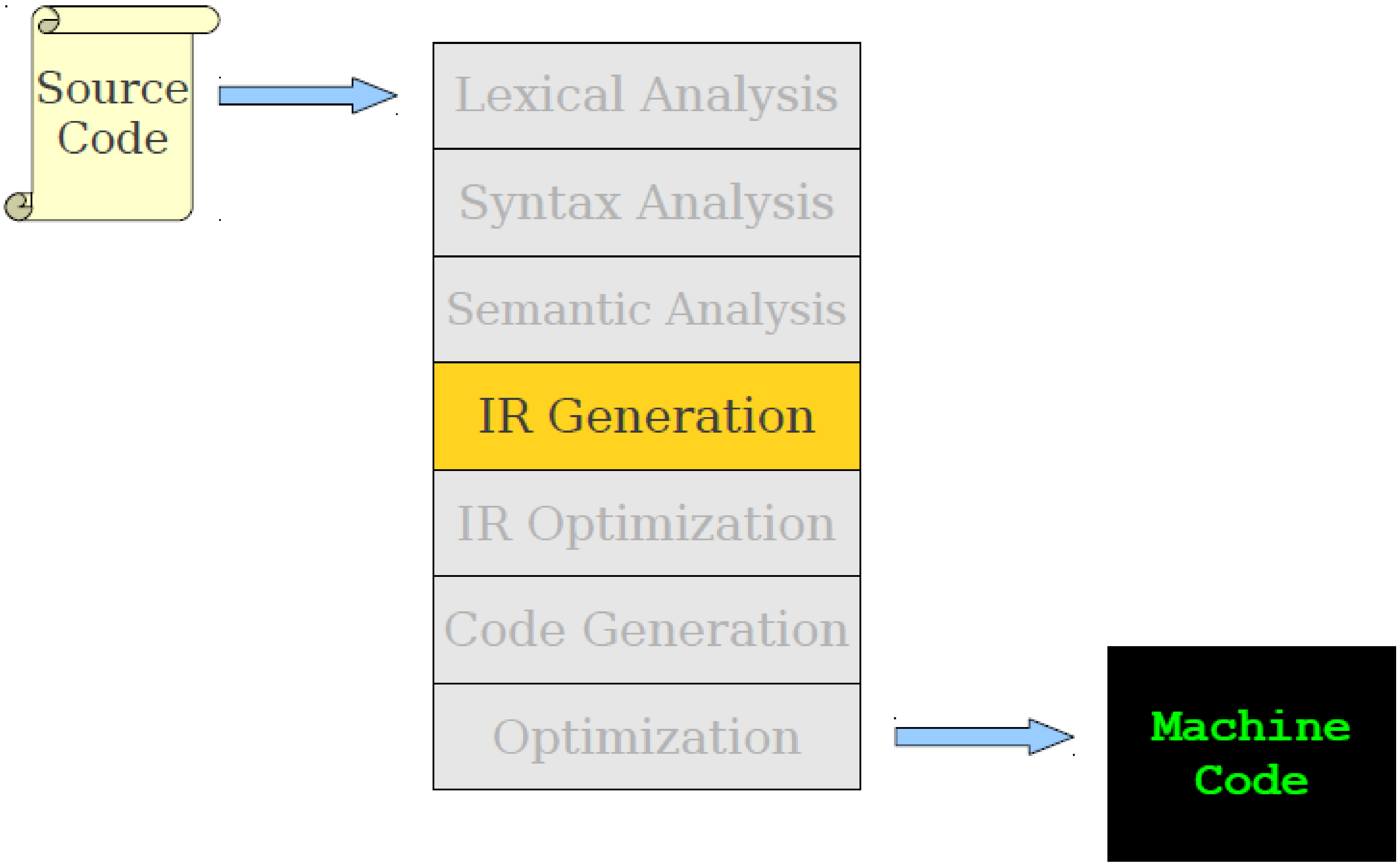
Run-Time Environment

Lecture 13


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

An Important Duality

- Programming languages contain high-level structures:
  - Functions
  - Objects
  - Exceptions
  - Dynamic typing
  - Lazy evaluation
  - (etc.)
- The physical computer only operates in terms of several primitive operations:
  - Arithmetic
  - Data movement
  - Control jumps

Run-time Environment

- Compiler must cooperate with OS and other system software to support implementation of different abstractions on the target machine
  - names,
  - scopes,
  - bindings,
  - data types,
  - operators,
  - procedures,
  - parameters,
  - flow-of-control

Run-time Environment

- Compiler does this by **Run-Time Environment** in which it assumes its target programs are being executed
- Run-Time Environment deals with
  - Layout and allocation of storage
  - Access to variable and data
  - Linkage between procedures
  - Parameter passing
  - Interface to OS, I/O devices etc

Run-Time Environments

- How do we allocate the space for the generated target code and the data object of our source programs?
- The places of the data objects that can be determined at compile time will be allocated statically.
- But the places for the some of data objects will be *allocated at* run-time.

Run-Time Environments

- The allocation and de-allocation of the data objects is managed by the run-time support package.
  - run-time support package is loaded together with the generated target code.
  - the structure of the run-time support package depends on the semantics of the programming language (especially the semantics of procedures in that language).

Procedure Activations

- Each execution of a procedure is called as activation of that procedure.

- An execution of a procedure **P** starts at the beginning of the procedure body;
- When a procedure **P** is completed, it returns control to the point immediately after the place where **P** was called.

- Lifetime of an activation of a procedure **P** is the sequence of steps between the first and the last steps in execution of **P** (including the other procedures called by **P**).

Procedure Activations

- If **A** and **B** are procedure activations, then their lifetimes are either non-overlapping or are nested.
- If a procedure is recursive, a new activation can begin before an earlier activation of the same procedure has ended.
- Activation life time can be depicted as a tree.

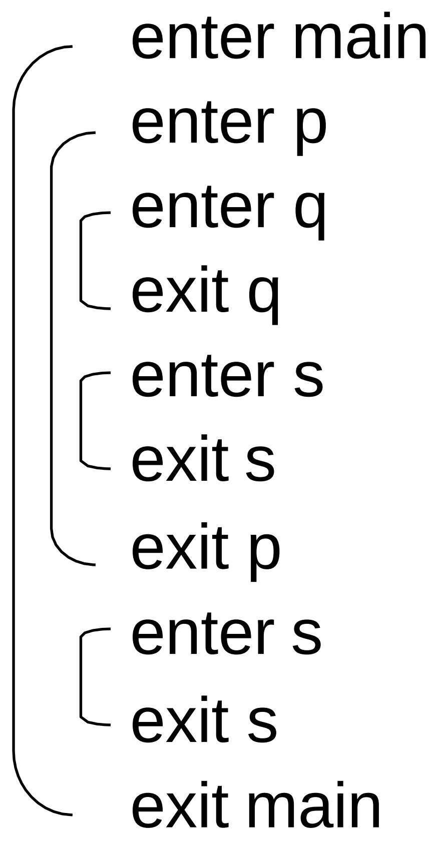
Activation Tree/ Call Tree

- We can use a tree (called **activation tree**) to show the way control enters and leaves activations.

- In an activation tree:
  - Each node represents an activation of a procedure.
  - The root represents the activation of the main program.
  - The node **A** is a parent of the node **B** iff the control flows from **A** to **B**.
  - The node **A** is left to to the node **B** iff the lifetime of **A** occurs before the lifetime of **B**.

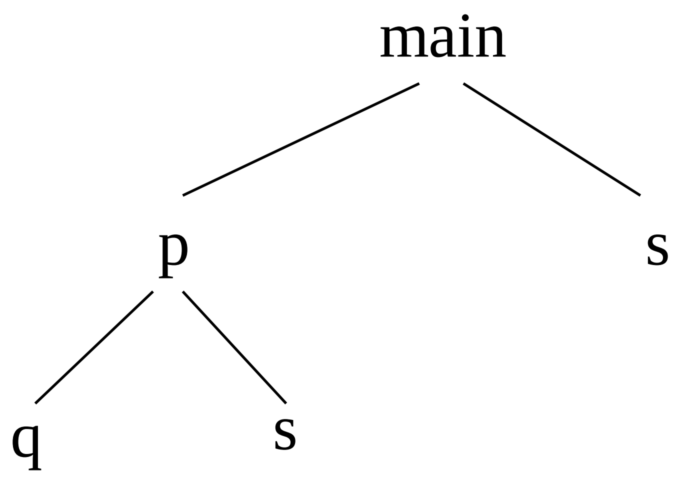
Activation Tree (cont.)

```
program main;
  procedure s;
    begin ... end;
  procedure p;
    procedure q;
      begin ... end;
    begin q; s; end;
  begin p; s; end;
```



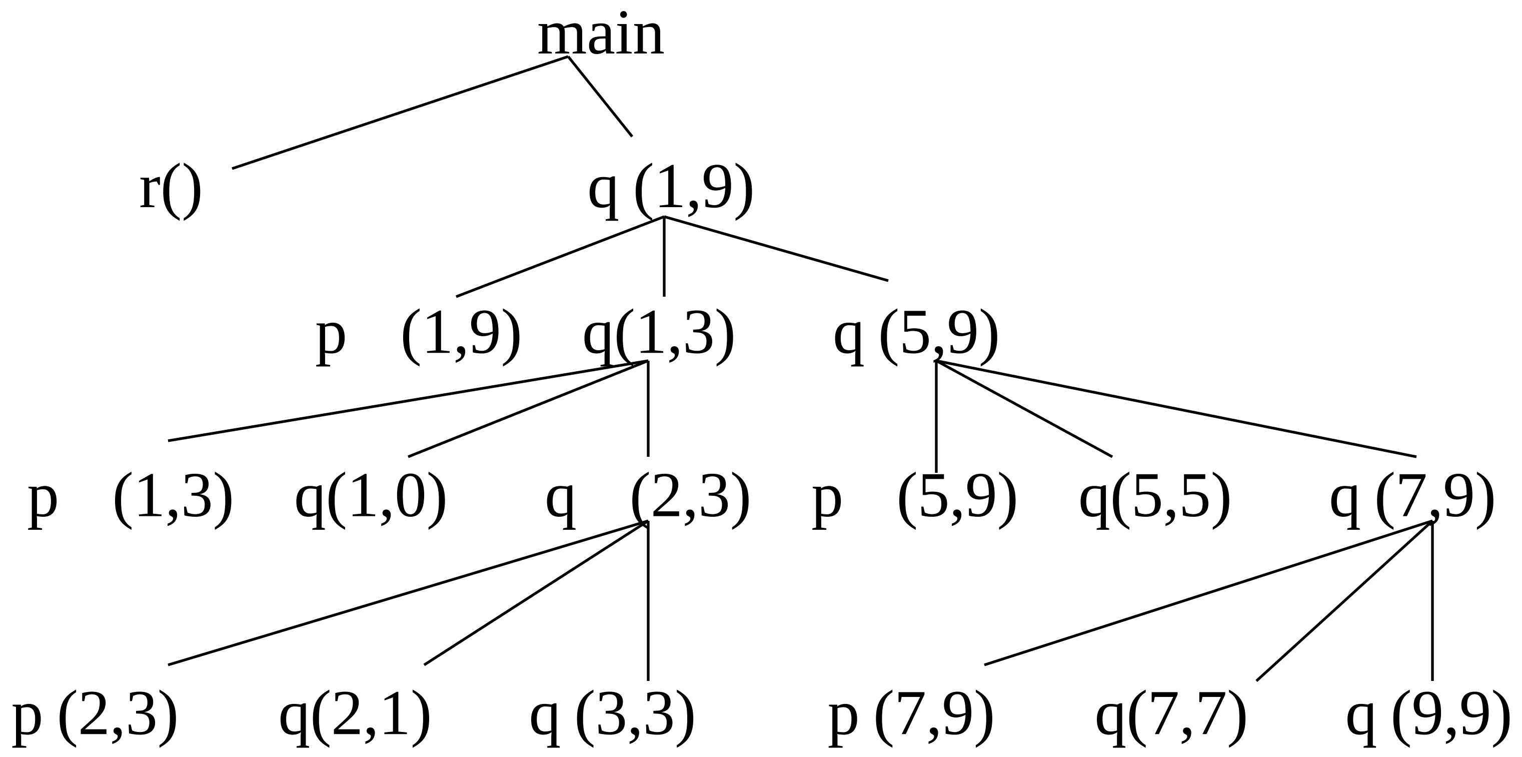
A Nested Structure

Activation Tree (cont.)



