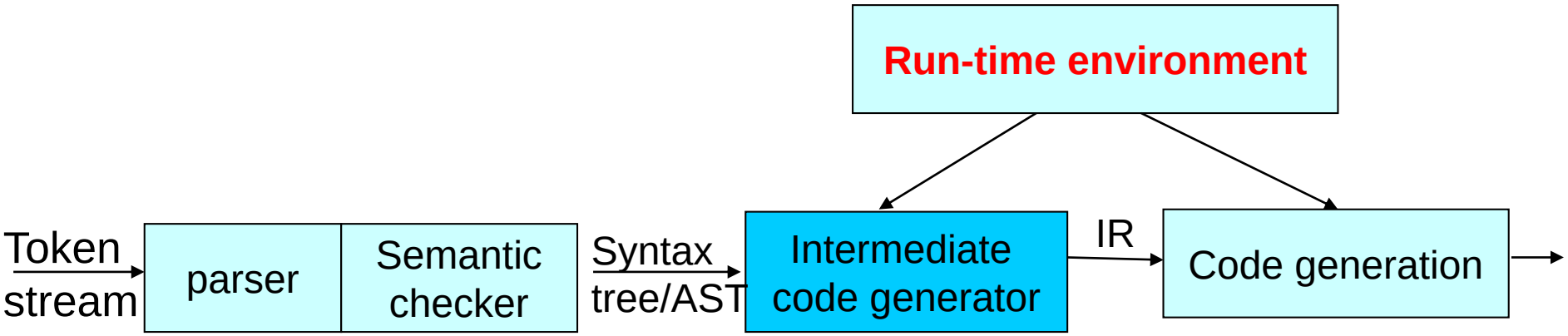
Run-Time Environments



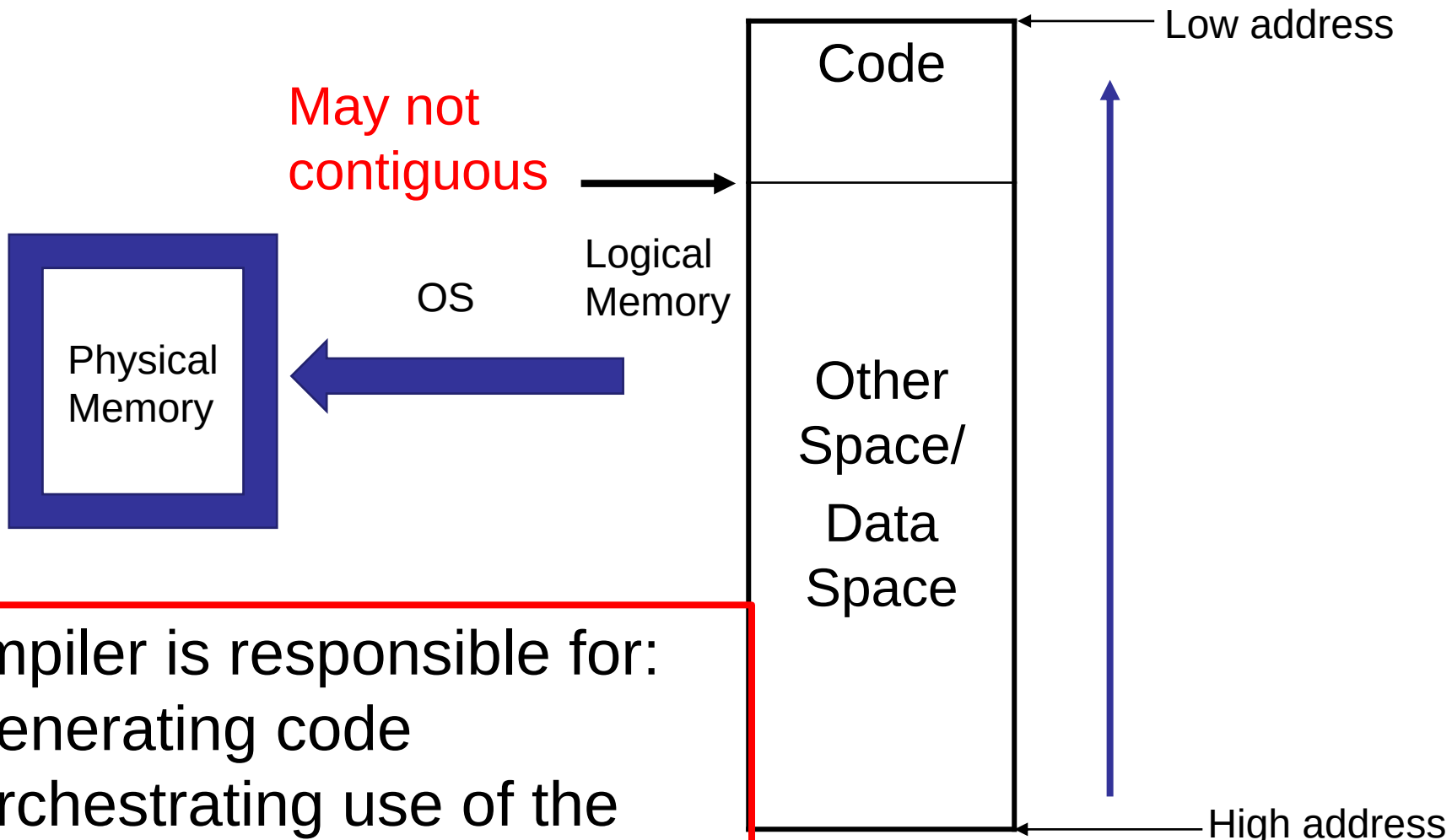
Run-Time Environments

- Management of run-time resources
- How to allocate the space for the generated target code and the data objects?
- Determined at compile time => *allocated statically*.
- Otherwise data objects => *allocated at run-time*.
- *Run-time support package :* allocation and de-allocation, e.g., malloc/free, new/delete
  - ✓ Loaded together with the generated target code
  - ✓ Depends on the semantics of the programming language

Run-time Resources

- Execution of a program is initially under the control of the operating system (loader)
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)

Memory Layout



Compiler is responsible for:

- Generating code
- Orchestrating use of the data area

Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order

2. When a procedure is called, control eventually returns to the point immediately after the call

Do these assumptions always hold?

Procedure Activations

- An **activation** is an invocation of a procedure starts at the beginning of the procedure body.
- The **lifetime** of an activation of a procedure is the sequence of the steps between the first and the last steps in the execution of that procedure (including the other procedures called by that procedure).
