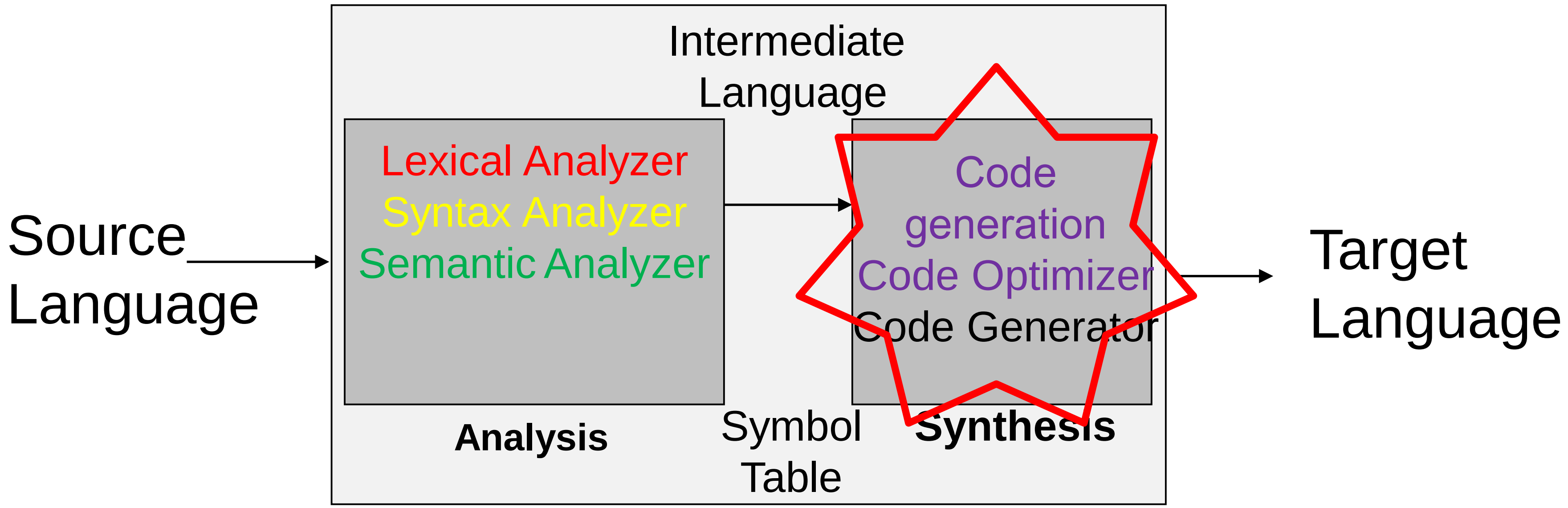
Run-Time Environments

Venkatesh
Department of Computer Science and Engineering
University Visvesvaraya College of Engineering

Current Progress



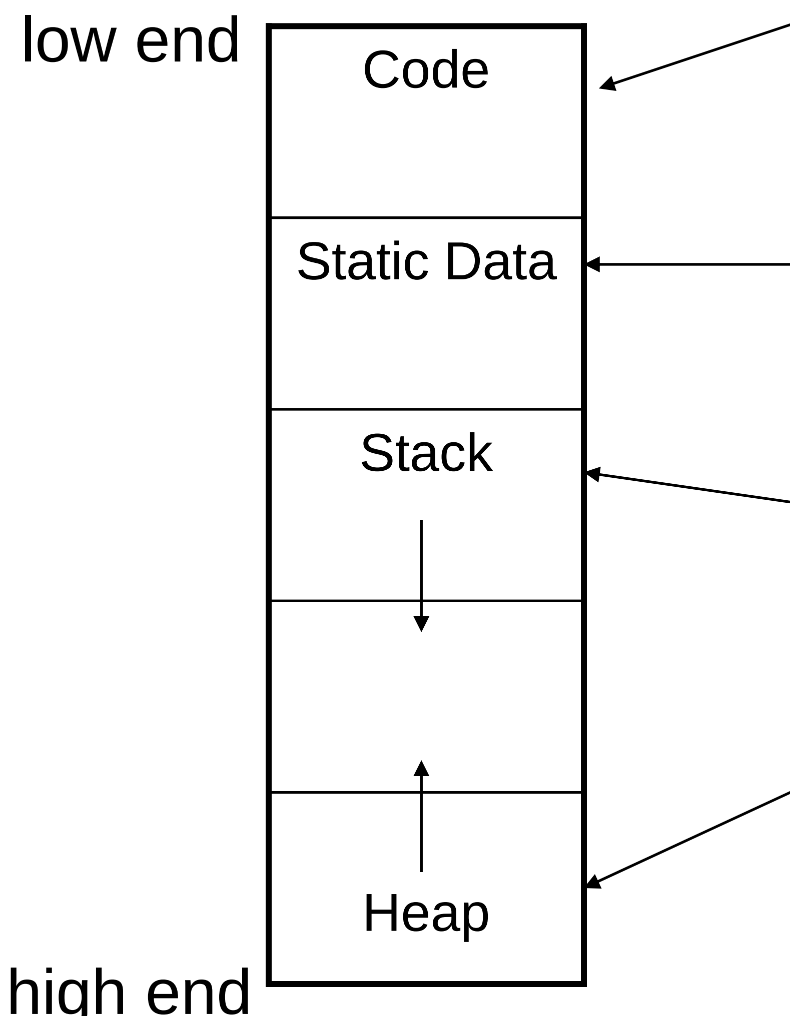
Run-Time Environment

- Before discussing code generation, we need to understand runtime Environment
- Accurately implementing abstractions embeddeid in source language definition
- abstractions: name, scope, data types, operator, procedures, parameters, flow of control constructs

Run-Time Environment Cont'd

- **Run-time environment** is created and managed by the compiler to support target programs execution.
  - allocation of storage locations for the objects
  - mechanisms to access variables
  - linkages between procedures
  - mechanisms for passing parameters
  - interfaces to the operating system
  - input/output devices, other programs=

Run-Time Storage Organization



Memory locations for code are determined at compile time.