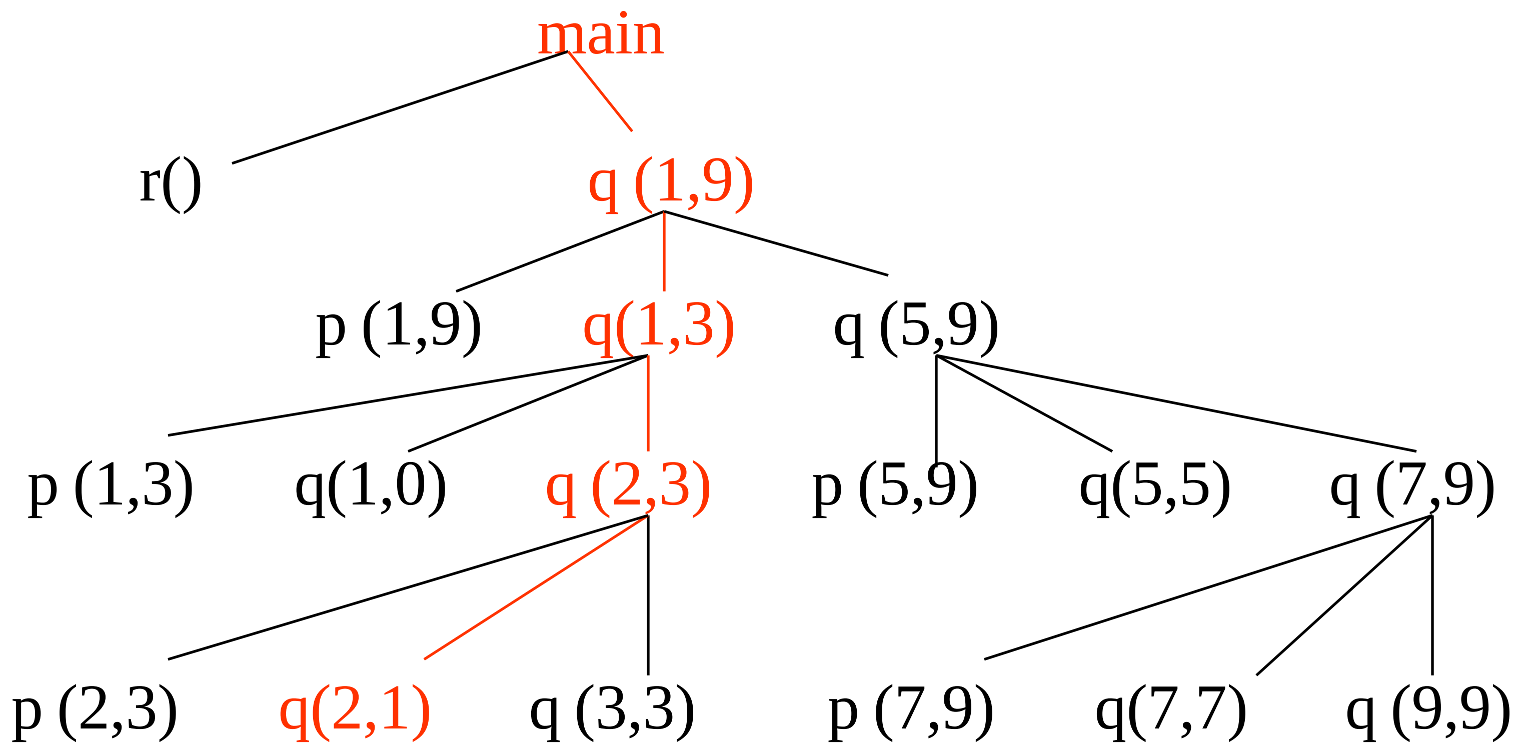
Activation Tree

- Activation Tree - cannot be computed statically
- Dynamic – may be different every time the program is run



Run-time Control Flow

Paths in the activation tree from root to some node represent a sequence of active calls at runtime



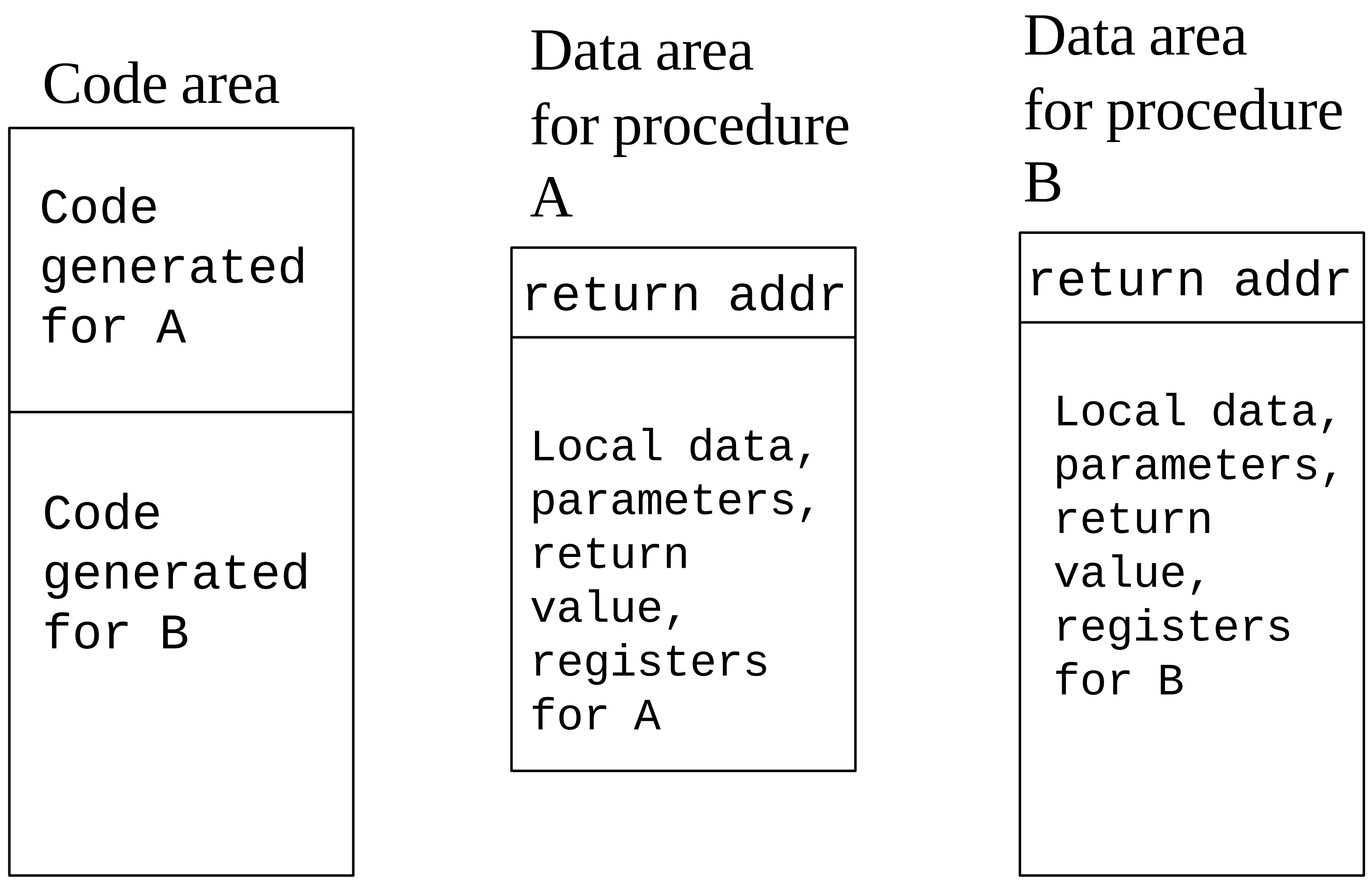
Implementing Run-time control flow

- **Procedure call/return** – when a procedure activation terminates, control returns to the caller of this procedure.
- **Parameter/return value** – values are passed into a procedure activation upon call. For a function, a value may be returned to the caller.
- **Variable addressing** – when using an identifier, language scope rules dictate the binding.

Static Allocation

- Historically, the first approach to solve the run-time control flow problem (Fortran)
- **All** space allocated at compile time
  - **Code area** – machine instructions for each procedure
  - **Static area / procedure call frame/ activation record**
    - single data area allocated for each procedure.
      - local vars, parameters, return value, saved registers
    - return address for each procedure.

Static Allocation

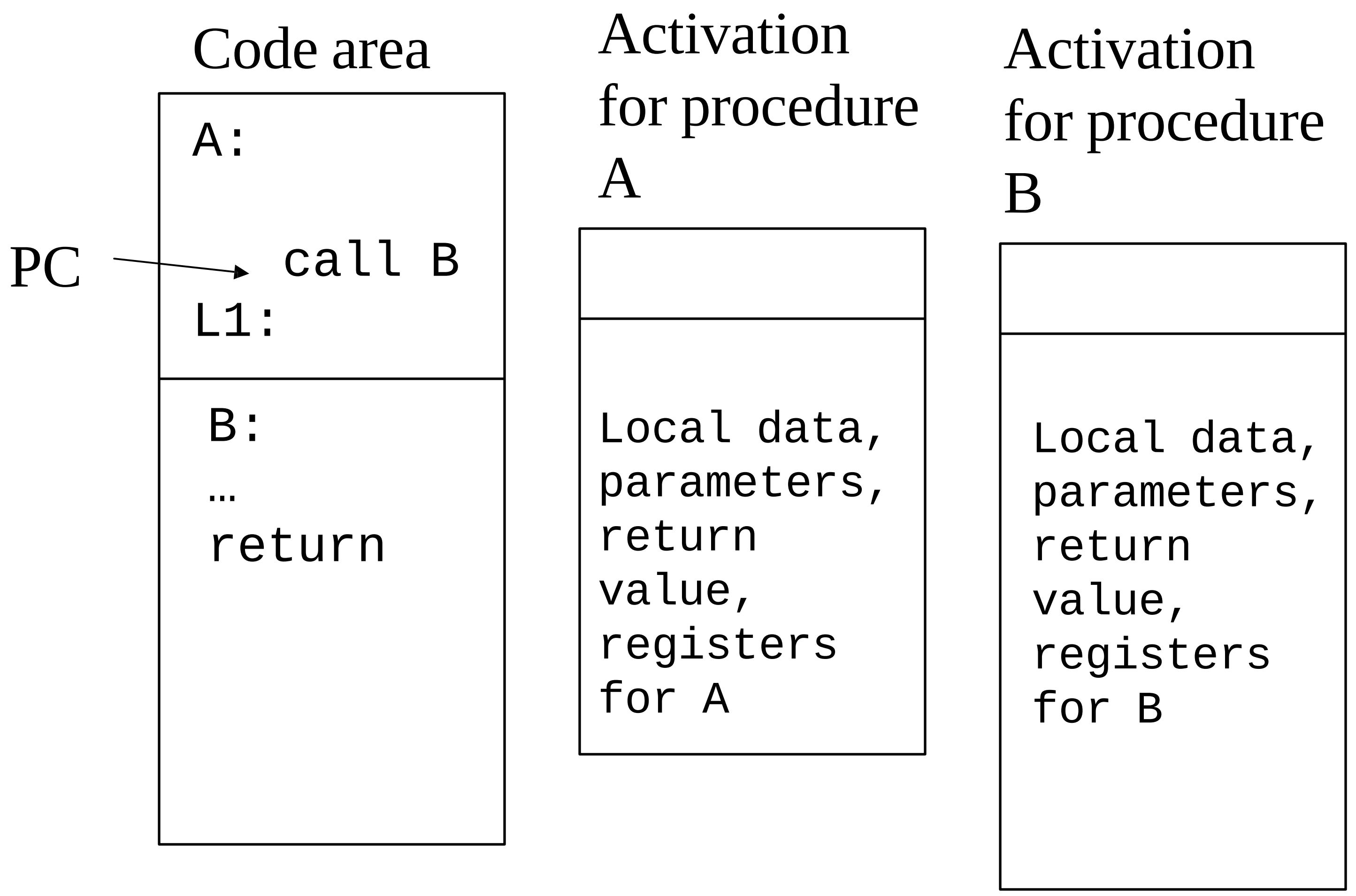

Code area
Code generated for A
Code generated for B
Data area for procedure A
return addr
Local data, parameters, return value, registers for A
Data area for procedure B
return addr
Local data, parameters, return value, registers for B

Call/Return processing in Static Allocation

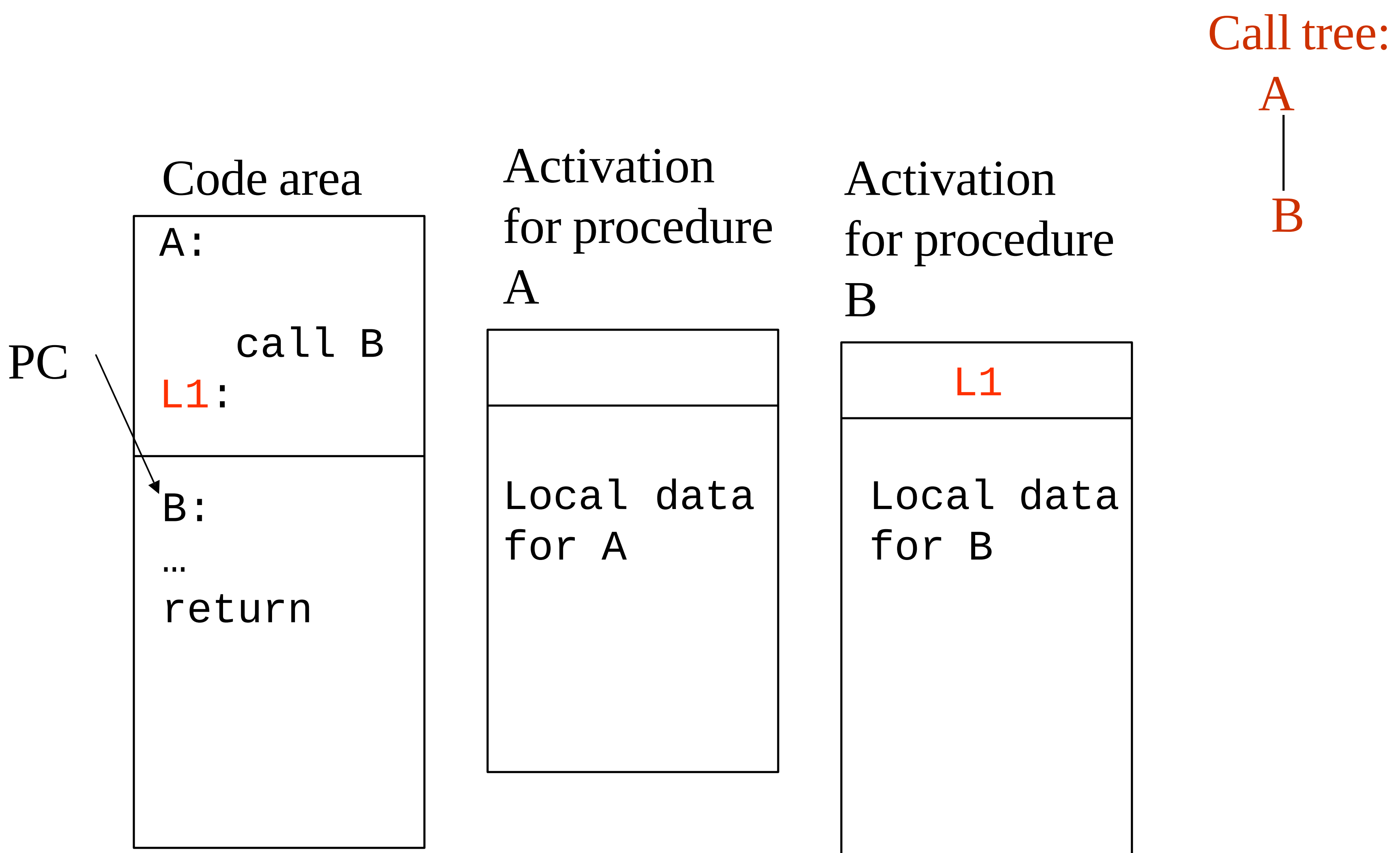
- **When A calls B**:
  - in A: evaluate actual parameters and place into B’s data area, place RA in B’s data area, save any registers or status data needed, update the program counter (PC) to B’s code
- **When the call returns**
  - in B: move return value to known place in data area, update PC to value in RA
  - in A: get return value, restore any saved registers or status data