- 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, i.e., nested activation
- The lifetime of a variable x is the portion of execution in which x is defined
  - Lifetime is dynamic concept
  - Scope is a static concept

Activation Trees

- An activation of a function is a particular invocation of that function
- Each activation will have particular values for the function parameters
- Each activation can call another activation before it becomes inactive
- The sequence of function calls can be represented as an ***activation tree***

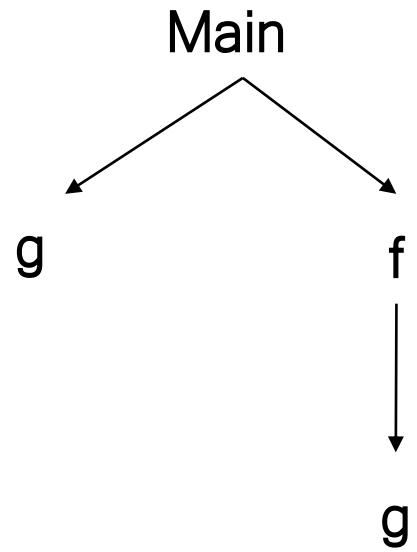
Activation Tree (cont.)

- 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 the node b iff the lifetime of a occurs before the lifetime of b.

Activation Tree Example

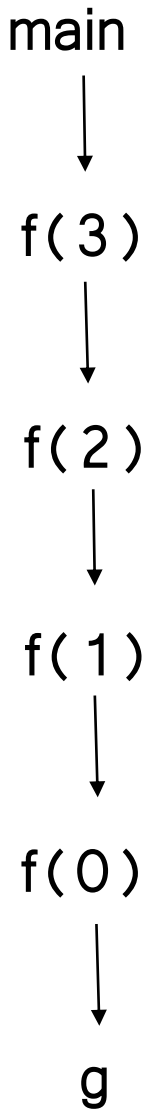
```
Class  Main  {
        g( )  :  Int  {  1  };
        f( ):  Int  {  g( )  };
        main( ):  Int  {{  g( );  f( );  }};
}
```



Activation Tree Example

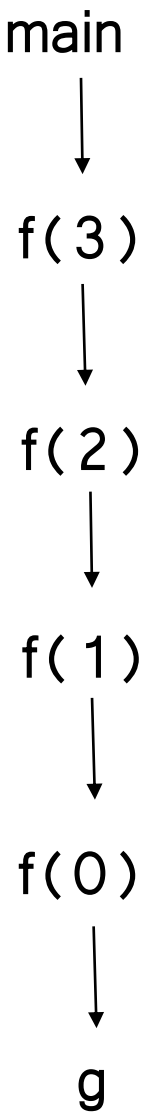
```
Class Main {
    g() : Int { 1 };
    f(x:Int): Int {
        if x = 0 then g()
            else f(x - 1)
        fi
    };

    main(): Int { {f(3); }};
}
```

What is the activation tree for this example?



