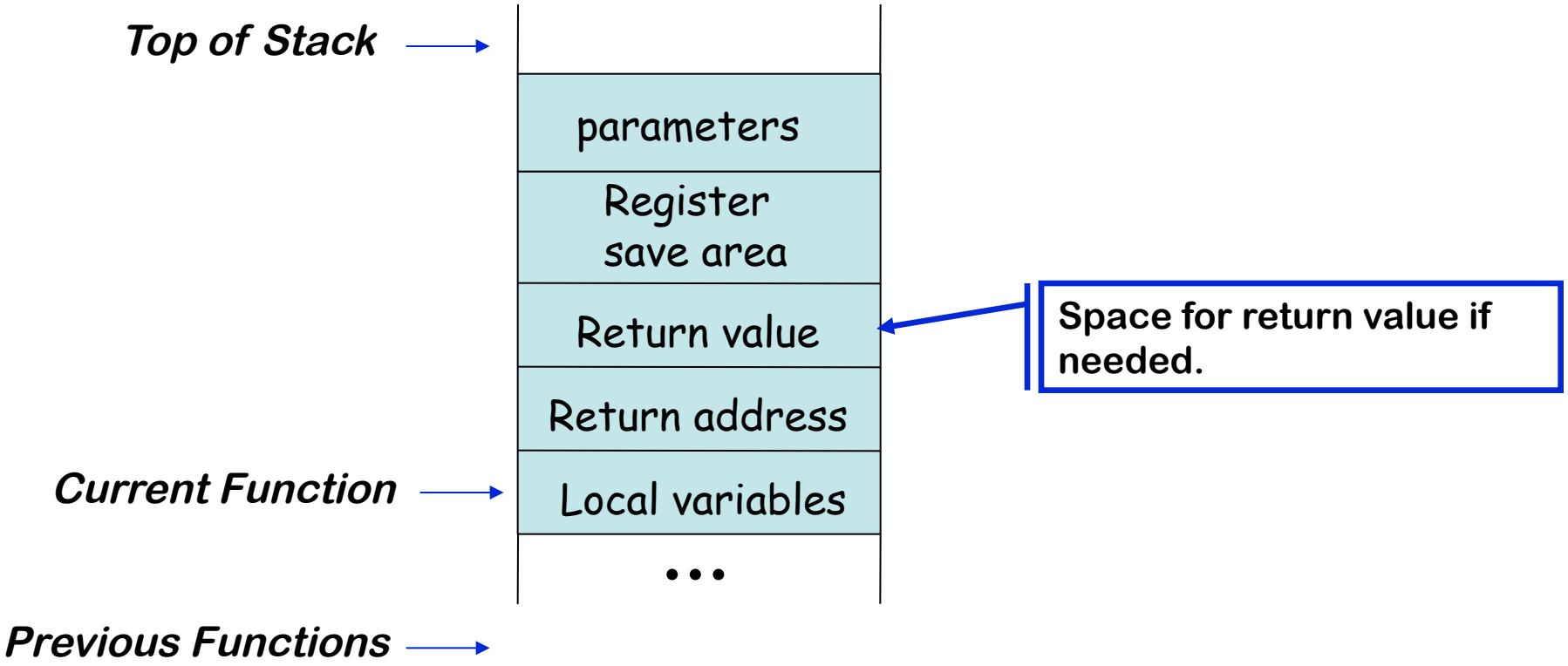
Activation Record



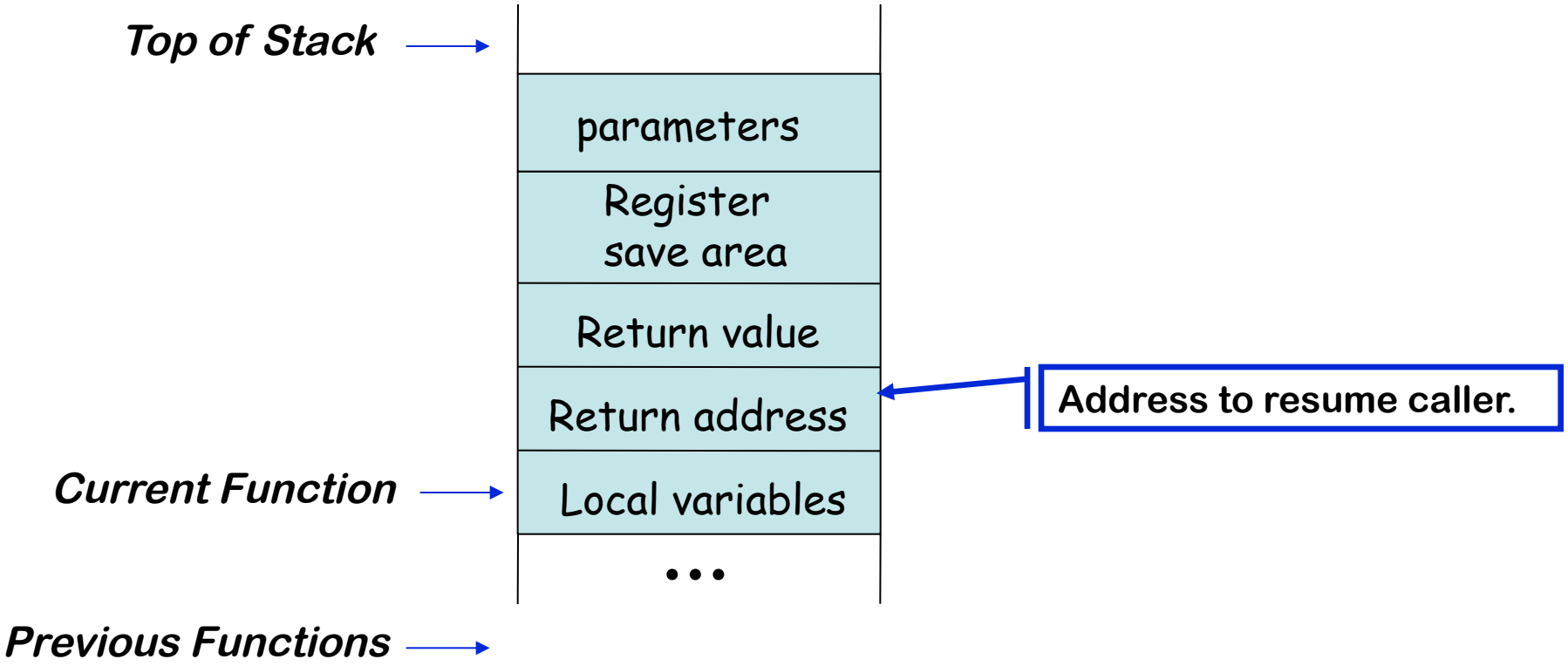
Activation Record



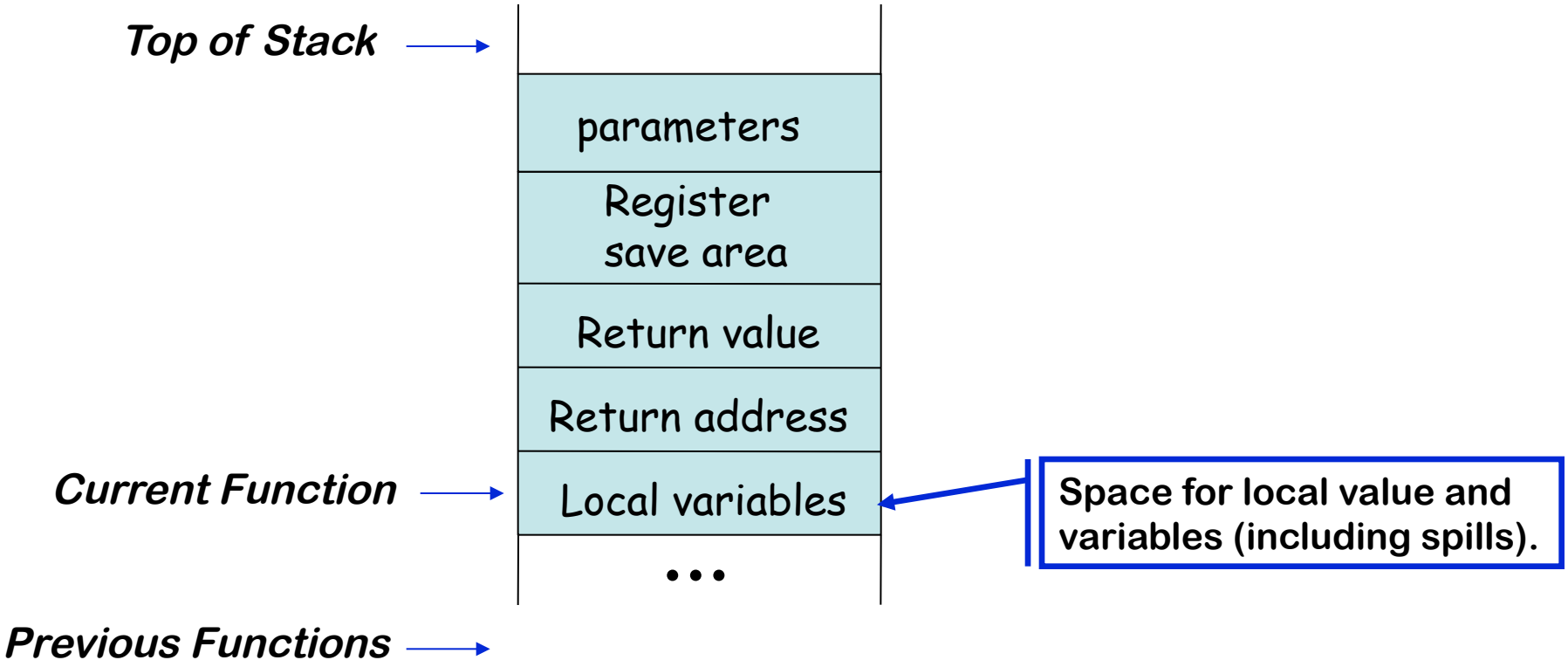
Activation Record



Activation Record



Activation Record