Static Allocation

Call tree:
A

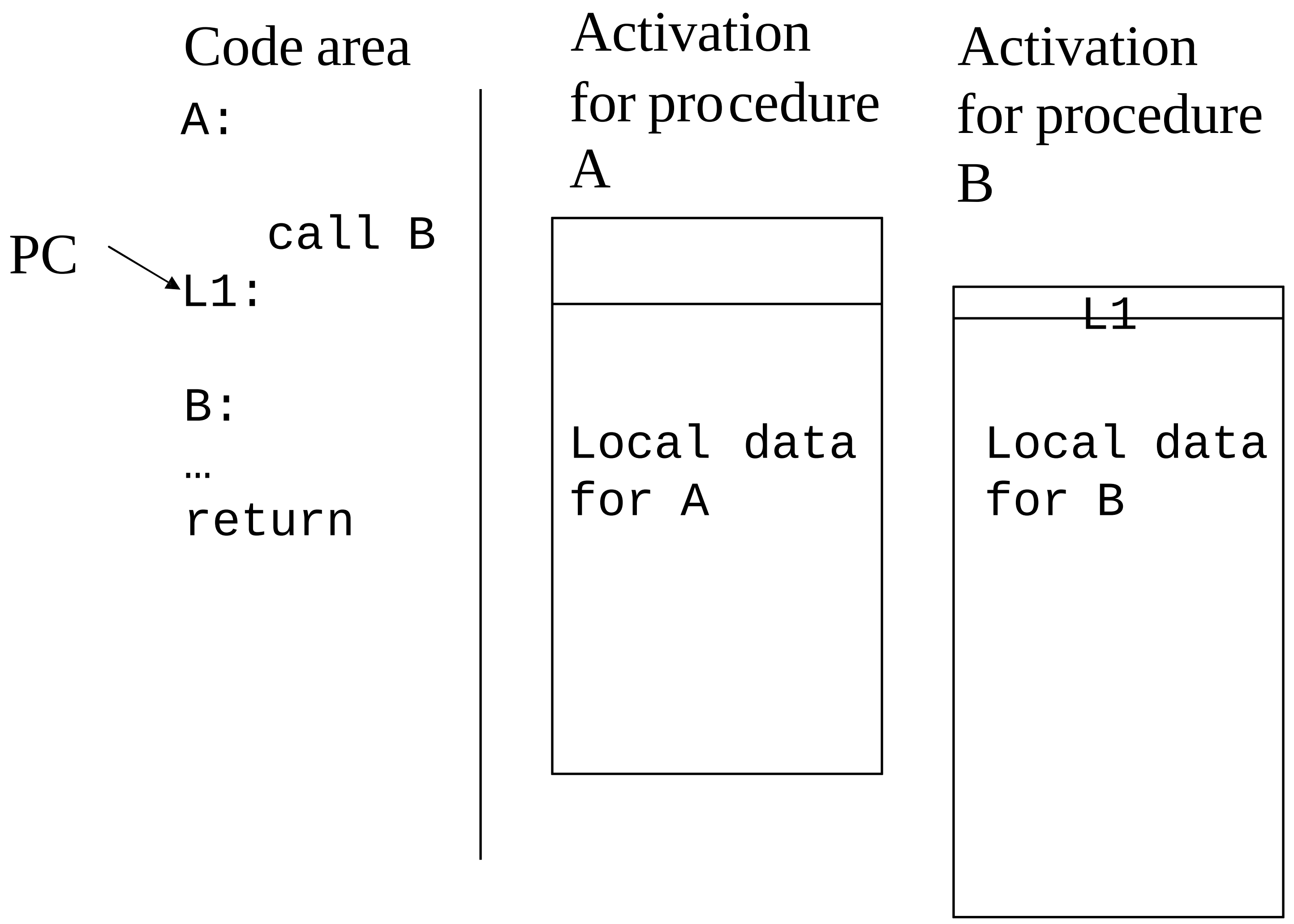

Code area
A:
PC
call B
L1:
B:
…
return
Activation for procedure A
Local data, parameters, return value, registers for A
Activation for procedure B
Local data, parameters, return value, registers for B

Static Allocation

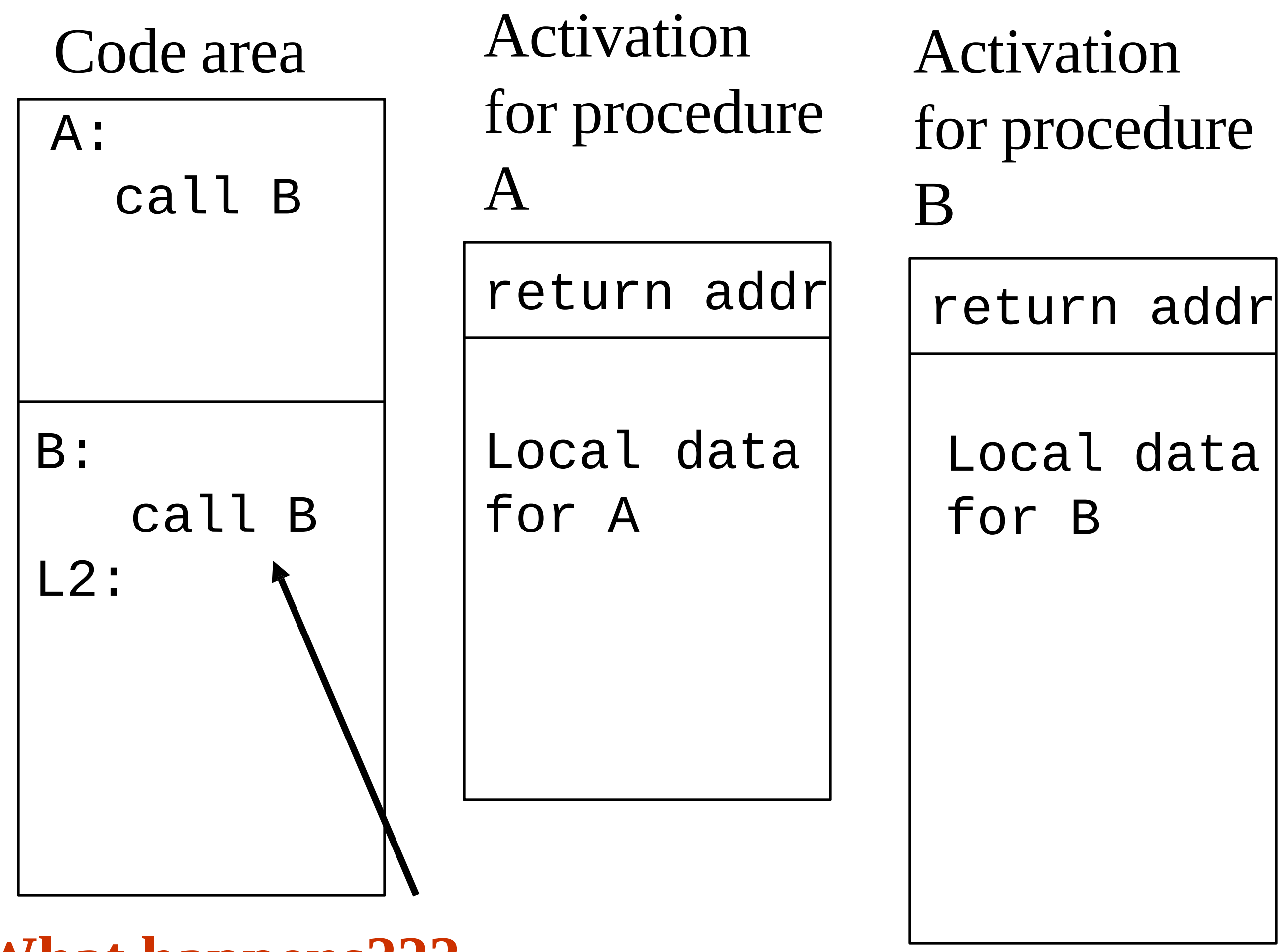


Static Allocation

Call tree:
A

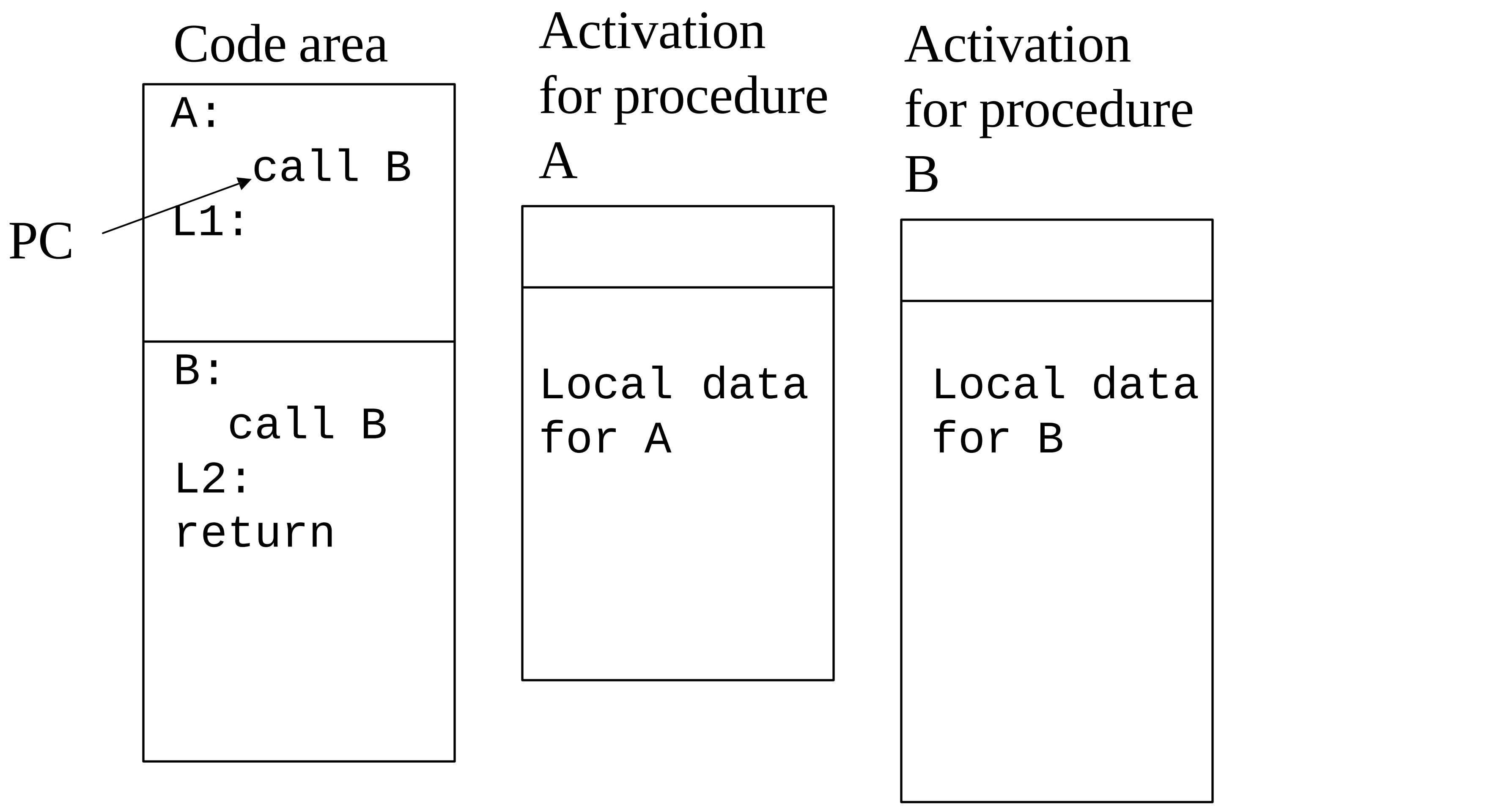


Static Allocation: Recursion?