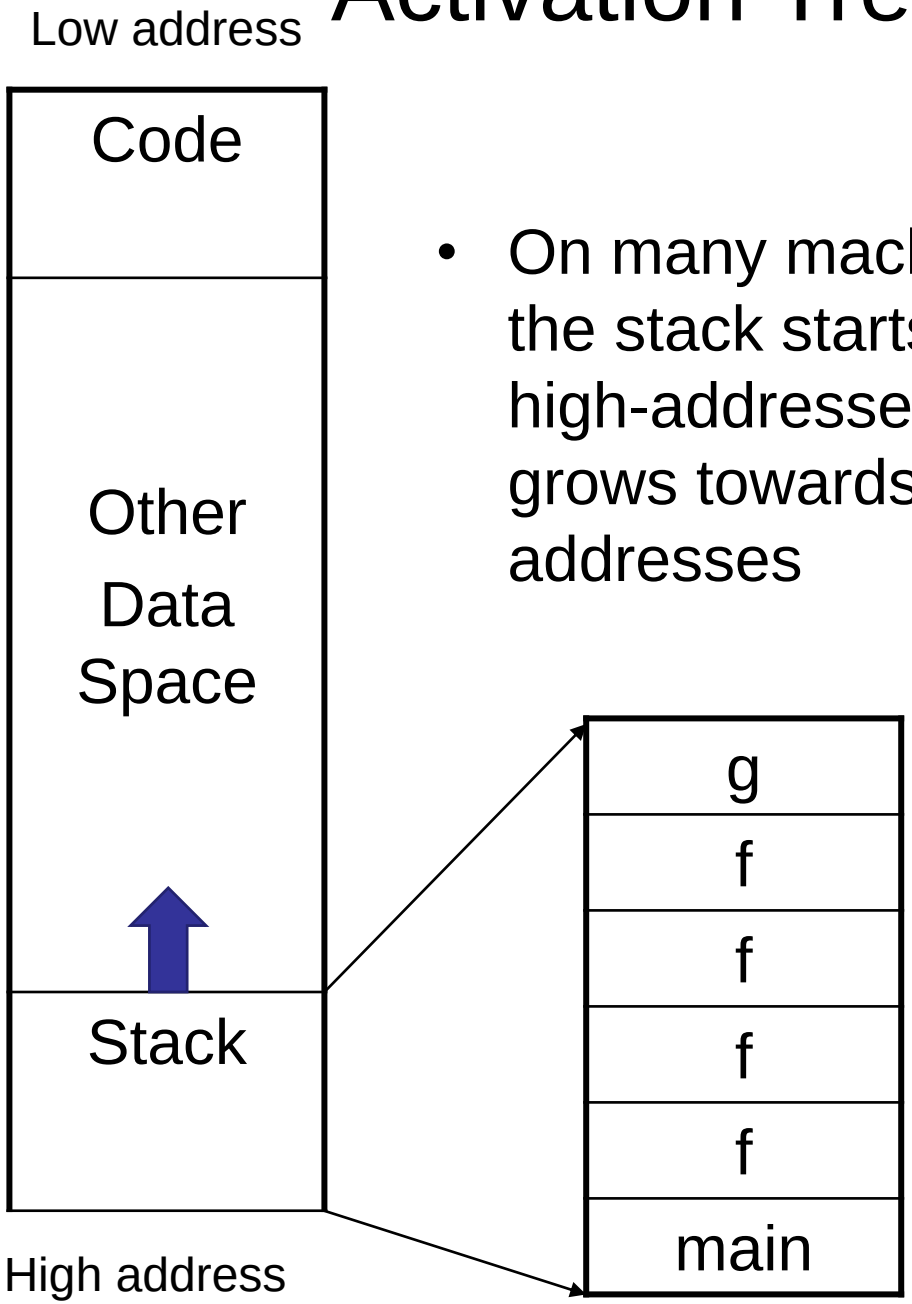
Activation Tree Example

```
Class Main {
    g() : Int { 1 };
    f(x:Int): Int {
        if x = 0 then g()
            else f(x - 1)
        fi
    };
    main(): Int { {f(3); }};
}
```