Locations of objects can also be determined at compile time. global variables, compiler generated constants

Data objects allocated at run-time. (Activation Records)

other dynamically allocated data objects at run-time. (For example, malloc area in C). Data that may outlive the call to procedure

Procedure Activations

- An execution of a procedure starts at the beginning of the procedure body;
- When the procedure is completed, it returns the control to the point immediately after the place where that procedure is called.
- Each execution of a procedure is called as its **activation**.
- **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. (single thread)
- If a procedure is recursive, a new activation can begin before an earlier activation of the same procedure has ended.
- decision-Static, if compiler looks at program code, not at what program does when it executes

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. *
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] s
       a[m..p-1] are less than v, a[p] = v, and a[p+1 .
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

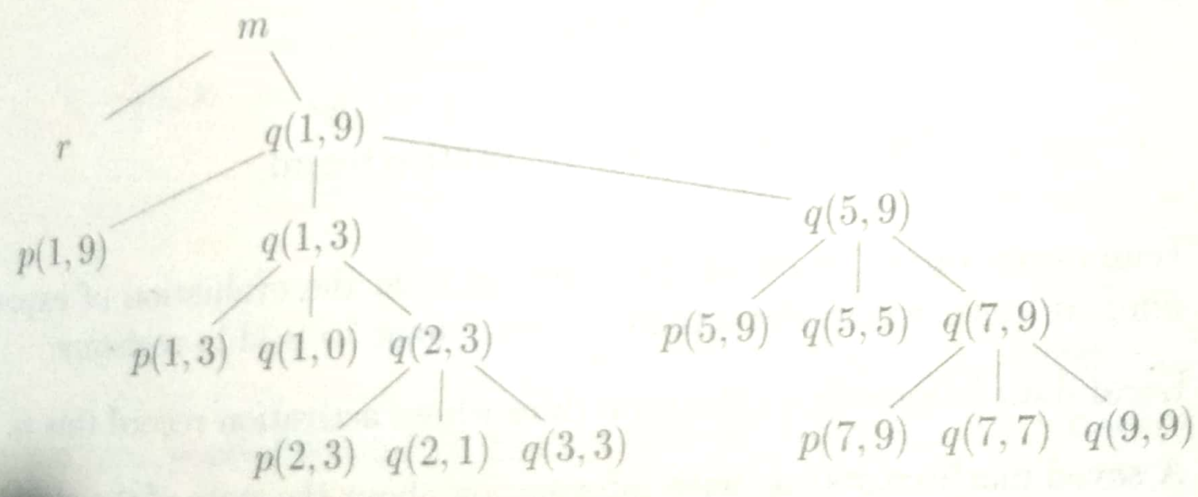


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

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

Activation Tree Cont'd

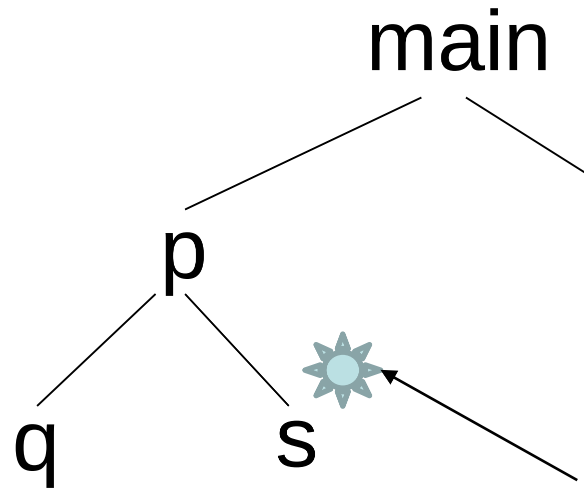
```
program main;
  procedure s;
    begin ... end;
  procedure p;
    procedure q;
      begin ... end;
    begin q; s; end;
  begin p; s; end;
```

```
enter main
    enter p
        enter q
        exit q
        enter s
        exit s
    exit p
    enter s
    exit s
exit main
```

A Nested Structure

Activation Tree Cont'd

Sequence of procedure is depth-first traversal of the activation tree



Now control is in s; what procedures are live? What is their order?

Activation Tree

Remarks for Activation Tree

- The activation tree depends on run-time behavior.

- The activation tree may be different for every program input.

why?

- Since activations are properly nested, a $\boxed{}$ can track currently active procedures.

What data structure?