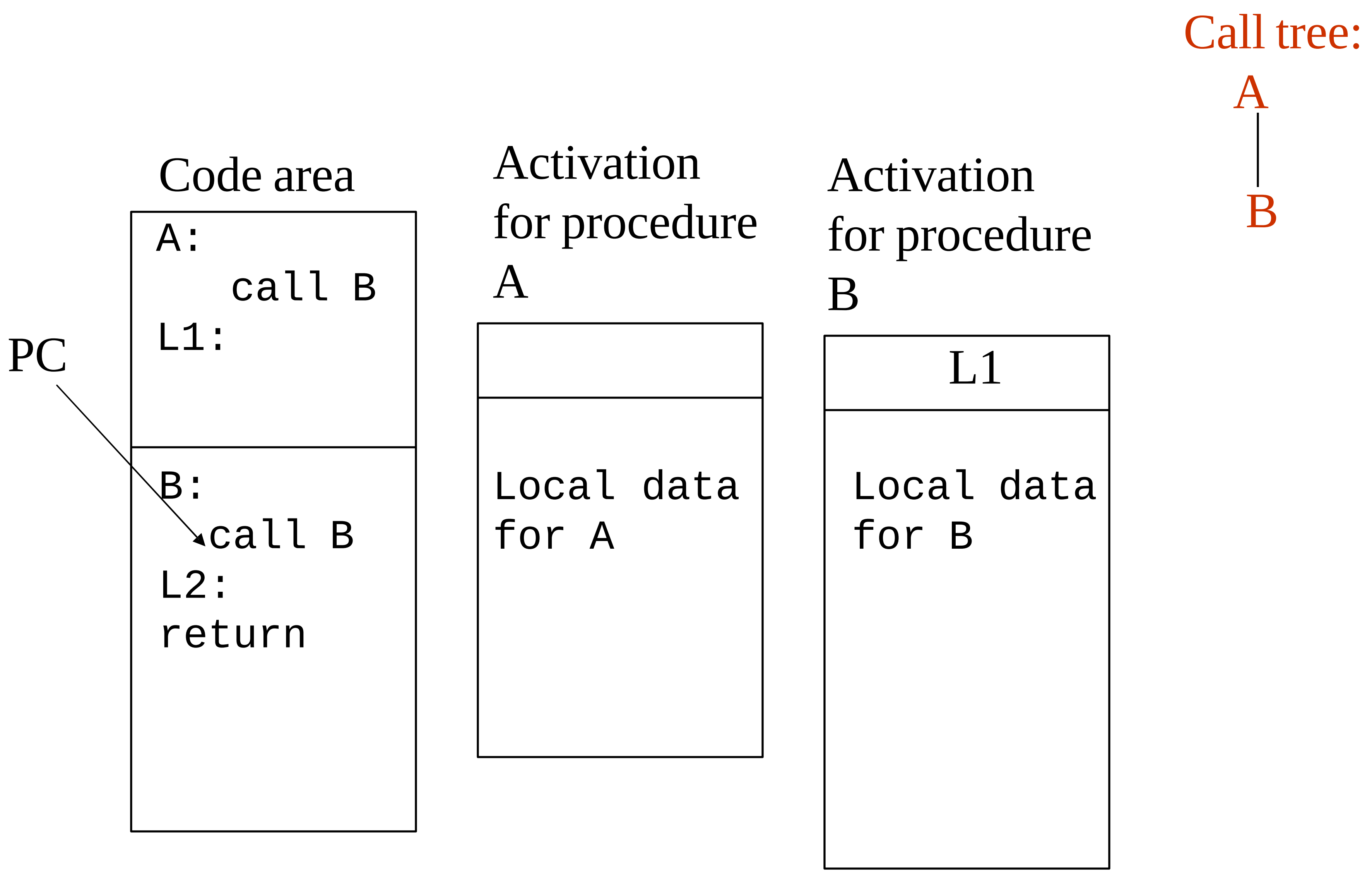


What happens???