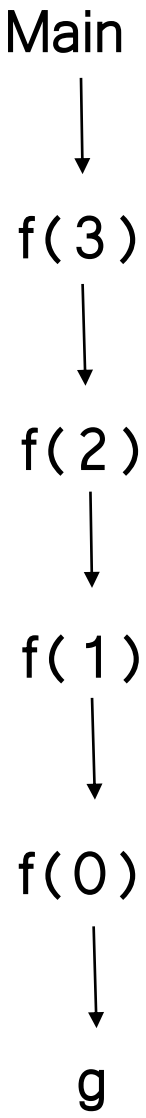


- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

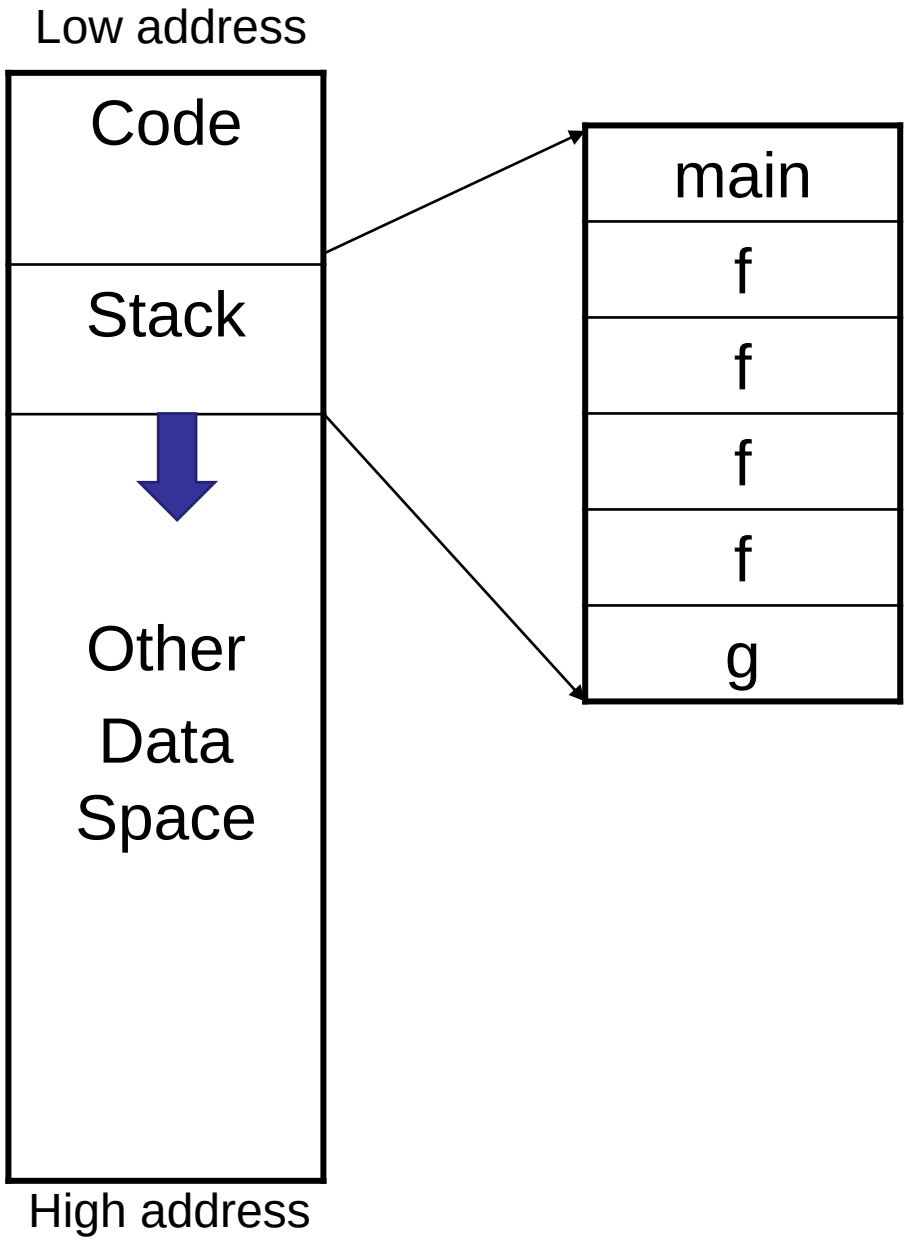
Activation Tree Example

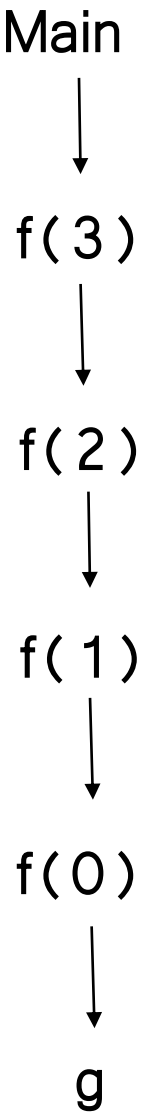


- On many machines the stack starts at high-addresses and grows towards lower addresses



Activation Tree Example



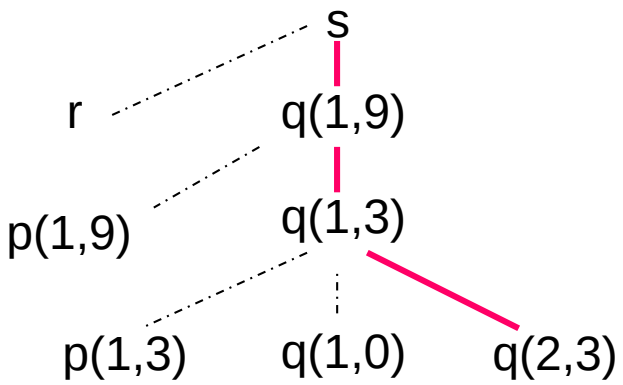


Stack and Activation Tree

- 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 in 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.

Cont.

- Control stack keeps track of live procedure activation.
- The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.



The control stack contains nodes along a path to the root.