Where Do Local Variables Go?

Local

- Keep them in procedure **activation record** or in a register
- Automatic ⇒ lifetime matches procedure's lifetime

Where Do Static Variables Go?

Static

- File scope ⇒ storage area affixed with file name
- Lifetime is entire execution

Where Do Global Variables Go?

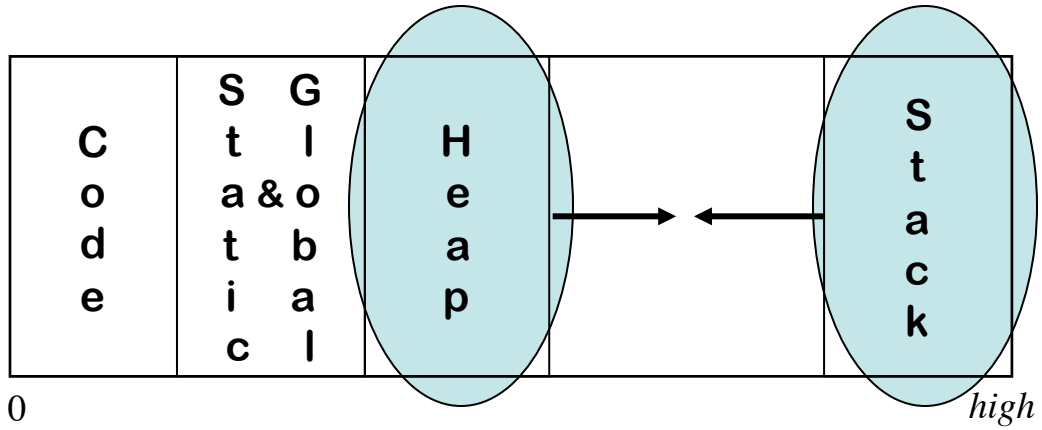
Global

- One or more named global data areas
- One per variable, or per file, or per program, …
- Lifetime is entire execution