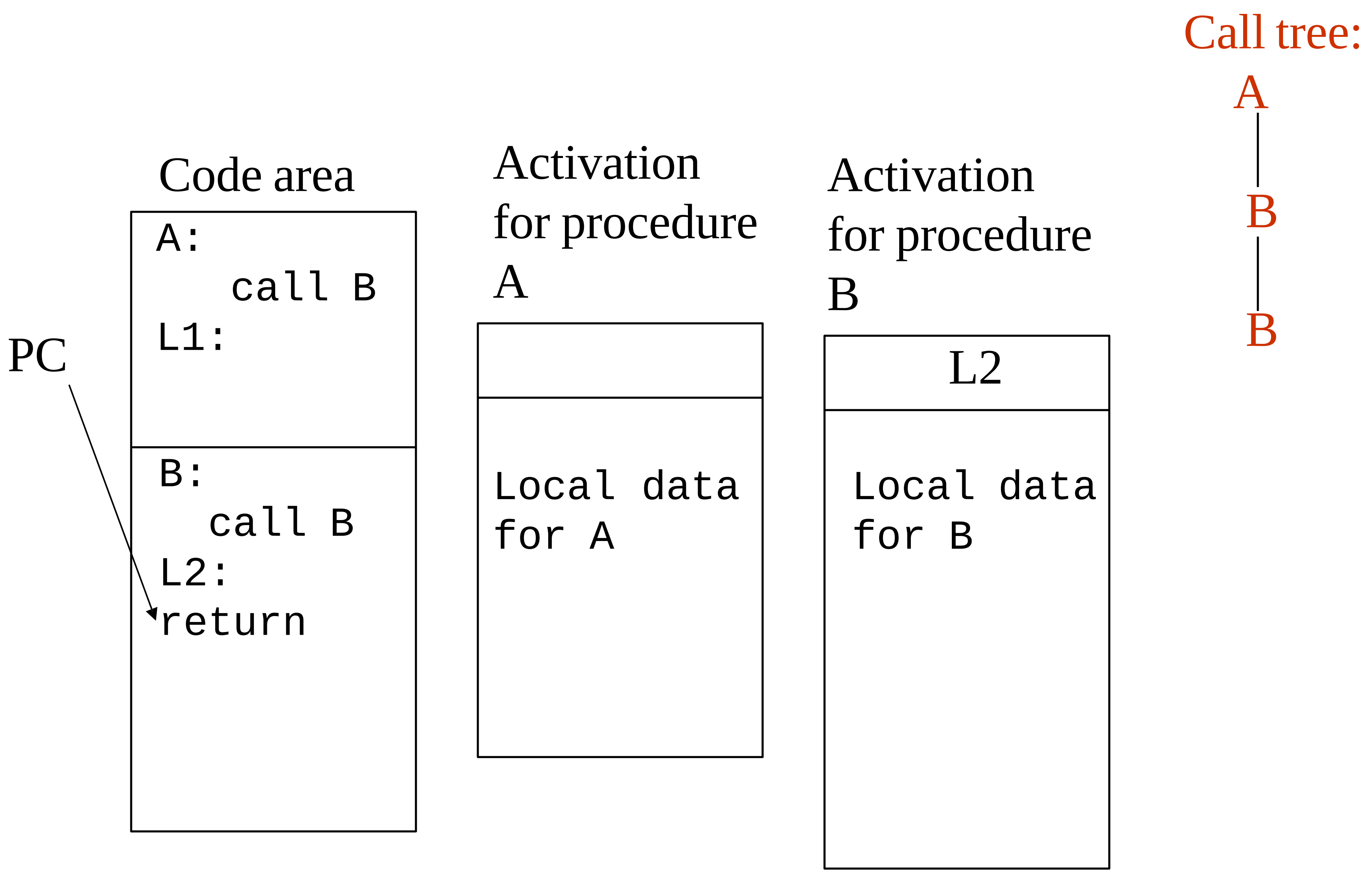
Static Allocation: Recursion



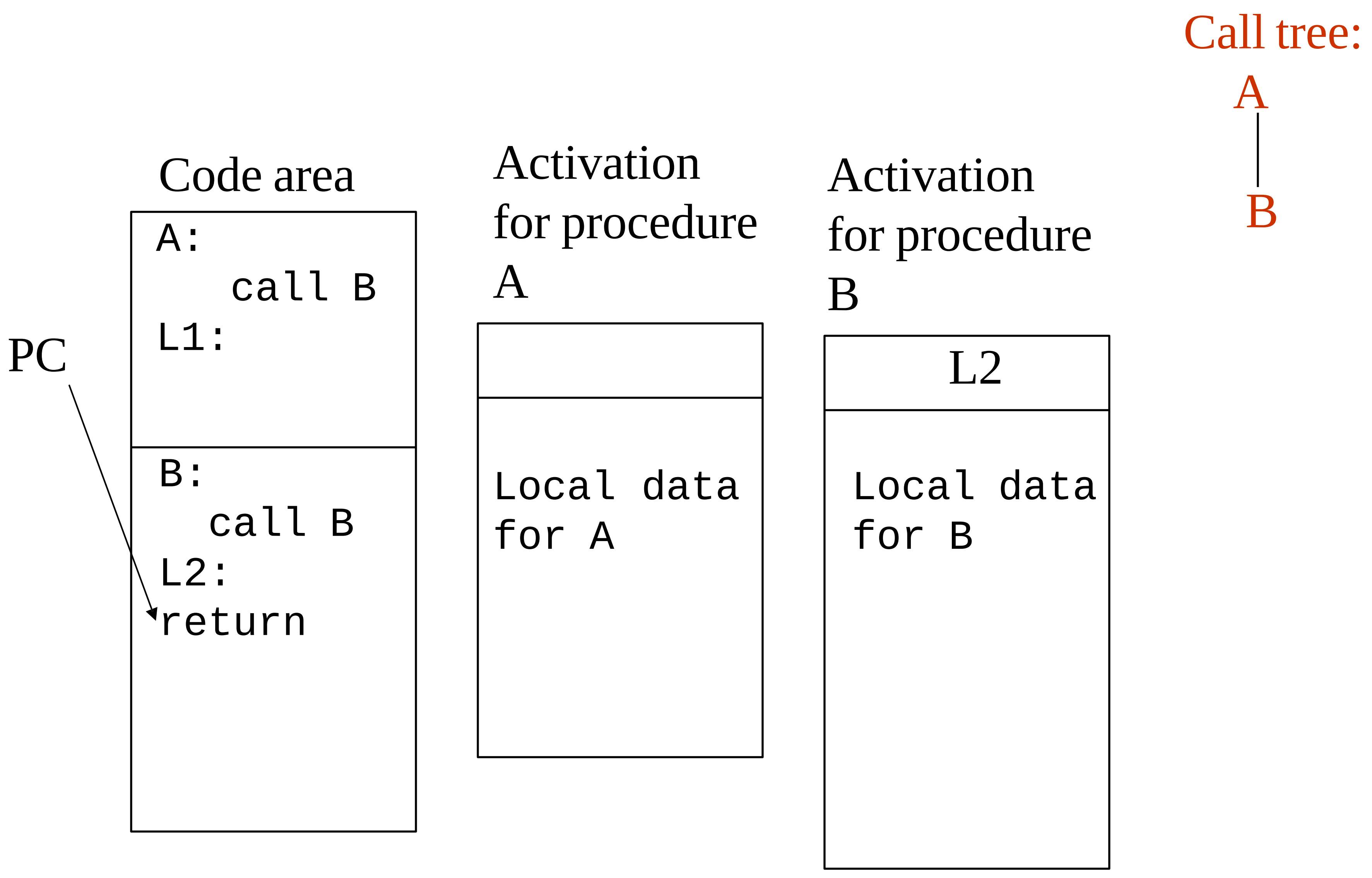
Static Allocation: Recursion



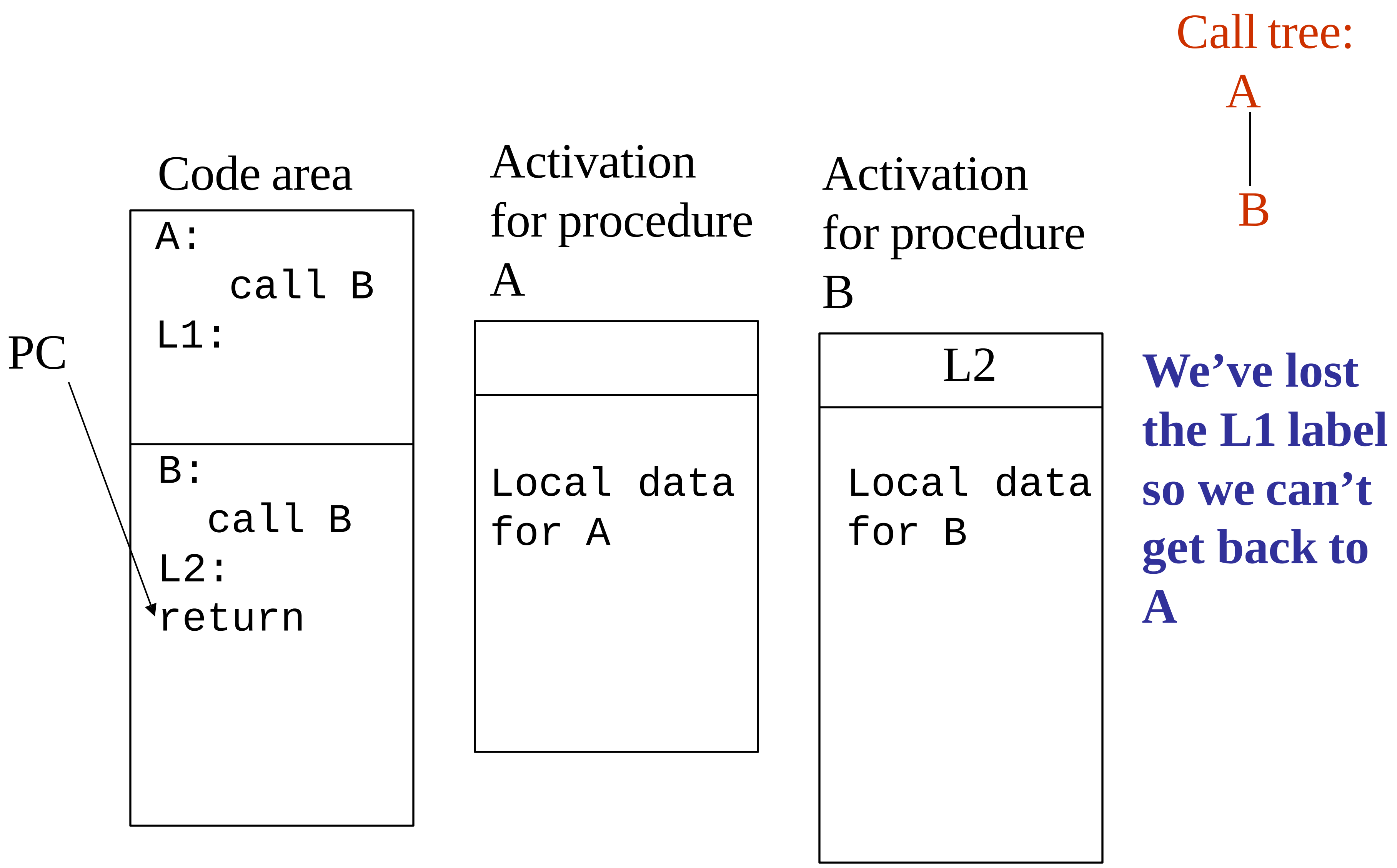
Static Allocation: Recursion



Static Allocation: Recursion



Static Allocation: Recursion



We've lost the L1 label so we can't get back to A

Runtime Addressing in Static Allocation

- Variable addresses hard-coded, usually as offset from data area where variable is declared.

  addr(x) = start of x's local scope + x's offset

Stack Allocation

Need a different approach to handle recursion.

- **Code area** – machine code for procedures
- **Static data** – often not associated with procedures
- **Stack (Control Stack)** – runtime information

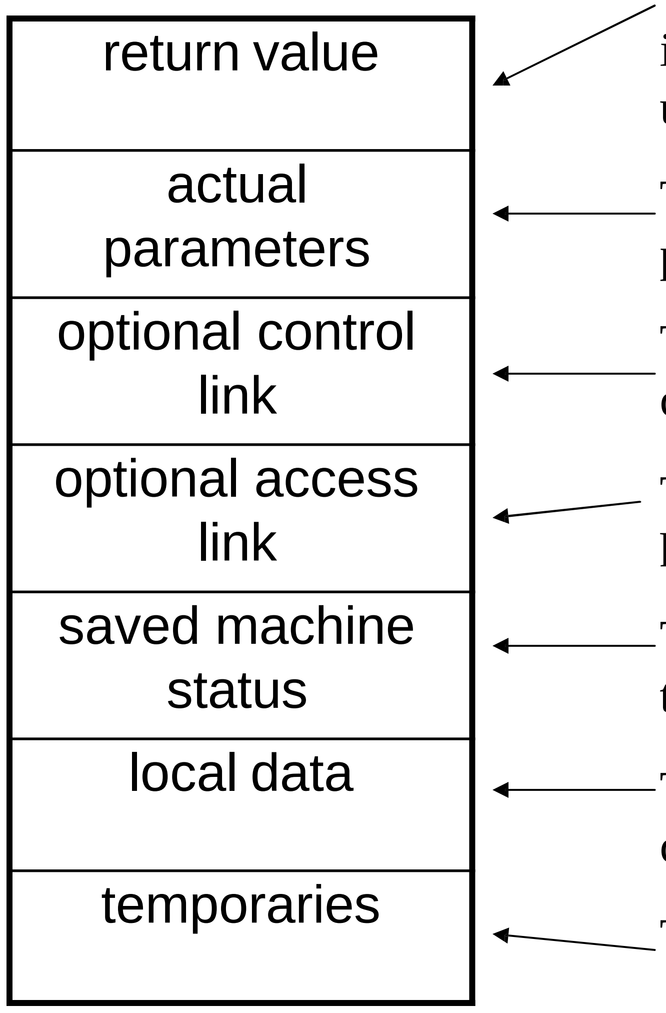
Control Stack

- The flow of the control in a program corresponds to a depth-first traversal of the activation tree that:
  - starts at the root,
  - visits a node before its children, and
  - recursively visits children at each node an a left-to-right order.
- A stack (called **control stack**) can be used to keep track of live procedure activations.
  - An activation record is pushed onto the control stack as the activation starts.
  - That activation record is popped when that activation ends.
  - Dynamic – grows and shrinks
- When node **n** is at the top of the control stack, the stack contains the nodes along the path from **n** to the root.

Activation Records

- Information needed by a single execution of a procedure is managed using a contiguous block of storage called **activation record**.
- An activation record is allocated when a procedure is entered, and it is de-allocated when that procedure exited.
- **Size of each field** can be determined at compile time (Although actual location of the activation record is determined at run-time).
  - Except that if the procedure has a local variable and its size depends on a parameter, its size is determined at the run time.

Activation Records (cont.)



The returned value of the called procedure is returned in this field to the calling procedure. In practice, we may use a machine register for the return value.

The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

The optional control link points to the activation record of the caller.

The optional access link is used to refer to nonlocal data held in other activation records.

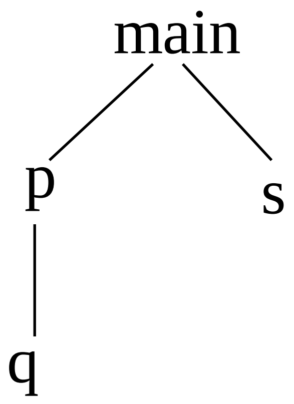
The field for saved machine status holds information about the state of the machine before the procedure is called.

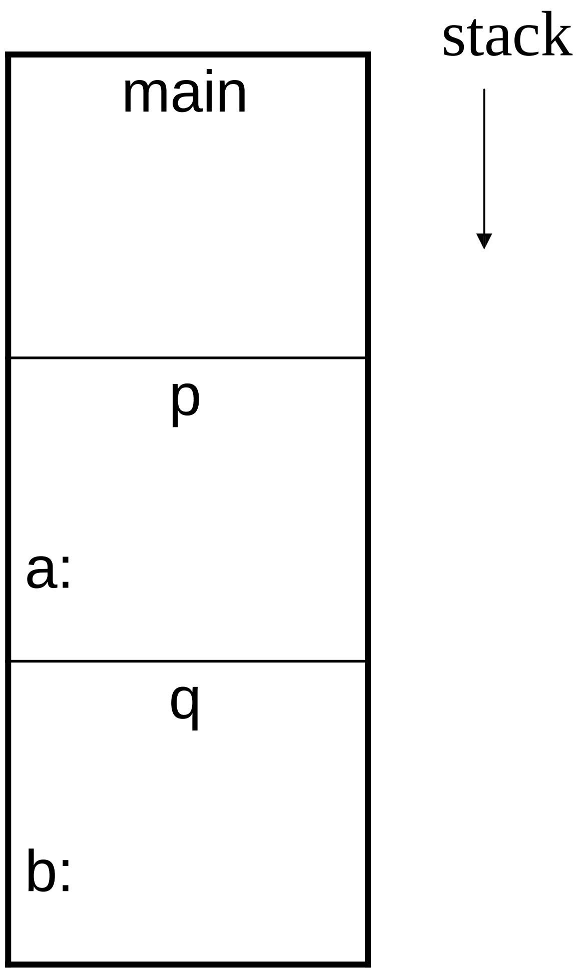
The field of local data holds data that local to an execution of a procedure..

Temporary variables is stored in the field of temporaries.

Activation Records (Ex1)

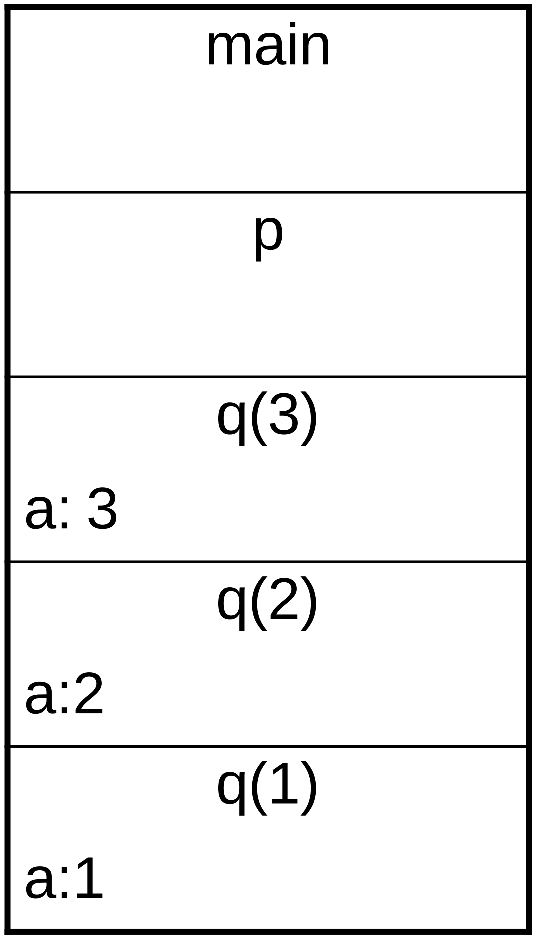
```
program main;
  procedure p;
    var a:real;
    procedure q;
      var b:integer;
      begin ... end;
    begin q; end;
  procedure s;
    var c:integer;
    begin ... end;
  begin p; s; end;
```