Variable Scopes

- The same variable name can be used in the different parts of the program.
- The scope rules of the language determine which declaration of a name applies when the name appears in the program.
- An occurrence of a variable (a name) is:
  - **local**: If that occurrence is in the same procedure in which that name is declared.
  - **non-local**: Otherwise (i.e. it is declared outside of that procedure)

```
procedure foo;
  var b:real;
  procedure bar;
    var a: integer;                    a is local
    begin a := 1;  b := 2; end;        b is non-local
  begin ... end;
```

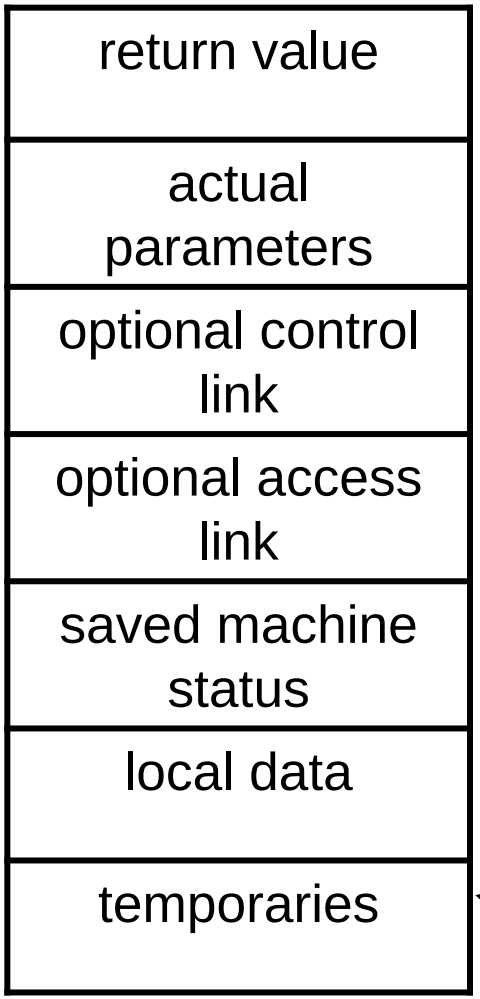
Activation Records/Frame

- Information for a single execution of a function is called an ***activation record*** or ***procedure call frame***
- A frame contains:
  - Temporary local register values for caller
  - Local data
  - Snapshot of machine state (important registers, PC)
  - Return address
  - Link to global data (access link)
  - Parameters passed to function (actual parameters)
  - Return value for the caller

The Main Point

- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record
- Thus, the AR layout and the code generator must be designed together!

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 callee to supply parameters to the callee.

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.

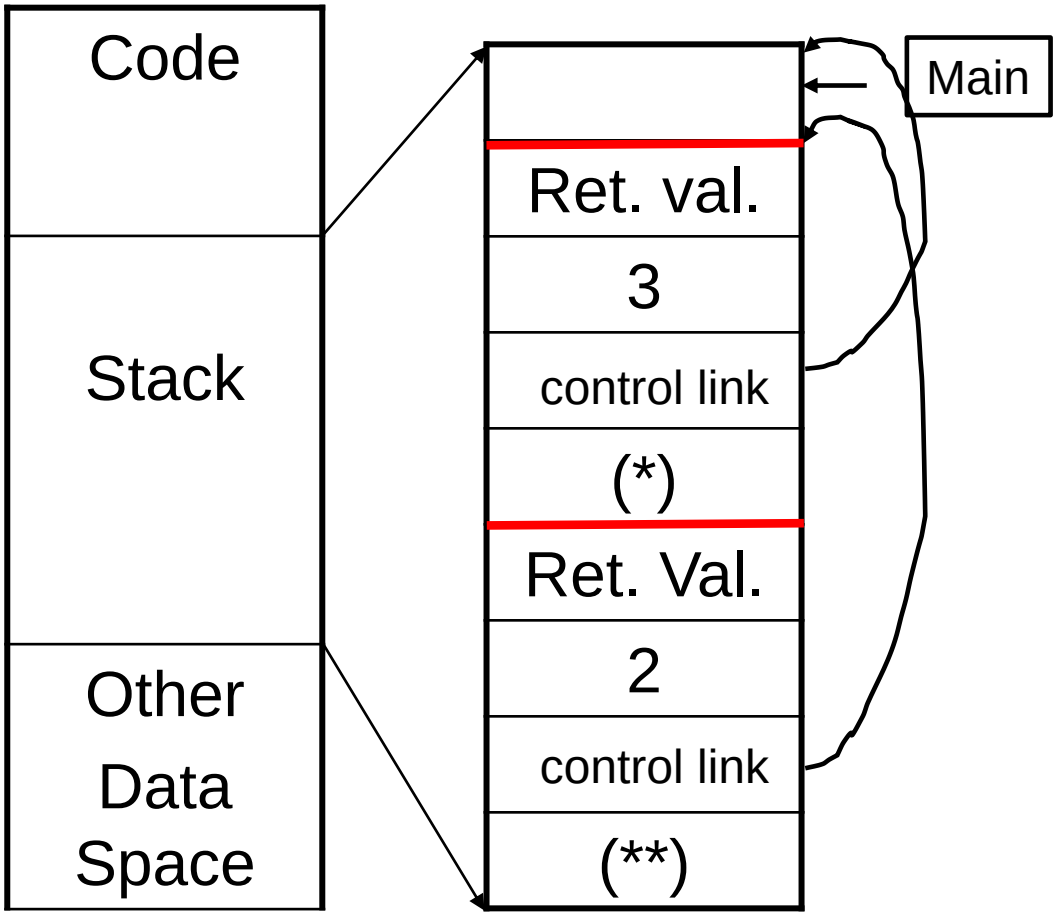
The field for saved machine status holds information about the state of the machine before the procedure is called.