Placing Run-time Data Structures

Classic Organization

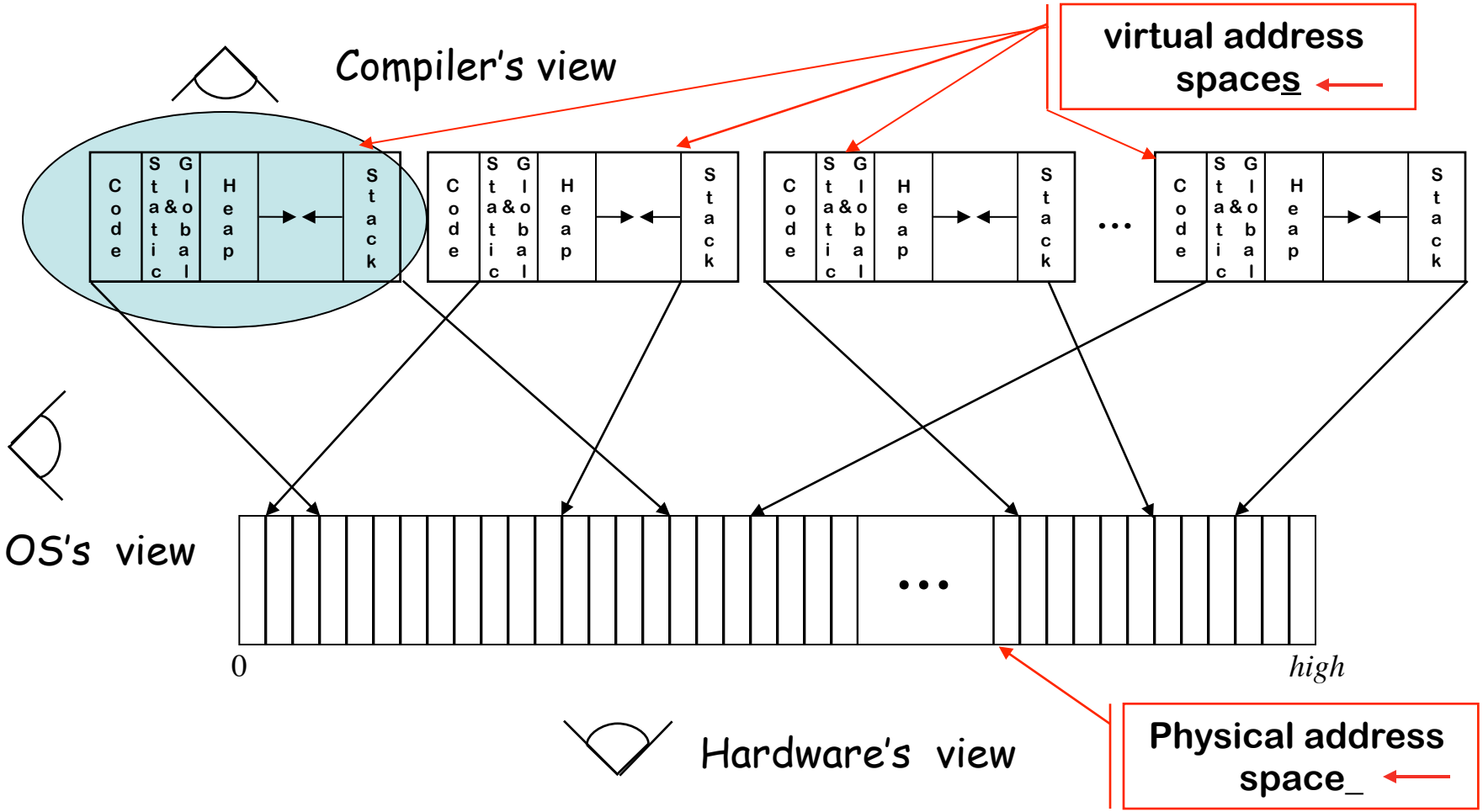
- Code, static, & global data have known size
- Heap & stack both grow & shrink over time
- This is a virtual address space



Single Logical Address Space

How Does This Really Work?