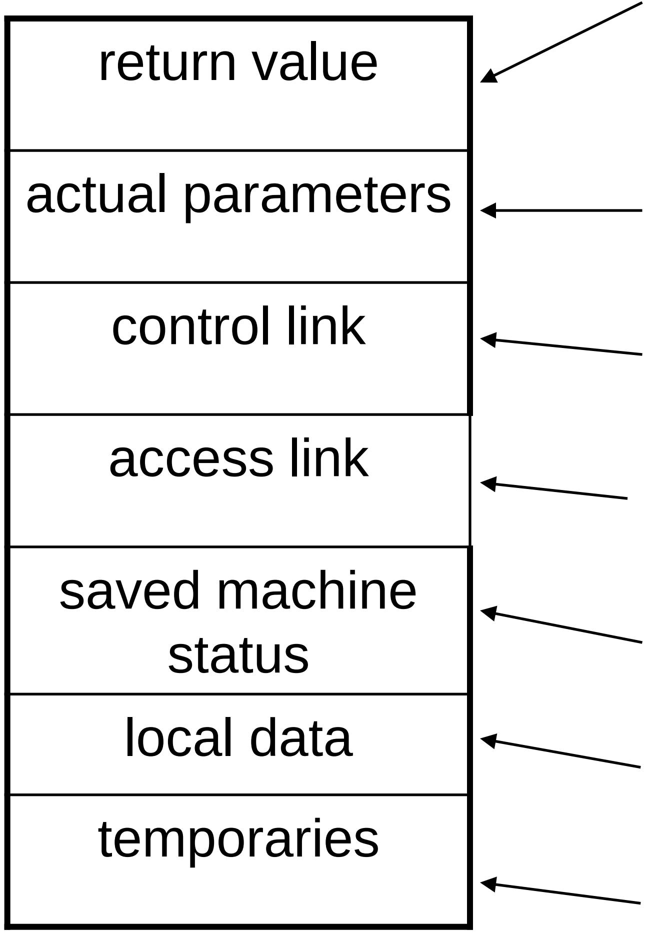
Control Stack

- 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 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.
- 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 exits.
- 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’d



store the returned values to the calling procedure. In practice, we may use a machine register for the return value.

used by the calling procedure to supply parameters to the called procedure

points to the activation record of the caller

used to refer nonlocal data, e.g., in other activation records

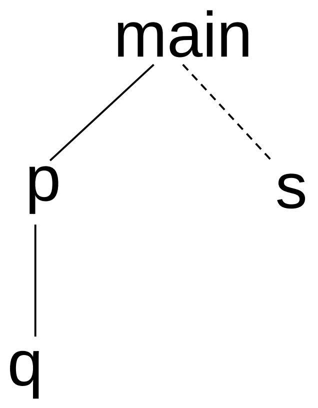
holds information about the state of the machine before the procedure is called. return address, PC, register contents muste be restored.

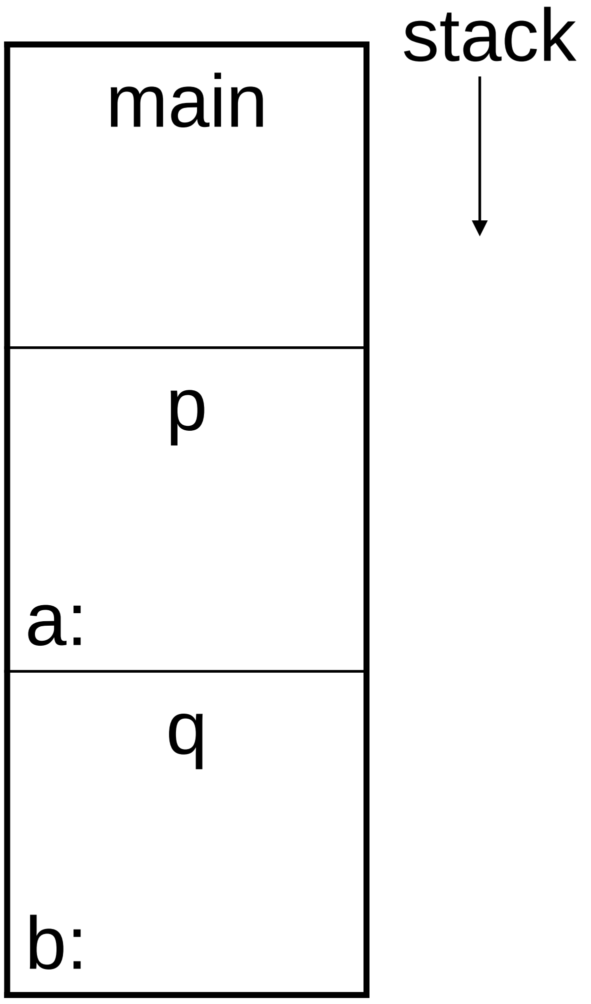
holds data that local to an execution of a procedure

temporary variables

Activation Records Example

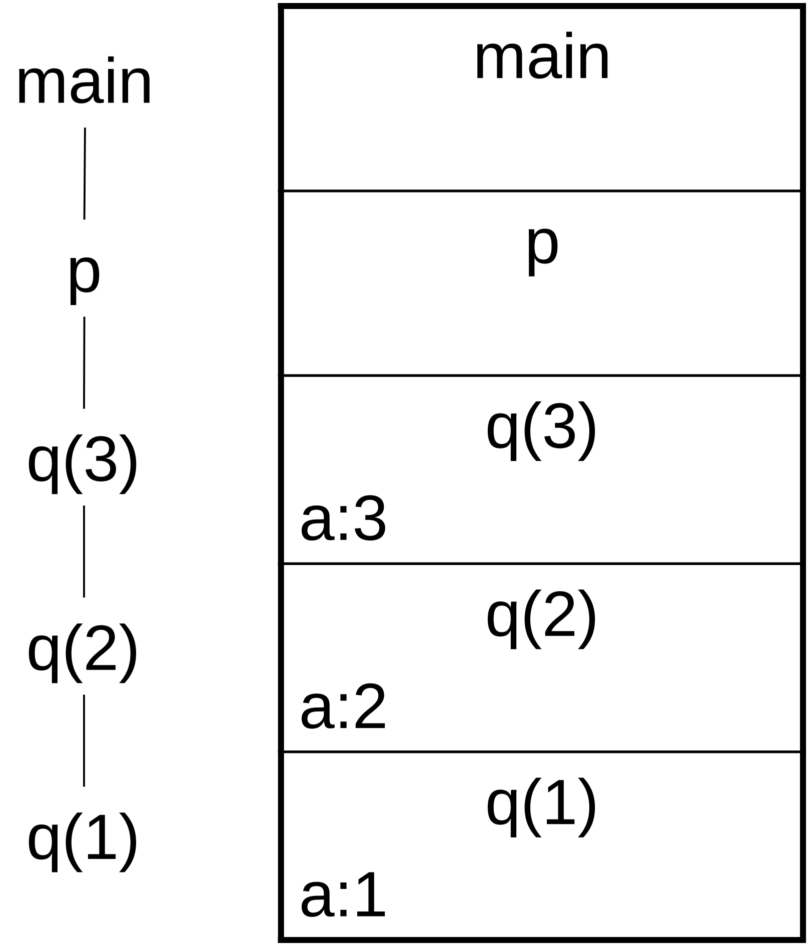
```
program main;
  procedure p;
    var a:real;
    procedure q;
      var b:integer;
      begin ... end;
    begin q; end;
  procedure s;
    var c:integer;
    begin ... end;
  begin p; s; end;
```