The field of local data holds data that local to the callee

Temporary variables are stored in the field of temporaries.

Creation of An Activation Record

```
Class Main {
      g() : Int { 1 };
      f(x:Int): Int {
          if x = 0 then g()
                else f(x - 1)
          fi (**)
      };
      main(): Int { {f(3);   (*)}};
}
```



Discussion

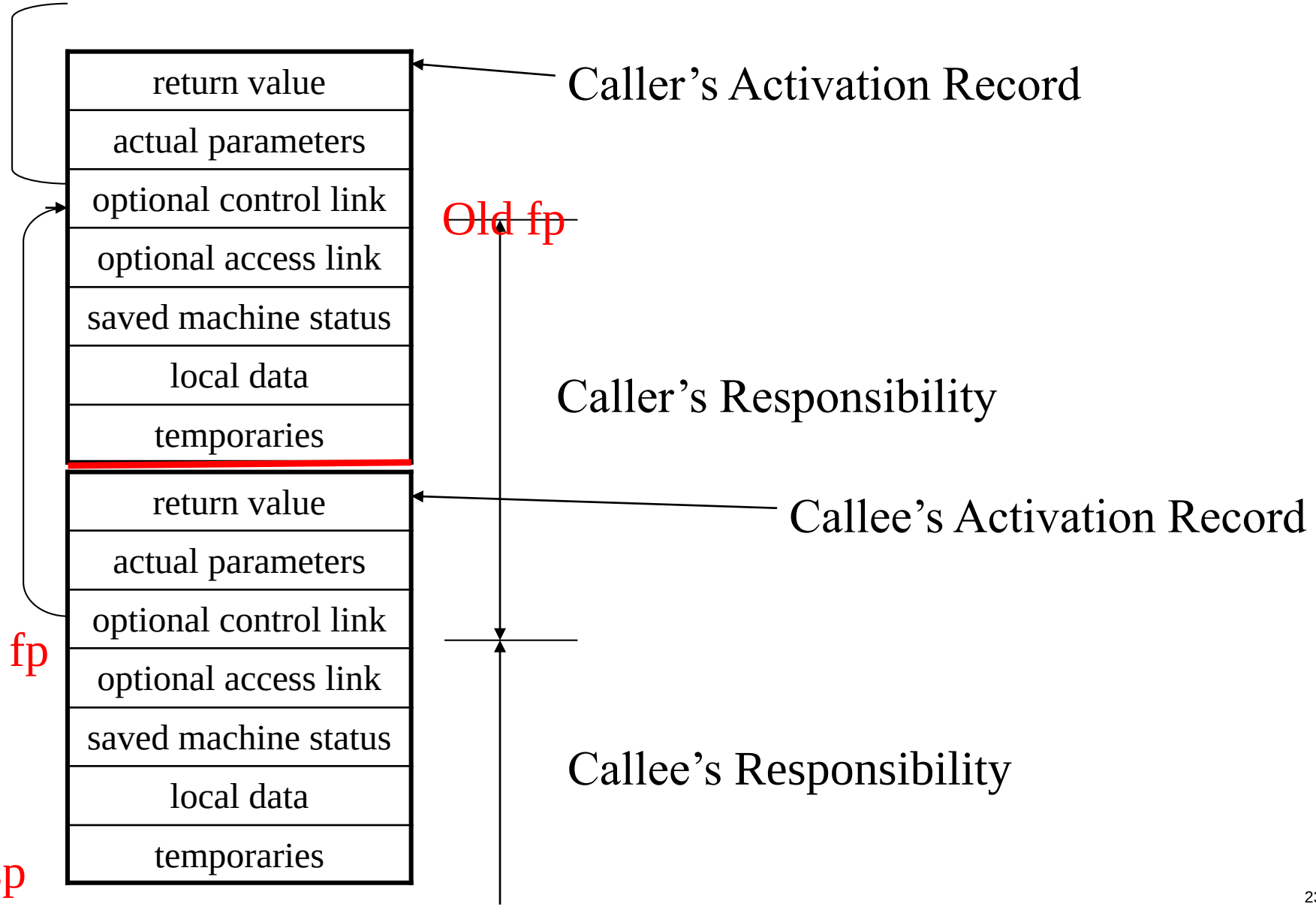
- Why put return value at the first place of the AR?
- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame
- There is nothing magic about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation
- Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments