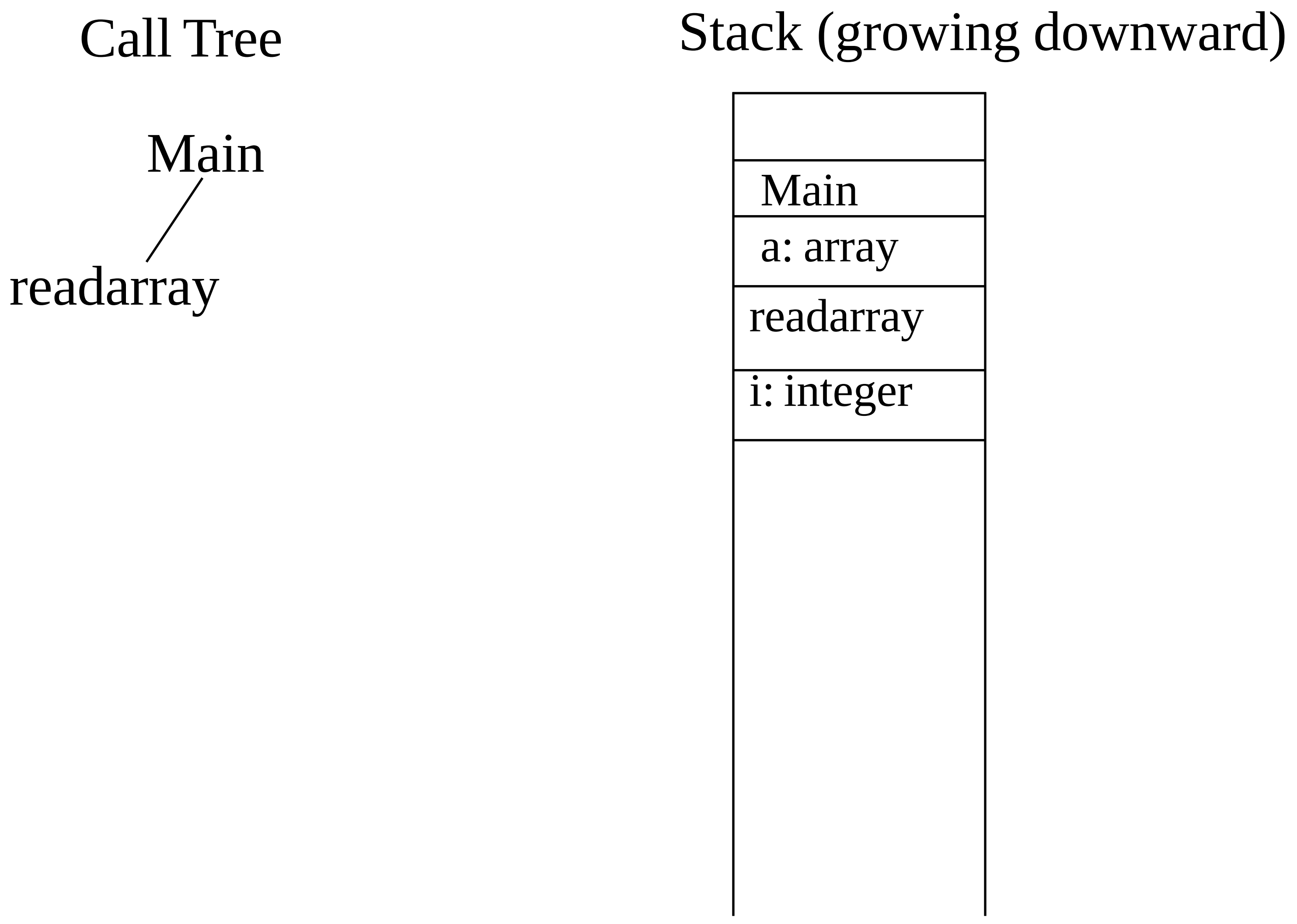


Activation Records for Recursive Procedures

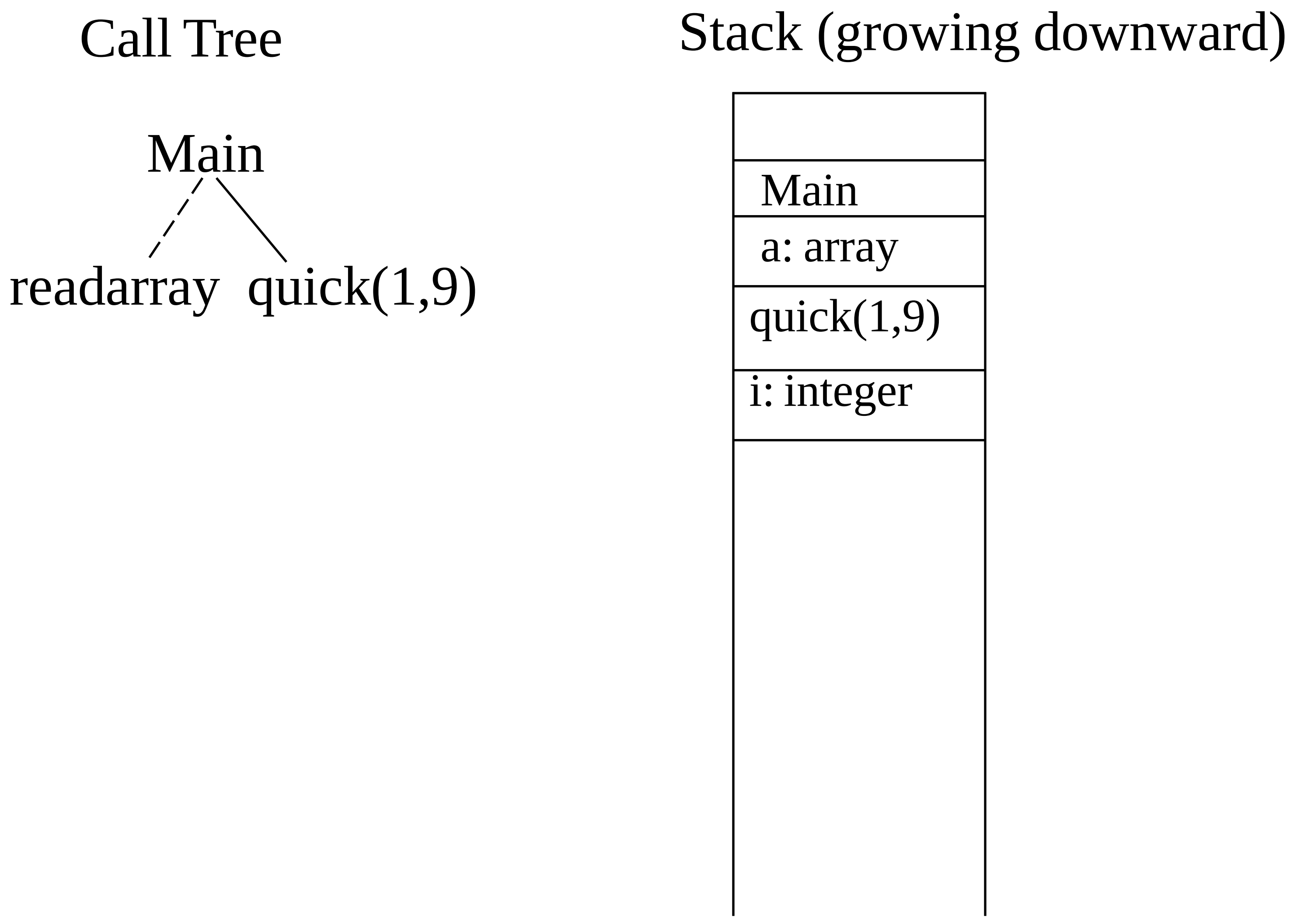
```
program main;
  procedure p;
    function q(a:integer):integer;
      begin
        if (a=1) then q:=1;
        else q:=a+q(a-1);
      end;
    begin q(3); end;
  begin p; end;
```



Stack Allocation for quicksort 1

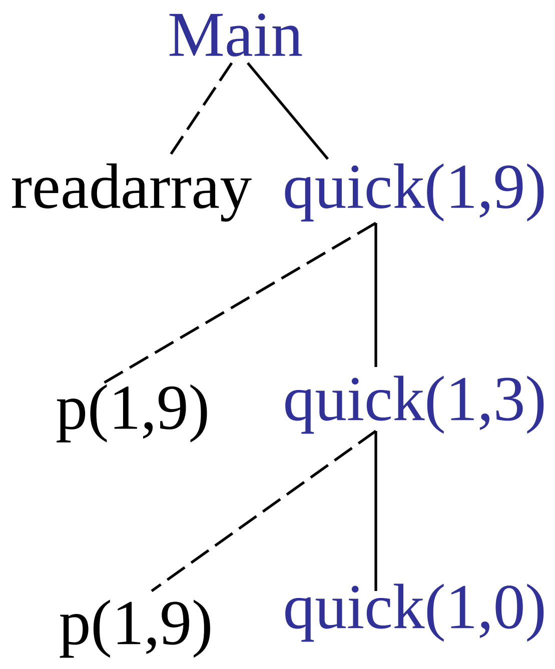


Stack Allocation for quicksort 2

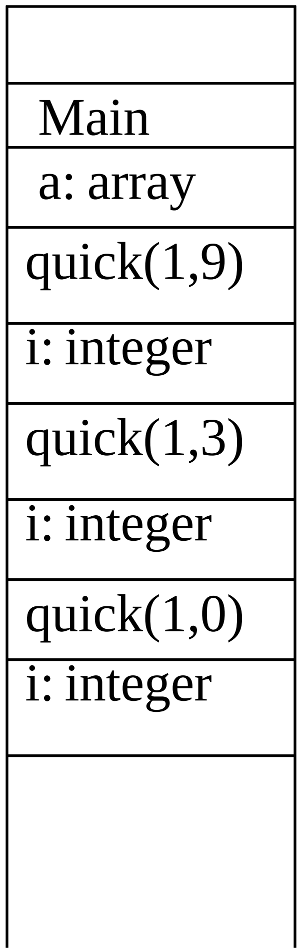


Stack Allocation for quicksort 3

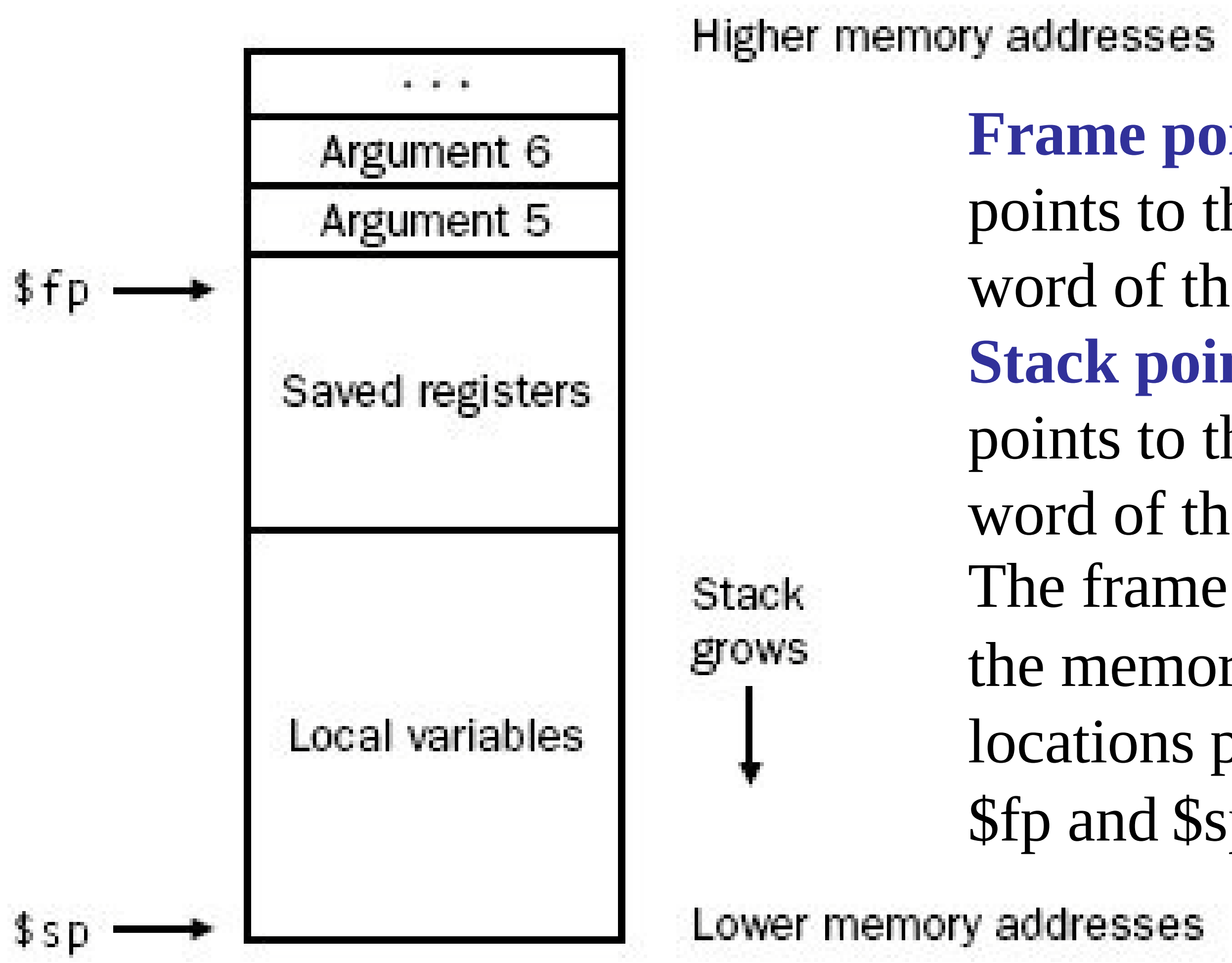
Call Tree



Stack (growing downward)



Layout of the stack frame



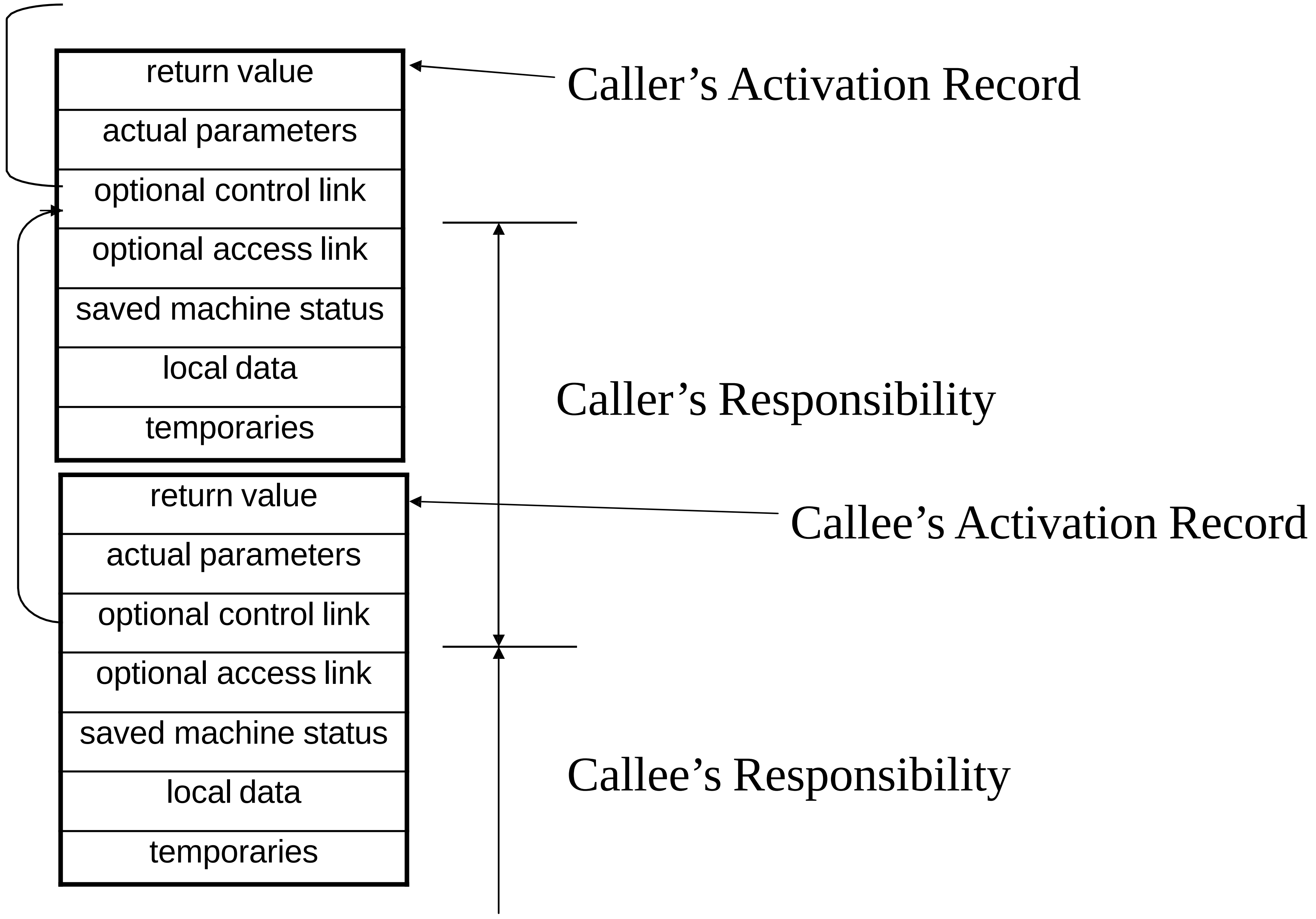
Frame pointer $fp: points to the first word of the frame, **Stack pointer ($sp):** points to the last word of the frame. The frame consists of the memory between locations pointed by $fp and $sp

Creation of An Activation Record

- Who allocates an activation record of a procedure?
  - Some part of the activation record of a procedure is created by that procedure immediately after that procedure is entered.
  - Some part is created by the caller of that procedure before that procedure is entered.