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;
```



Test Yourself

- The following function computes Fibonacci numbers recursively. Assume that the initial call is f(4).

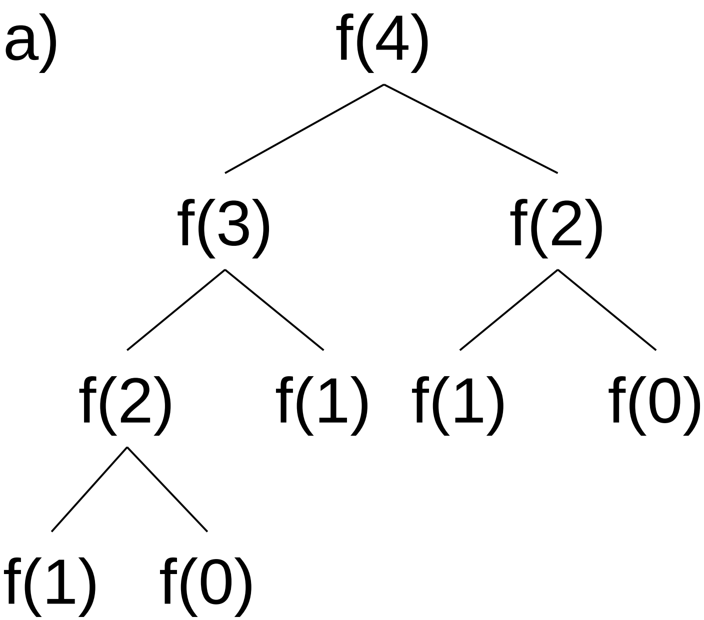
```
function f(n:integer):integer;
  var s:integer;
  var t:integer;
  begin
    if (n<2) then f:=1;
    else
      begin
        s:=f(n-1);
        t:=f(n-2);
        f:=s+t;
      end;
  end;
```

a) Draw the complete activation tree.

b) Draw the activation records in the stack the first time f(0) is about to return?

Solution

a)



b)

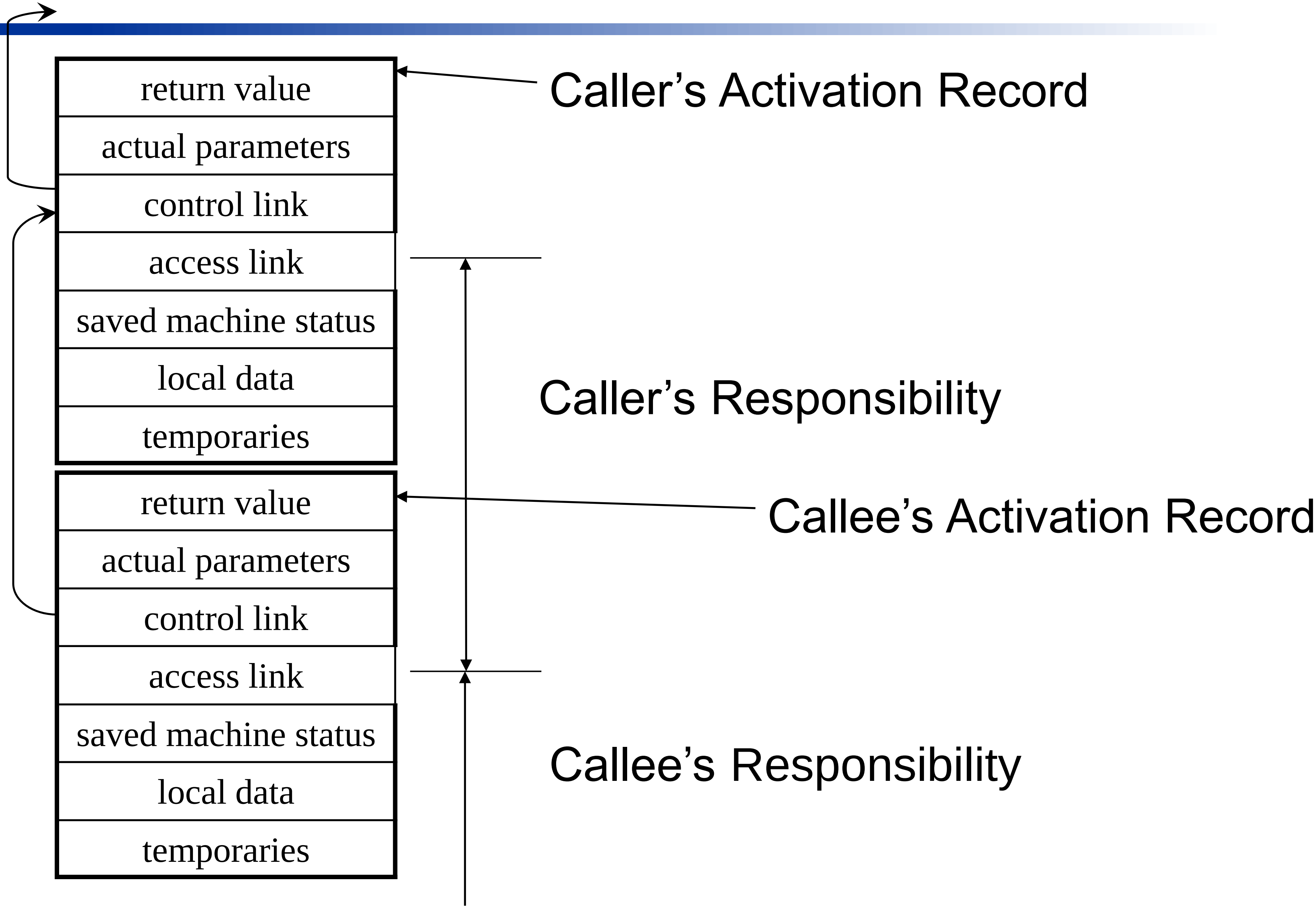
f(4)
n: 4 s: t:
f(3)
n: 3 s: t:
f(2)
n: 2 s: 1 t:
f(0)
n: 0 s: t:

Creation of An Activation Record