Creation of An Activation Record

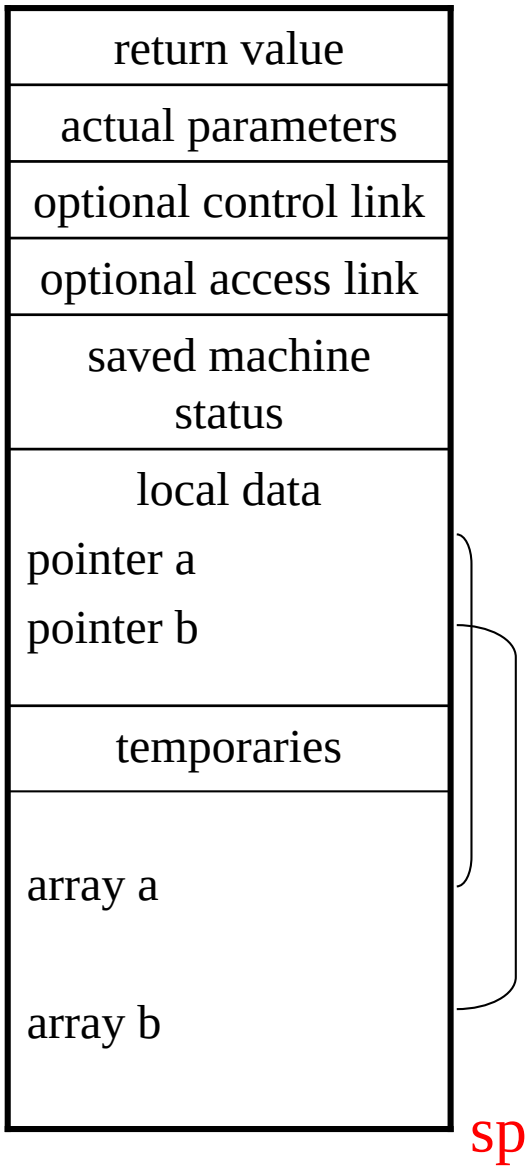
- Who allocates an activation record of a procedure?
  - Some part is created by the caller of that procedure before that procedure is entered.
  - Some part of the activation record of a procedure is created by callee immediately after that callee is entered.

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

Creation of An Activation Record (cont.)



Variable Length Data



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

Access to Nonlocal Names

- Scope rules of a language determine the treatment of references to nonlocal names.
- Scope Rules:
  - **Lexical Scope (Static Scope)**
    - Determines the declaration that applies to a name by examining the program text alone at compile-time.
    - Most-closely nested rule is used.
    - Pascal, C, ..
  - **Dynamic Scope**
    - Determines the declaration that applies to a name at run-time.
    - Lisp, APL, ...

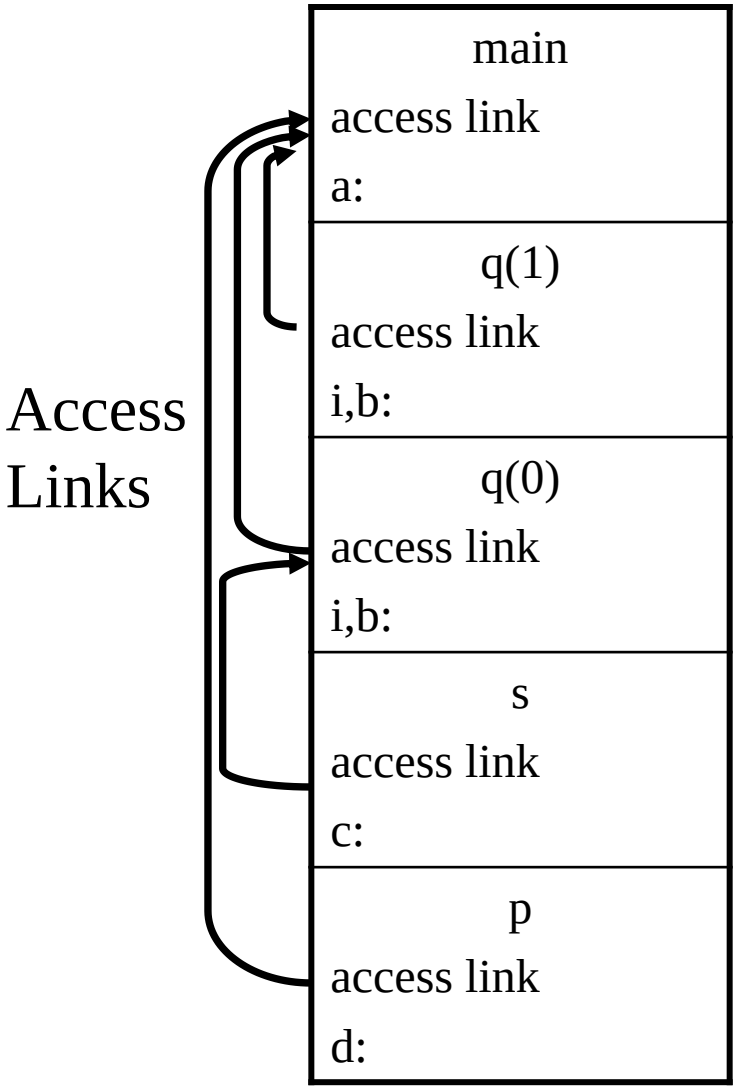
Lexical Scope

- The scope of a declaration in a block-structured language is given by the *mostly closed rule*.
- Each procedure (block) will have its own activation record.
  - procedure
  - begin-end blocks