- Who deallocates?
  - Callee de-allocates the part allocated by Callee.
  - Caller de-allocates the part allocated by Caller.

Creation of An Activation Record (cont.)



Callee’s responsibilities before running

- **Allocate memory for the activation record/frame** by subtracting the frame’s size from $sp
- **Save callee-saved registers** in the frame. A callee must save the values in these registers ($s0–$s7, $fp, and $ra) before altering them

  [since the caller expects to find these registers unchanged after the call.]
  - Register $fp is saved by every procedure that allocates a new stack frame.
  - $ra only needs to be saved if the callee itself makes a call.
  - The other callee-saved registers that are used also must be saved.
- **Establish the frame pointer** by adding the stack frame’s size minus four to $sp and storing the sum in $fp.

Caller’s Responsibility

- **Pass arguments**: By convention, the first four arguments are passed in registers \$a0–\$a3.

  Any remaining arguments are pushed on the stack and appear at the beginning of the called procedure’s stack frame.

- **Save caller-saved registers**: The called procedure can use these registers (\$a0–\$a3 and \$t0–\$t9) without first saving their value. If the caller expects to use one of these registers after a call, it must save its value before the call.

- Execute a **jal** instructionwhich jumps to the callee’s first instruction and saves the return address in register **\$ra**.

Call Processing: Callee

- Initializes local data, calculate the offset of each variable from the start of the frame

- The executing procedure uses the frame pointer to quickly access values in its stack frame.

  For example, an argument in the stack frame can be loaded into register $v0 with the instruction

  ```
  lw $v0, 0($fp)
  ```

- Begins local execution

Callee’s responsibilities on returning

- If the callee is a function that returns a value, place the returned value in a special register e.g. $v0.
- Restore all callee-saved registers that were saved upon procedure entry.
- Pop the stack frame by adding the frame size to $sp.
- Return by jumping to the address in register $ra.

Added at the ‘return’ point(s) of the function

Return Processing: Caller

- Restore registers and status
- Copy the return value (if any) from activation
- Continue local execution

Added after the point of the call

Heap

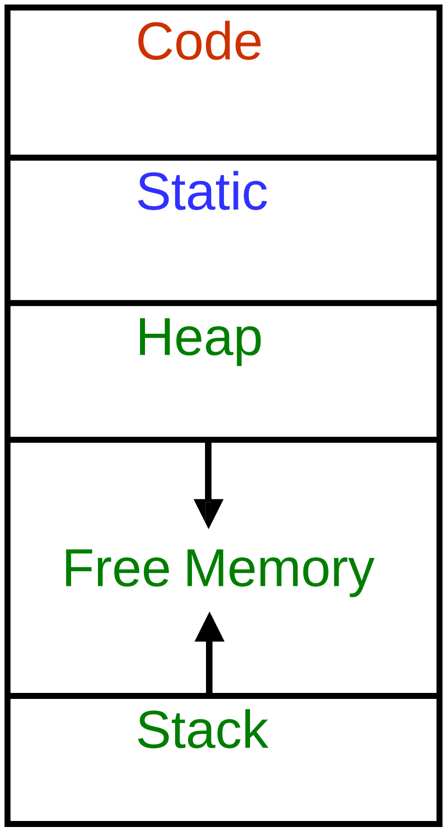
- Heap is used for data that lives indefinitely or until the program explicitly deletes it
- Local variables become inaccessible when their procedures end
- Many language enable us to create objects or other data whose existence is not tied to their procedure activations
  - In C++/Java we can create objects using new keyword and may be passed to other procedures

Heap Management

- Memory Manager
  - Allocates and deallocates space within the heap.
  - Serves as an interface between application programs and operating system
  - For some languages that need to deallocate space manually MM is responsible for implementing that.
- Garbage Collector
  - Responsible for finding spaces within the heap that are no longer used by the program and the deallocate them
  - GC is an important subsystem of the memory manager

Storage Organization

- Compiler deals with logical address space
- OS maps the logical addresses to physical addresses



Memory locations for code are determined at compile time. Usually placed in the low end of memory

Size of some program data are known at compile time – can be placed another statically determined area

Dynamic space areas – size changes during program execution.

- Heap
  - Grows towards higher address
  - Stores data allocated under program control
- Stack
  - Grows towards lower address
  - Stores activation records

Typical subdivision of run-time memory

Run time Addressing

- Given a variable reference in the code, how can we find the correct instance of that variable?
- Things are trickier – variables can live on the stack.
- Tied to issues of scope

Storage and access for locally declared data

- The executing procedure uses the frame pointer to quickly access values in its stack frame as the frame pointer points to the start of the frame.

- Then add the variable’s offset from the start of the frame. Calculating local data’s offset (to be stored in the symbol table)

- However problem arise for variable length data

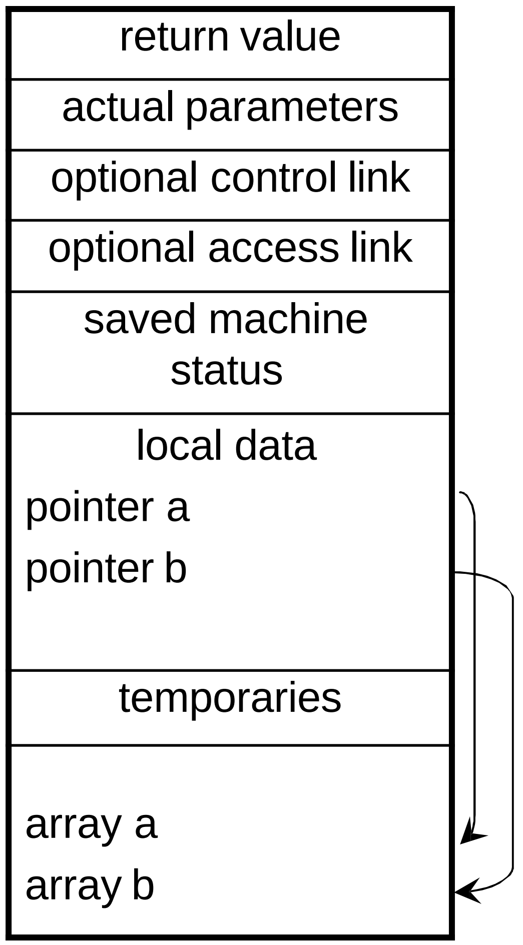
Variable-Length Local Variables

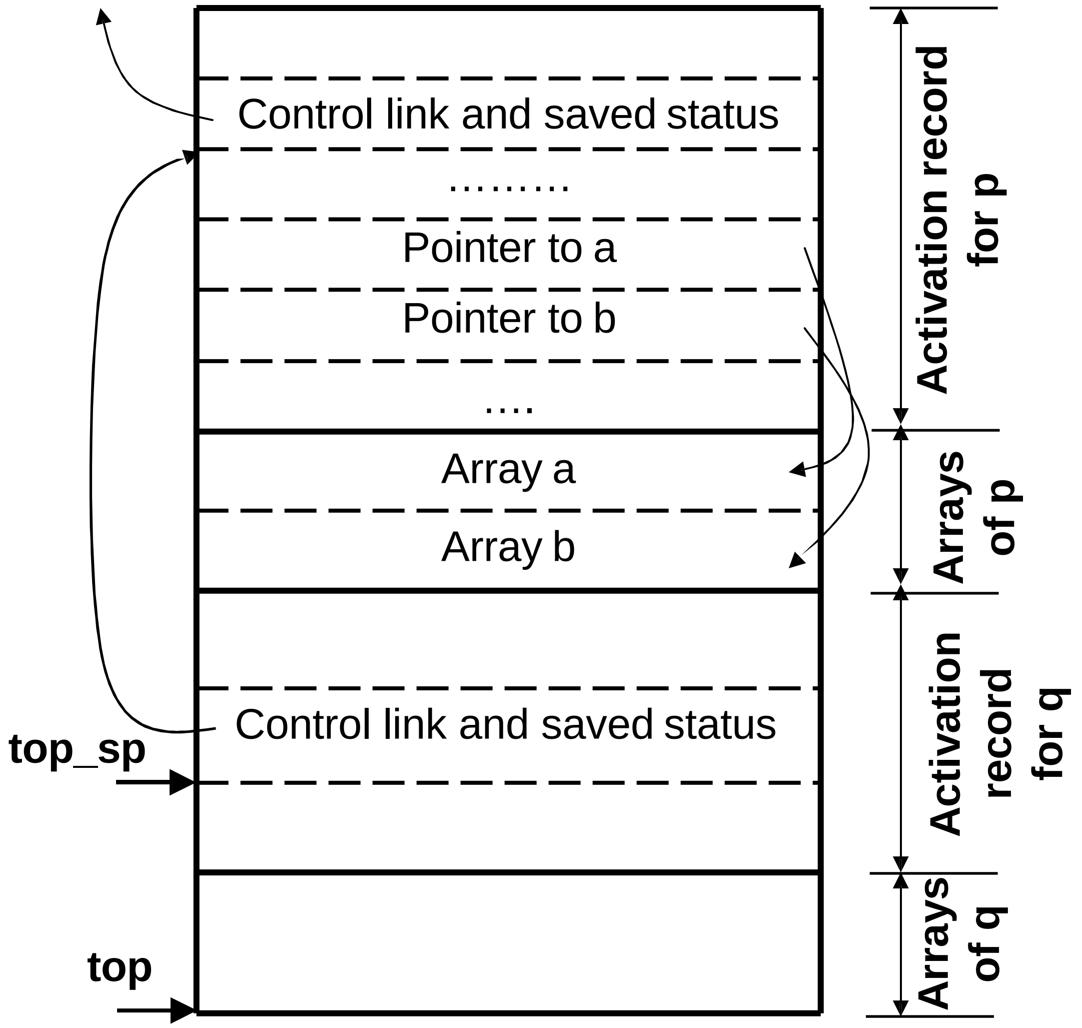
- **Goal:**
  Allow a routine to have variable-length data
  (i.e., dynamically-sized arrays) as local data in frame

- **Option 1:**
  Allocate the variable on the heap
  Work with pointers to the data
  Auto free the data when the routine returns

- **Option 2:**
  Create the variable on the stack, dynamically
  Effectively: Enlarge the frame as necessary
  Still need to work with pointers

Variable Length Data





Variable length data is allocated after temporaries, and there is a link to from local data to that array.

Data Access Without Nested Procedures

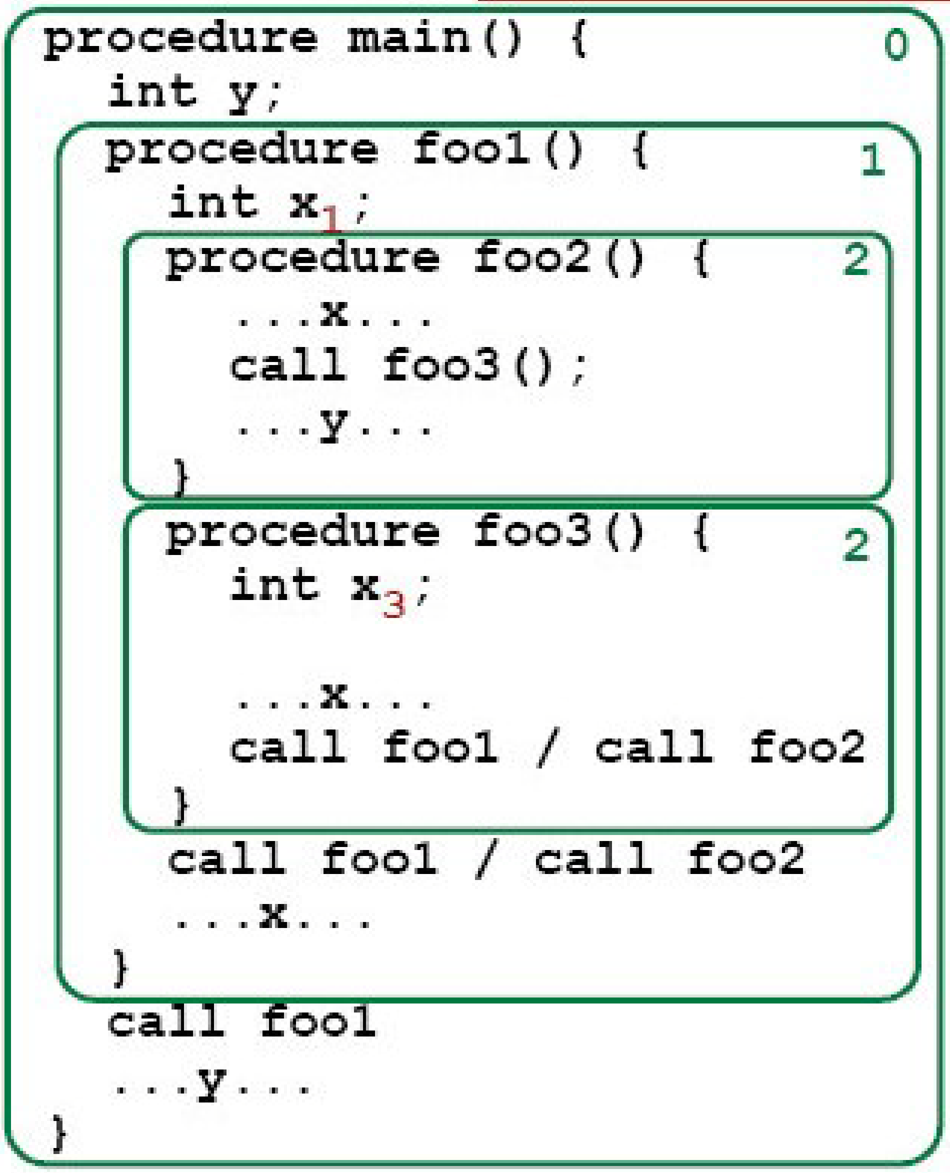
- Does not allow nested procedure declaration
  - Variables are either local or global
- Allocation and access to variables are simple
  - Global variables
    - Allocated static storage
    - Locations remain fixed and known at compile time
    - Access possible using statically determined address
  - Local variables may be accesses using *top_sp* pointer

Scope

The **scope** of a variable is that portion of the programs to which the variable applies.