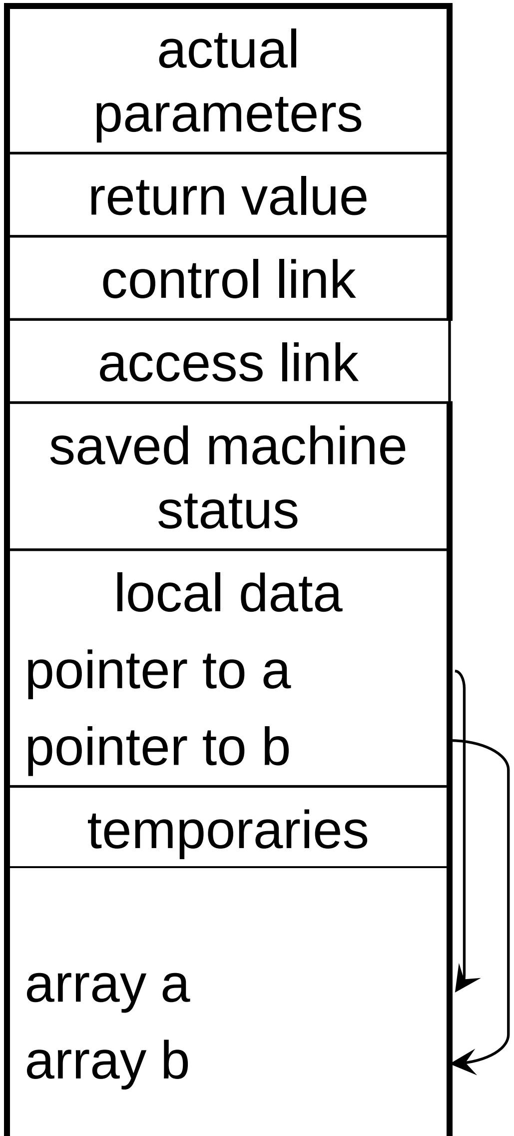
- Who allocates an activation record of a procedure?
  - Some part is created by the caller of that procedure before that procedure is entered.
  - Some part of the activation record of a procedure is created by that procedure itself immediately after that procedure is entered.

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

Creation of An Activation Record Cont'd



Variable-Length Data



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

Variable Scopes

- The same variable name can be used in 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 p;
  var b:real;
  procedure q;
    var a: integer;
    begin a := 1;  b := 2; end;
  begin ... end;
```