    (treated same as procedure without creating most part of its activation record)
- A procedure may access to a nonlocal name using:
  - access links in activation records, or
  - displays (an efficient way to access to nonlocal names)
  - A display **d** is an array which consists of one pointer for each nesting depth, **d[i]** points to the i^{th} AR

Access Links

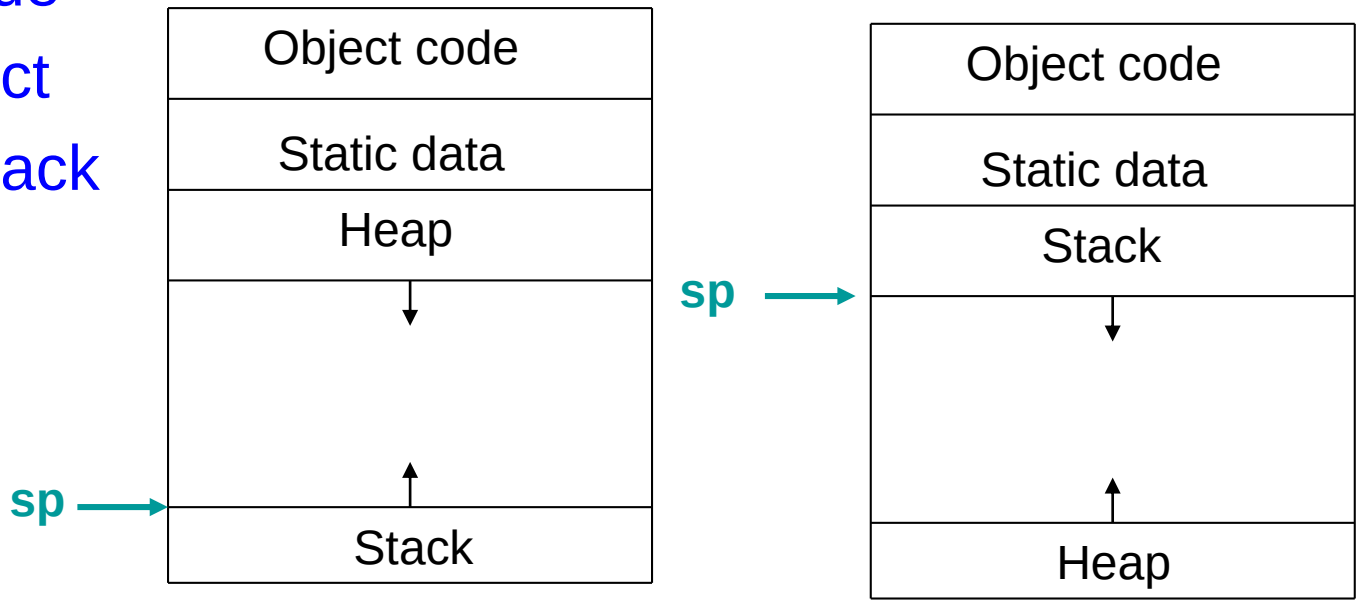
```
program main;
  var a:int;
  procedure p;
    var d:int;
    begin a:=1; end;
  procedure q(i:int);
    var b:int;
    procedure s;
      var c:int;
      begin p; end;
    begin
      if (i!=0) then q(i-1)
      else s;
    end;
  begin q(1); end;
```



Access links must be passed with procedure parameters.

Storage organization

- The run-time storage might be subdivided:
  - Target code
  - Data object
  - Control stack
  - Heap



Typical subdivision of run-time memory into code and data areas.

Storage-allocation strategies

- Three allocation strategies:
  - **Static allocation** lays out storage for all data objects at compile time.
  - **Stack allocation** manages the run-time storage as a stack.
  - **Heap allocation** allocates and de-allocates storage as needed at run time from a data area known as a heap.

Storage Allocation for Functions

- Static Allocation
  - Layout all storage for all data objects at compile time
  - Essentially every variable is stored globally
  - But the symbol table can still control local activation and de-activation of variables
  - Very restricted recursion is allowed
  - Example: Fortran 77 program

Storage Allocation for Functions

- Stack Allocation
  - Activation records are associated with each function activation
  - Activation records are pushed onto the stack when a call is made to the function
  - Storage for recursive functions is organized as a stack: **last-in first-out (LIFO)** order
  - Size of activation records can be fixed or variable

Storage Allocation for Functions

- Stack Allocation
  - Sometimes a minimum size is required
  - Variable length data is handled using pointers
  - Storage for the locals is contained in the activation record for that call.
  - Locals are deleted after activation ends
  - Caller locals are reinstated and execution continues
  - Example: C, Pascal and most modern programming languages

Storage Allocation for Functions

- Heap Allocation
  - In some special cases stack allocation is not possible
  - If local variables must be retained after the activation ends
  - If called activation outlives the caller
  - Anything that violates the last-in first-out nature of stack allocation e.g. functions as return values
  - Malloc/free, new/delete, garbage collector

Notes

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by malloc and free

Run-time Memory (MIPS)