The Big Picture

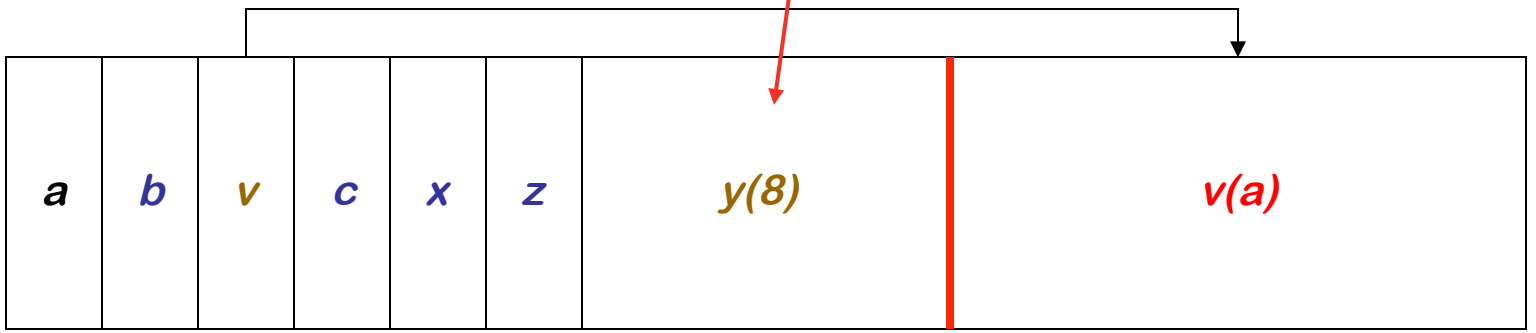


Variable-length Data

```
B0: {
    int a, b
    int v(a), c, x
    int z, y(8)
        ....
    }
```

Arrays

→ If size is fixed at compile time, store in fixed-length data area



Includes variable length data for all blocks in the procedure …

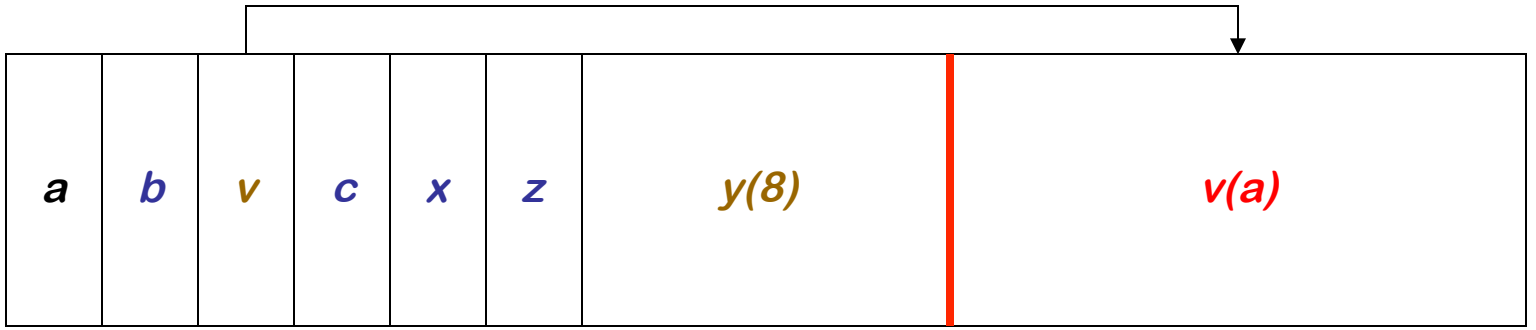
Variable-length data

Variable-length Data

```
B0: {
    int a, b
    int v(a), c, x
    int z, y(8)
          ....
    }
```