- A variable is **local** to a procedure if the declaration occurs in that procedure.
- A variable is **non-local** to a procedure if it is not local to that procedure but the declaration occurs in an enclosing scope of that procedure.
- A variable is **global** if it occurs in the outermost scope.

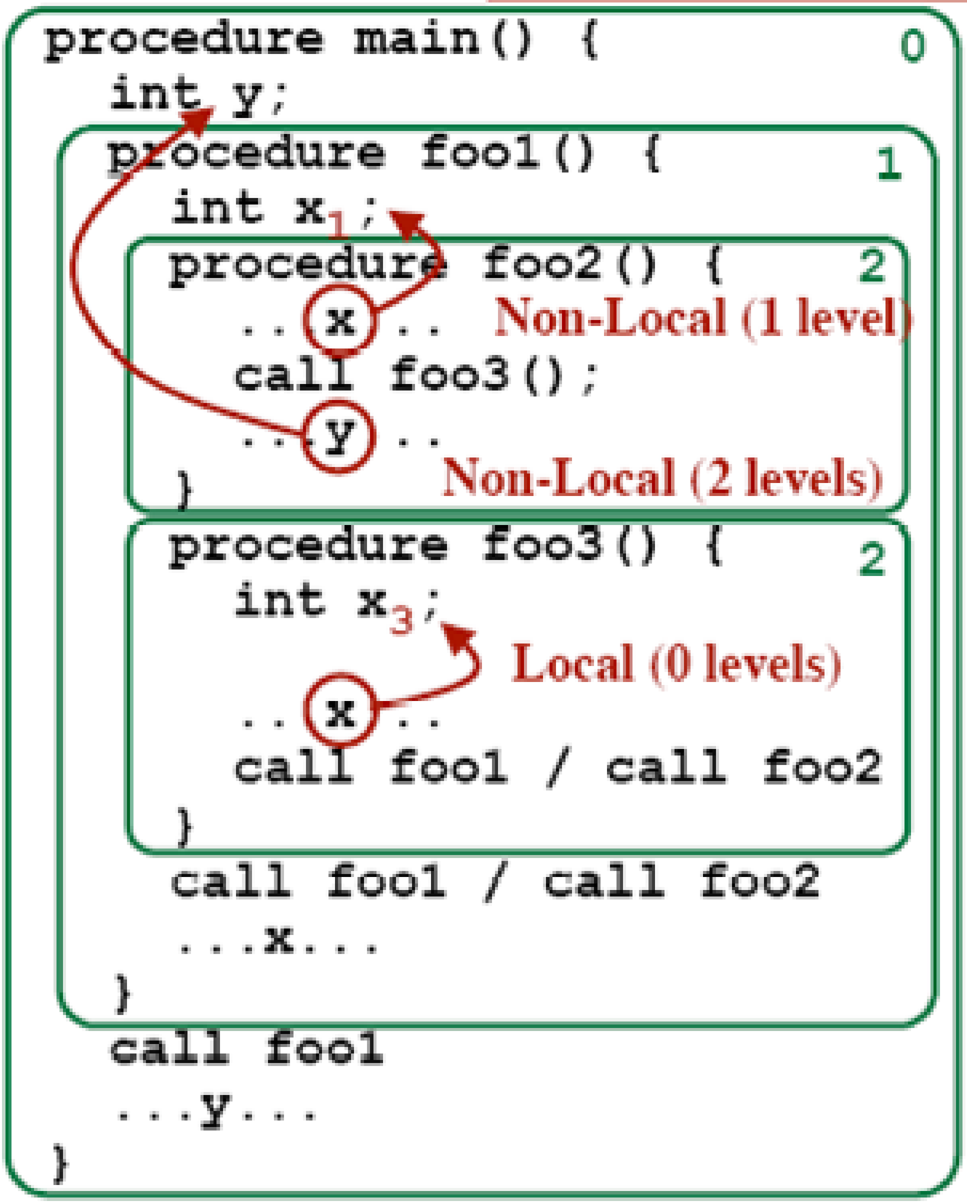
Local / Non-Local Variables



```
procedure main() {
   int y;
   procedure foo1() {
      int x_1;
      procedure foo2() {
         ...x...
         call foo3();
         ...y...
      }
      procedure foo3() {
         int x_3;

         ...x...
         call foo1 / call foo2
      }
      call foo1 / call foo2
      ...x...
   }
   call foo1
   ...y...
}
```

Local / Non-Local Variables



```
procedure main() {
  int y;
  procedure foo1() {
    int x1;
    procedure foo2() {
      ..x..
      call foo3();
      ..y..
    }
    procedure foo3() {
      int x3;

      ..x..
      call foo1 / call foo2
    }
    call foo1 / call foo2
    ...x...
  }
  call foo1
  ...y...
}
```

Parameter Passing

Various approaches to passing data into and out of a procedure via parameters

1. **Call-by-value** – data is copied at the callee into activation and any item changes do not affect values in the caller.
   - Ex: C parameters

2. **Call-by-reference** – pointer to data is placed in the callee activation and any changes made by the callee are indirect references to the actual value in the caller.
   - C++ - call-by-value and call-by-reference

Call-by-value vs Call-by-reference

```
var a,b : integer
   procedure swap(x,y : integer);
   var t: integer;
   begin  t := x; x := y; y := t;  end;
begin
a  :=  1;b := 2;
swap(a,b);
write ('a = ',a);
write ('b = ',b);
end.
```

	value	reference
write(a)	1	2
write(b)	2	1

Parameter Passing

3. **Call-by-value-result (copy-restore)** – hybrid of call-by-value and call-by-reference. Data copied at the callee. During the call, changes do not affect the actual parameter. After the call, the actual value is updated.

4. **Call-by-name** – the actual parameter is in-line substituted into the called procedure. This means it is not evaluated until it is used.

Call-by-value-result vs. Call-by-reference

```
var a: integer
    procedure foo(x: integer);
    begin  a := a + 1; x := x + 1;  end;
begin
a := 1;
foo(a);
write ('a = ',a);
end.
```

	Value-result	reference
write(a)	2	3

Call-by-name vs. Call-by-reference

```
var a : array of integer; i: integer
    procedure swap(x,y : integer);
    var t: integer;
    begin  t := x; x := y; y := t;  end;
begin
i := 1;
swap(i,a[i]);
write (‘i,a[i] = ’,i,a[i]);
end.
```

Memory Manager

- Keeps track of all the free space in heap storage at all times.
- Two basic functions
  - Allocation –
    - In response to some request MM produces a chunk of contiguous heap memory
    - If not possible then seeks to increase heap storage from virtual memory
  - Deallocation –
    - MM returns deallocated space to the pool of free space

Memory Manager

- MM would be simpler if
  - All requests were for the same sized chunks
  - Storage were released in first-allocated first-deallocated style

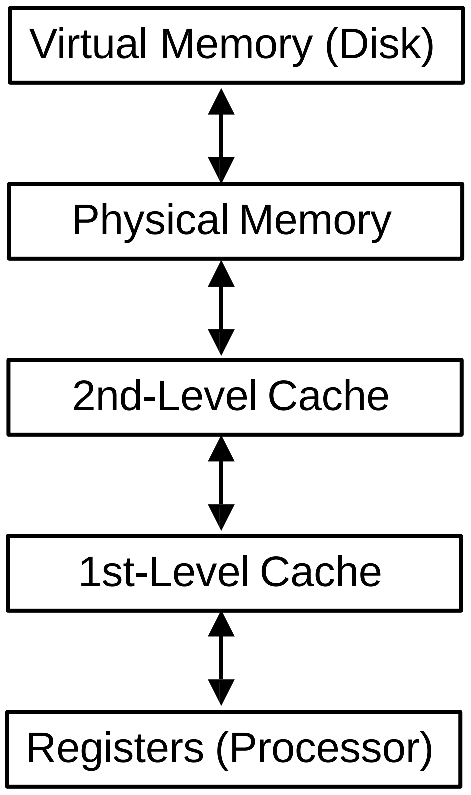
- BUT!! None is possible in most languages

- MM should be prepared to
  - Serve requests in any order
  - Serve requests of any size

Properties of Memory Manager

- Desired properties of MM
  - Space Efficiency
    - Should minimize the total heap spaced needed
    - Can be achieved by minimizing fragmentation
  - Program Efficiency
    - Should make good use of the memory to allow program to run faster
    - Should take advantage of ‘locality’ characteristics of program
  - Low Overhead
    - Allocation and deallocation operations should be as efficient as possible

Memory Hierarchy



Typical Sizes		Typical Access Times
> 2 GB	Virtual Memory (Disk)	3-15 ms
256MB-2GB	Physical Memory	100-150ns
128KB-4MB	2nd-Level Cache	40-60ns
16KB-64KB	1st-Level Cache	5-10ns
32 Words	Registers (Processor)	1ns

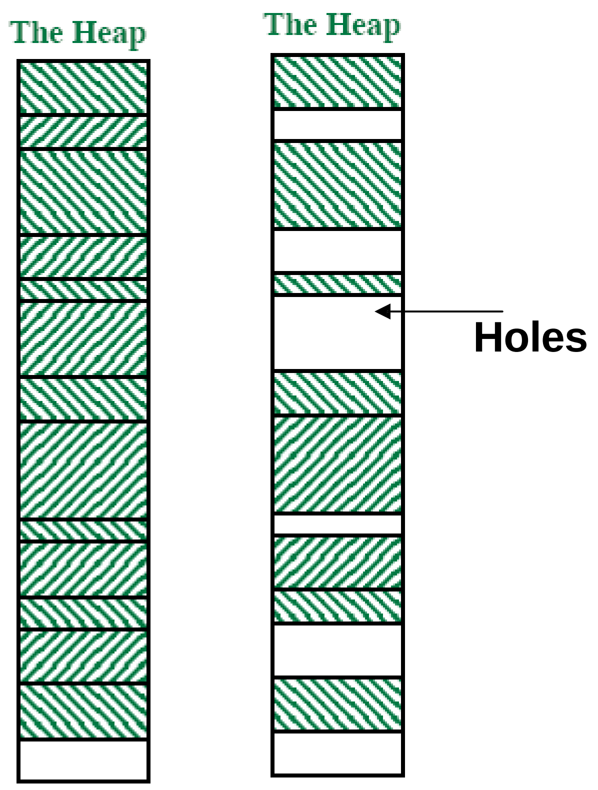
- **Registers are managed by the generated code**
- **All other levels are managed automatically**
- **With each memory access the machine searches each level of the memory in succession starting from the lowest level**

Fragmentation

- Variable-sized chunks of memory are allocated.
- Some chunks are freed (in more-or-less random order).
- The resulting free space become "fragmented".
- Need to allocate more space?
- Adequate space is available
  ... but it is not contiguous!

- Even if holes are larger than the request
  …. We need to split the hole

  …. Creating yet smaller hole



Fragmentation Reduction

- Control how the MM places new objects in the heap
- Best-Fit Algorithm
  - Allocate the requested memory in the smallest available hole that is large enough
  - Spares larger holes for subsequent larger requests
  - Good strategy for real-life programs
- First-Fit Algorithm
  - Object is placed in the first hole in which it fits
  - Takes less time to place objects
  - Overall performance is inferior to best-fit

Problems with Manual Deallocation

- Two common mistakes
  1. Memory Leak
     - Failing ever to delete data that will never be referenced
     - Slow down the program due to increased memory usage
     - Critical for long running/nonstop program such as OS or server
     - Does not affect program correctness
     - Automatic garbage collection gets rid of it
       - Even the program may use more memory than necessary

Problems with Manual Deallocation

- **Two common mistakes**

2. Dangling Pointer Reference
   - Delete some storage and try to refer to the data in the deallocated storage
   - Once the freed storage is reallocated any read, write or deallocation via dangling pointer can produce random effects
   - Dereferencing a dangling pointer creates program error hard to debug

Thank You