a is local
b is non-local

Access to Nonlocal Names

- Scope rules of a language determine the treatment of references to nonlocal names.
- Scope rules:
  - **Lexical scope (static scope)**
    - Most-closely nested rule is used.
    - Determines the declaration that applies to a name by examining the program text alone at compile-time.
    - 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)

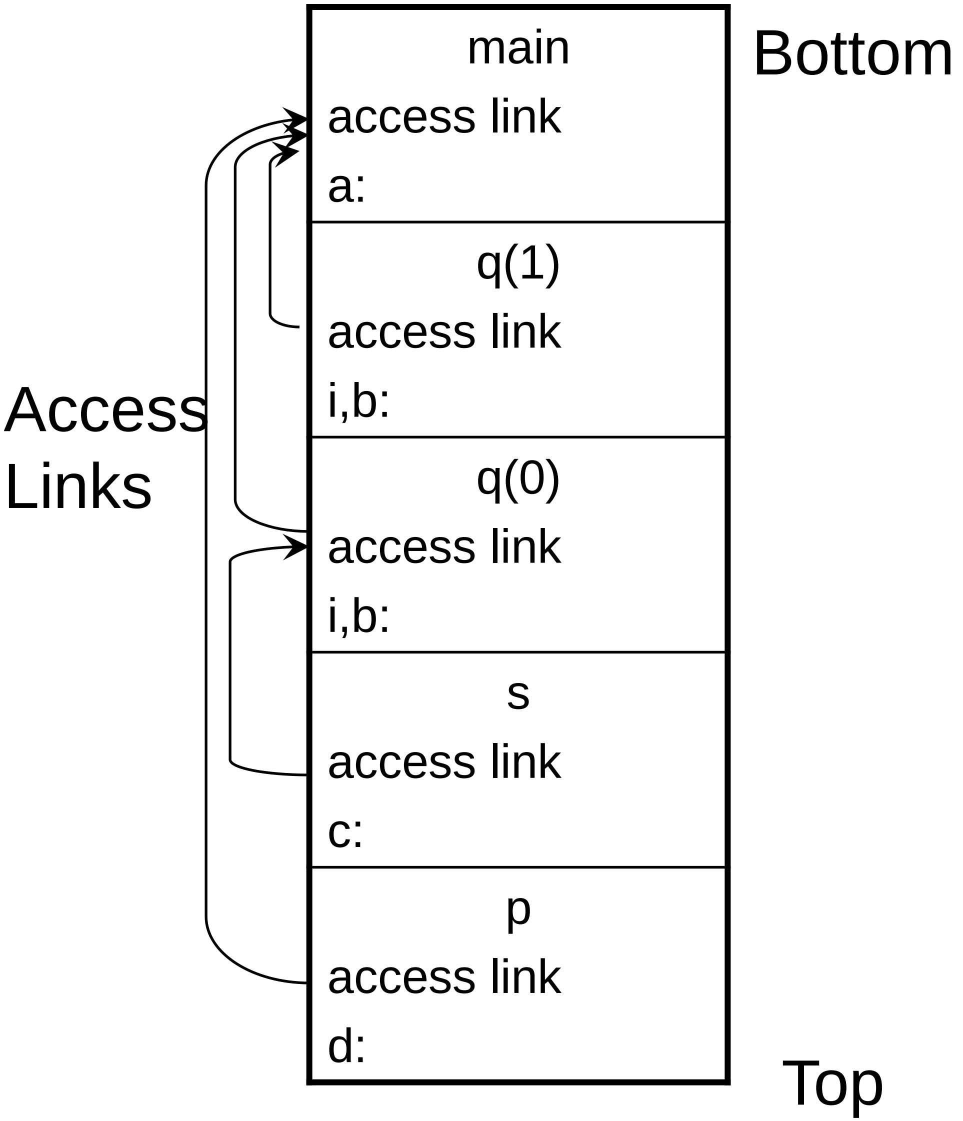
Access Links

- Access links are implemented by adding pointers to each activation record.
  - If procedure p is nested immediately within procedure q in the source code, then the access link in any activation of p points to the most recent activation of q.

- Access links form a chain from the activation record at the top of the stack to a sequence of activations at lower nesting depths. Along this chain are all the activations whose data and procedures are accessible to the currently executing procedure.

Access Links Example

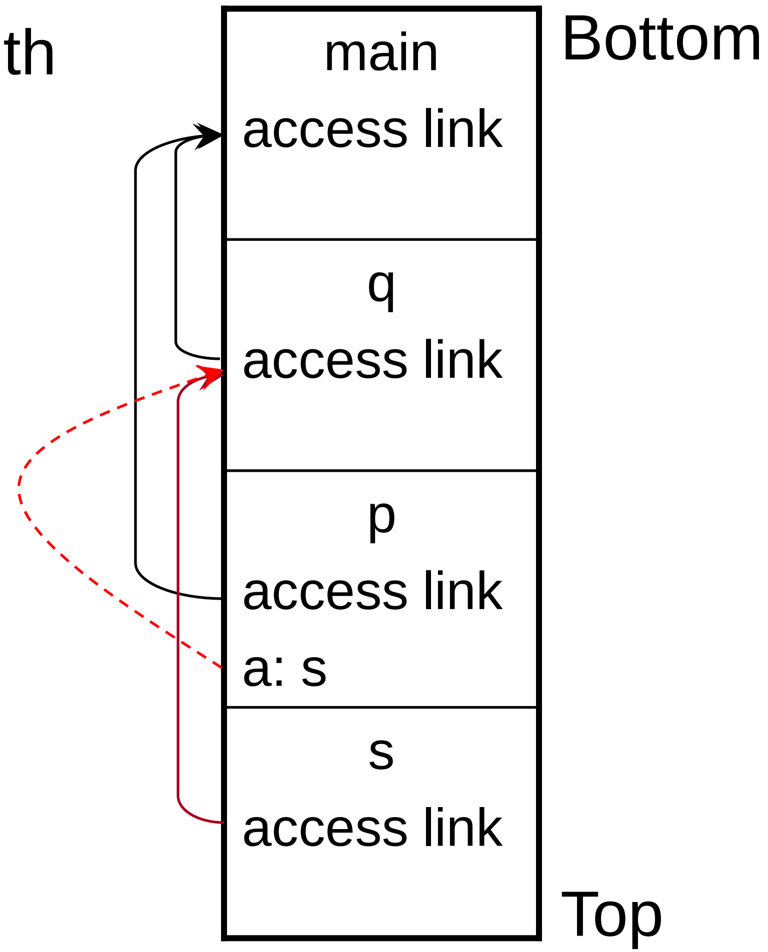
```
program main;
  var a:int;
  procedure p;
    var d:int;
    begin a:=1; end;
  procedure q(i:int);
    var b:int;
    procedure s;
      var c:int;
      begin p; end;
    begin
      if (i<>0) then q(i-1)
      else s;
    end;
  begin q(1); end;
```