Arrays

- → If size is variable, store descriptor in fixed length area, with pointer to variable length area
- → Variable-length data area is assigned at the end of the fixed length area for block in which it is allocated



Includes variable length data for all blocks in the procedure …

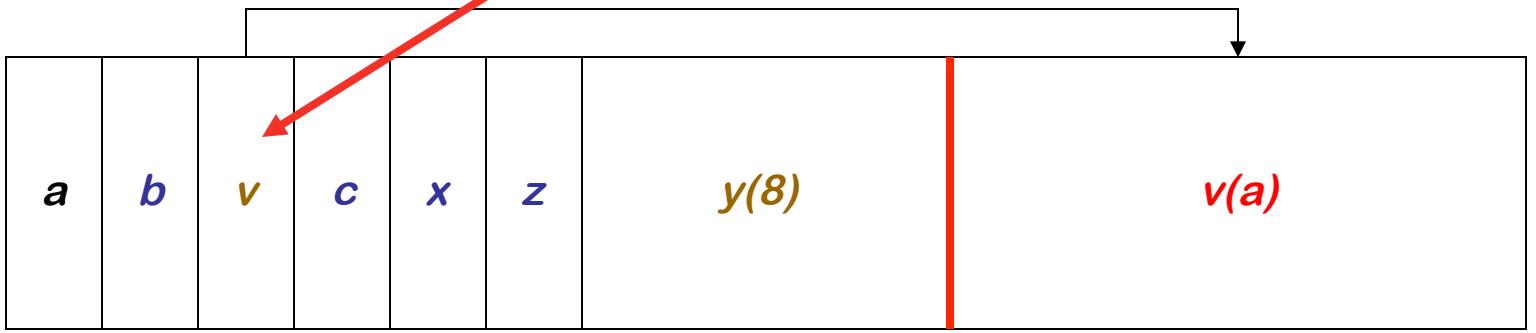
Variable-length data

Variable-length Data

```
B0: {
    int a, b
    int v(a), c, x
    int z, y(8)
          ....
    }
```

Arrays

→ If size is variable, store descriptor in fixed length area, with pointer to variable length area



Includes variable length data for all blocks in the procedure …

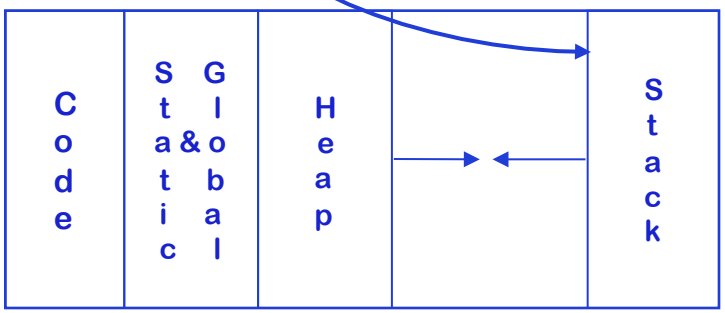
Variable-length data

Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, *AND*
- If code normally executes a "return"

⇒Keep ARs on a stack



Activation Record Details

- If a procedure can outlive its caller, *OR*
- If it can return an object that can reference its execution state

⇒ ARs must be kept in the heap

- If a procedure makes no calls

⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap

Communicating Between Procedures

Most languages provide a parameter passing mechanism

⇒ Expression used at "call site" becomes variable in callee

Two common binding mechanisms

- Call-by-reference passes a pointer to actual parameter
  - → Requires slot in the AR (for address of parameter)
  - → Multiple names with the same address?

```
call fee(x,x,x);
```

- Call-by-value passes a copy of its value at time of call
  - → Requires slot in the AR
  - → Each name gets a unique location *(may have same value)*
  - → Arrays are mostly passed by reference, not value