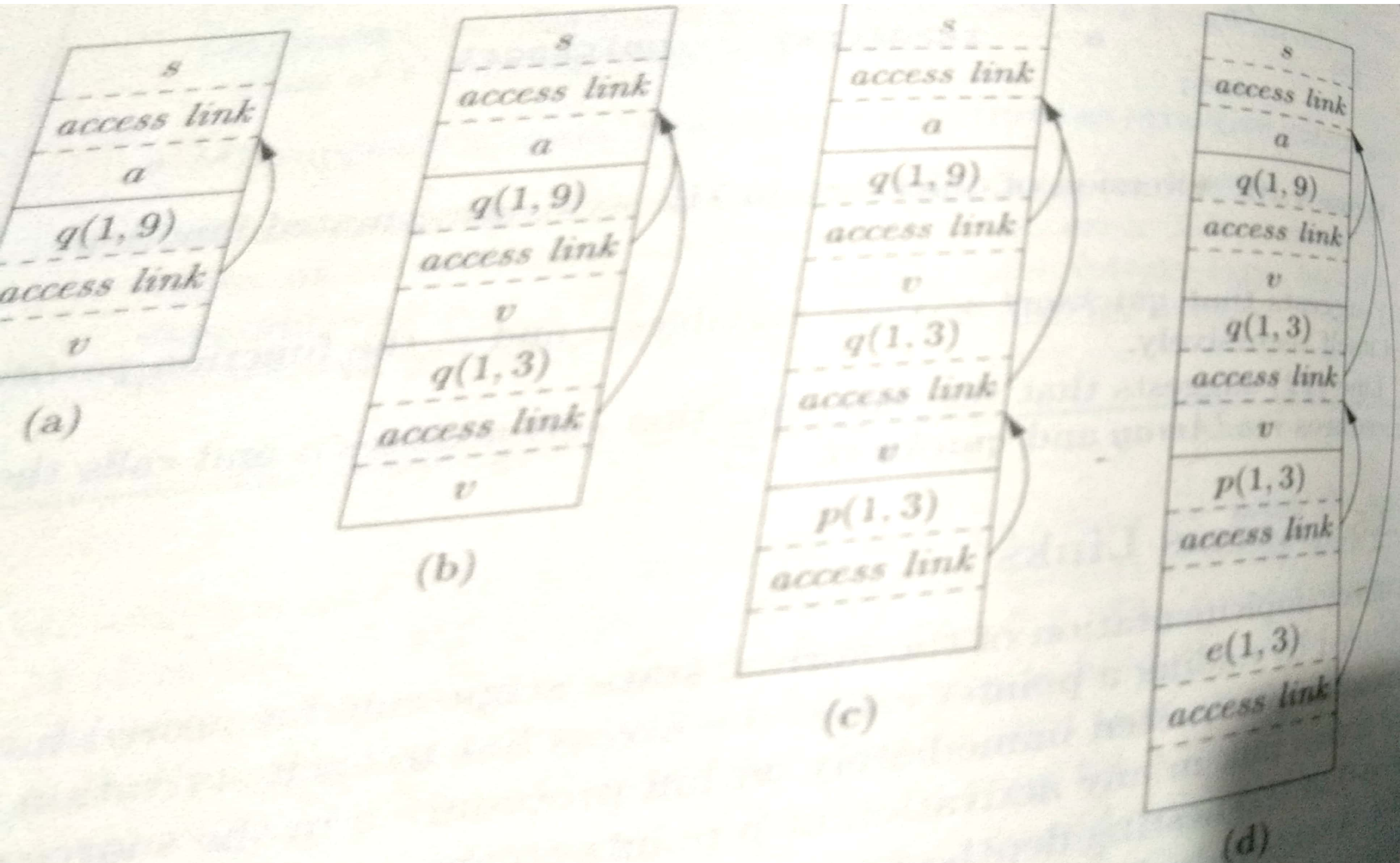
Procedure Parameters

- Access links must be passed with procedure parameters.

```
program main;
    procedure p(procedure a);
        begin a; end;
    procedure q;
        procedure s;
            begin ... end;
        begin p(s) end;
    begin q; end;
```

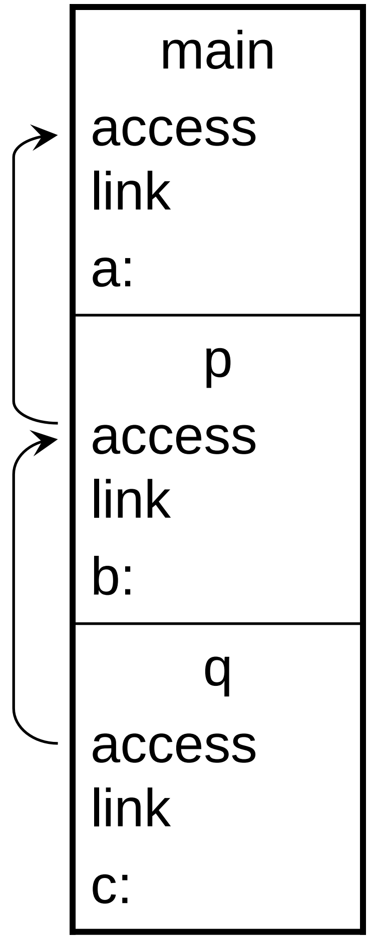


```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ··· ;
4)                 ··· a ··· ;
5)         fun exchange(i,j) =
6)                 ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                        ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
   in
12)        ··· a ··· readArray ··· quicksort ···
   end;
```


s
access link
a
q(1, 9)
access link
v
q(1, 3)
access link
v
p(1, 3)
access link
e(1, 3)
access link
(a)
(b)
(c)
(d)