- Can always use global variables …

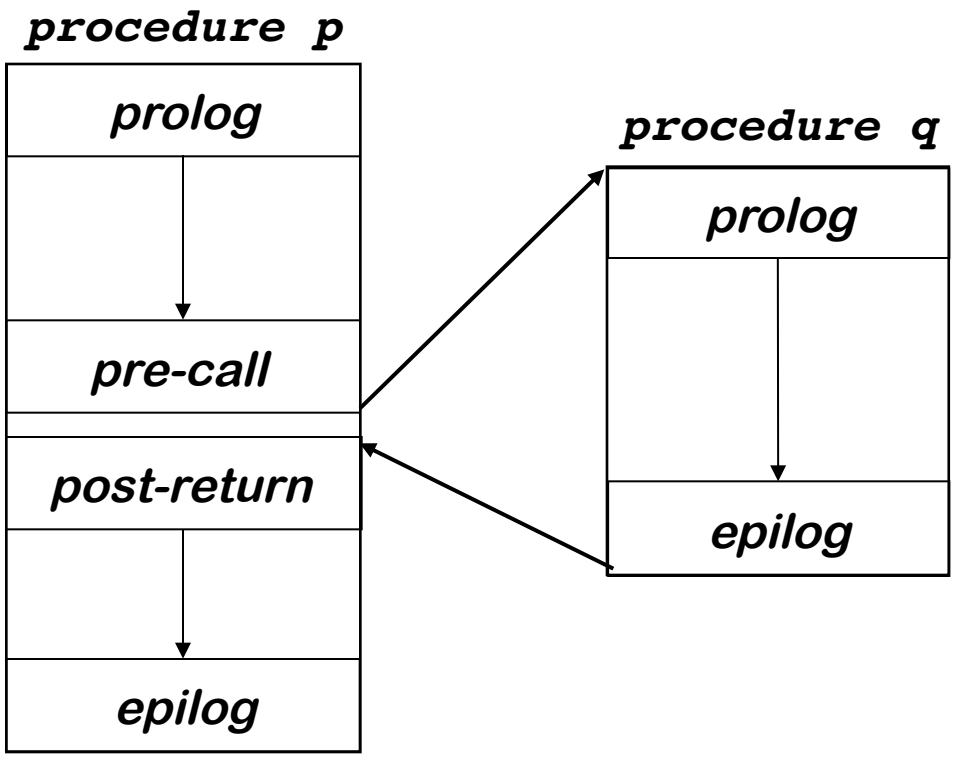
Extra Slides

- The following slides review topics discussed in

Lecture: The Procedure Abstraction, Part I (11/16)

Procedure Linkages

Standard procedure linkage



Procedure has

- standard prolog
- standard epilog

Each call involves a

- pre-call sequence
- post-return sequence

These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters

Procedure Linkages

Pre-call Sequence

- Sets up callee's basic AR
- Helps preserve its own environment

The Details

- Allocate space for the callee's AR
  - → except space for local variables
- Evaluates each parameter & stores value or address
- Saves return address, caller's ARP into callee's AR
- If access links are used
  - → Find appropriate lexical ancestor & copy into callee's **AR**
- Save any caller-save registers
  - → Save into space in caller's AR
- Jump to address of callee's prolog code

Procedure Linkages

Post-return Sequence

- Finish restoring caller's environment
- Place any value back where it belongs

The Details

- Copy return value from callee's AR, if necessary
- Free the callee's AR
- Restore any caller-save registers
- Restore any call-by-reference parameters to registers, if needed
  - → Also copy back call-by-value/result parameters
- Continue execution after the call

Procedure Linkages

Prolog Code

- Finish setting up the callee's environment
- Preserve parts of the caller's environment that will be disturbed

The Details

- Preserve any callee-save registers
- If display is being used
  - → Save display entry for current lexical level
  - → Store current ARP into display for current lexical level
- Allocate space for local data
  - → Easiest scenario is to extend the AR
- Find any static data areas referenced in the callee
- Handle any local variable initializations

With heap allocated AR, may need to use a separate heap object for local variables

Procedure Linkages

Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

The Details

- Store return value? No, this happens on the return statement
- Restore callee-save registers
- Free space for local data, if necessary (on the heap)
- Load return address from AR
- Restore caller's ARP
- Jump to the return address

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop to its pre-call value.)