Accessing Nonlocal Variables

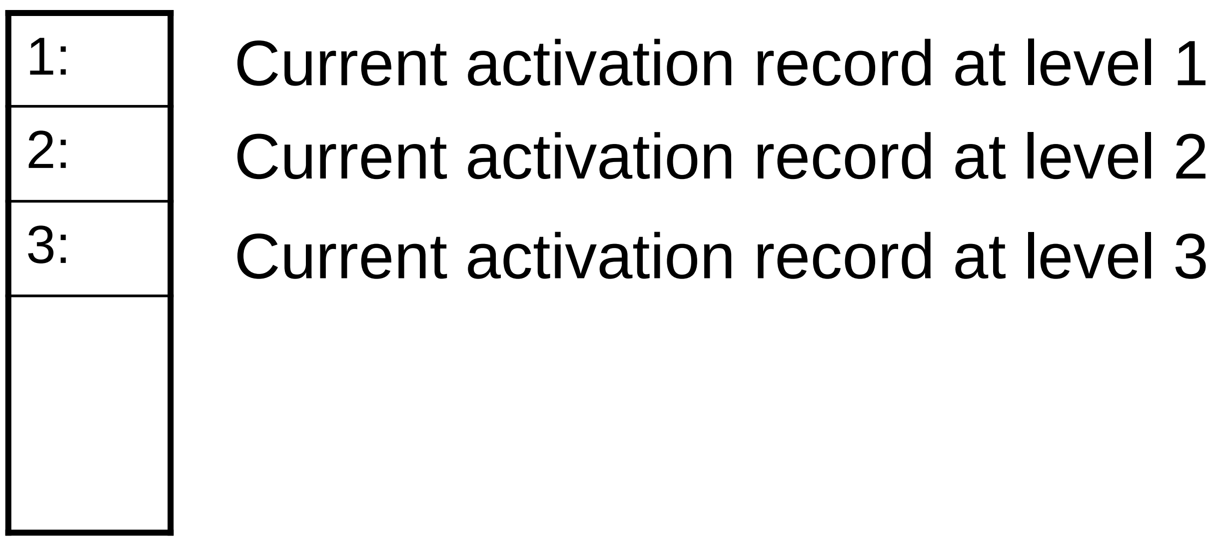
```
program main;
    var a:int;
    procedure p;
        var b:int;
        procedure q();
            var c:int;
            begin
                c:=a+b;
            end;
        begin q; end;
    begin p; end;
```



addrC := offsetC(curr_pos)
pos := traceback(curr_pos)
addrB := offsetB(pos)
pos := traceback(pos)
addrA := offsetA(pos)
ADD addrA, addrB, addrC

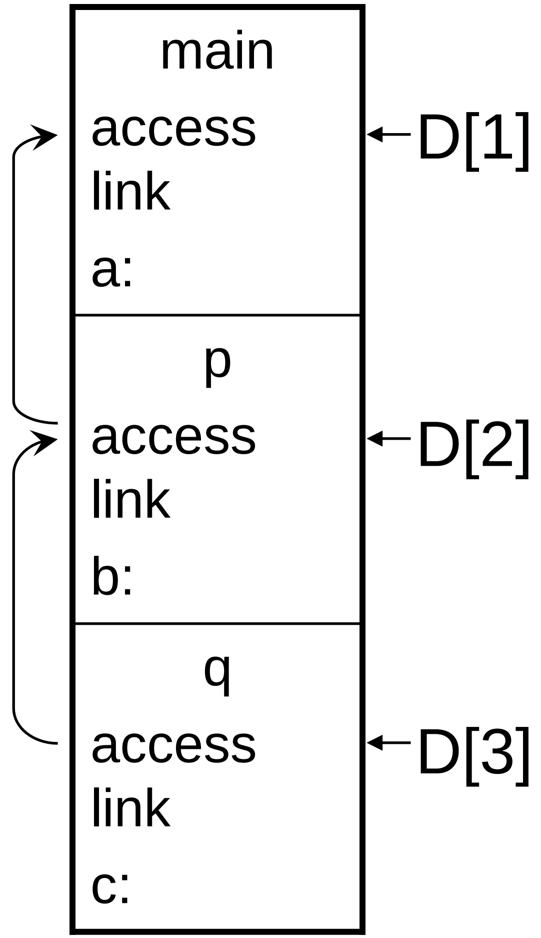
Displays

- An array of pointers to activation records can be used to access activation records.
- This array is called as displays.
- For each level, there will be an array entry.



Accessing Nonlocal Variables using Display

```
program main;
    var a:int;
    procedure p;
        var b:int;
        procedure q();
            var c:int;
            begin
                c:=a+b;
            end;
        begin q; end;
    begin p; end;
```



addrC := offsetC(D[3])
addrB := offsetB(D[2])
addrA := offsetA(D[1])
ADD addrA, addrB, addrC

Dynamic Scope

- A nonlocal name's binding is not determined lexically, but inherits that of the caller.

- Example:

```
int x = 0;
int f() { return x; }
int main() { int x = 1; return f(); }
```

  - Lexical scope: return 0
  - Dynamic scope: return 1

Implementing Dynamic Scope

- Stack-based method:
  - traverses the runtime stack, checking each activation record for the first value of the identifier

- Table-based method:
  - uses a table to associate each name with its current meaning
  - when a procedure is activated, locate each local name to the entry in the table and store the name’s previous meaning in the activation record

Problems with Dynamic Scope

- Bad program readability, understanding of the program relies on a simulation of the it

- Not suitable for static type checking

- Therefore, dynamic scope is not commonly used

Heap

- A value that outlives the procedure that creates it cannot be kept in the activation record

    Example: `int* func() { return new int[10]; }`

- Languages with dynamically allocated data use a heap to store dynamic data

Where to Put the Heap?

- The heap is an area of memory which is dynamically allocated.
- Like a stack, it may grow and shrink during runtime.

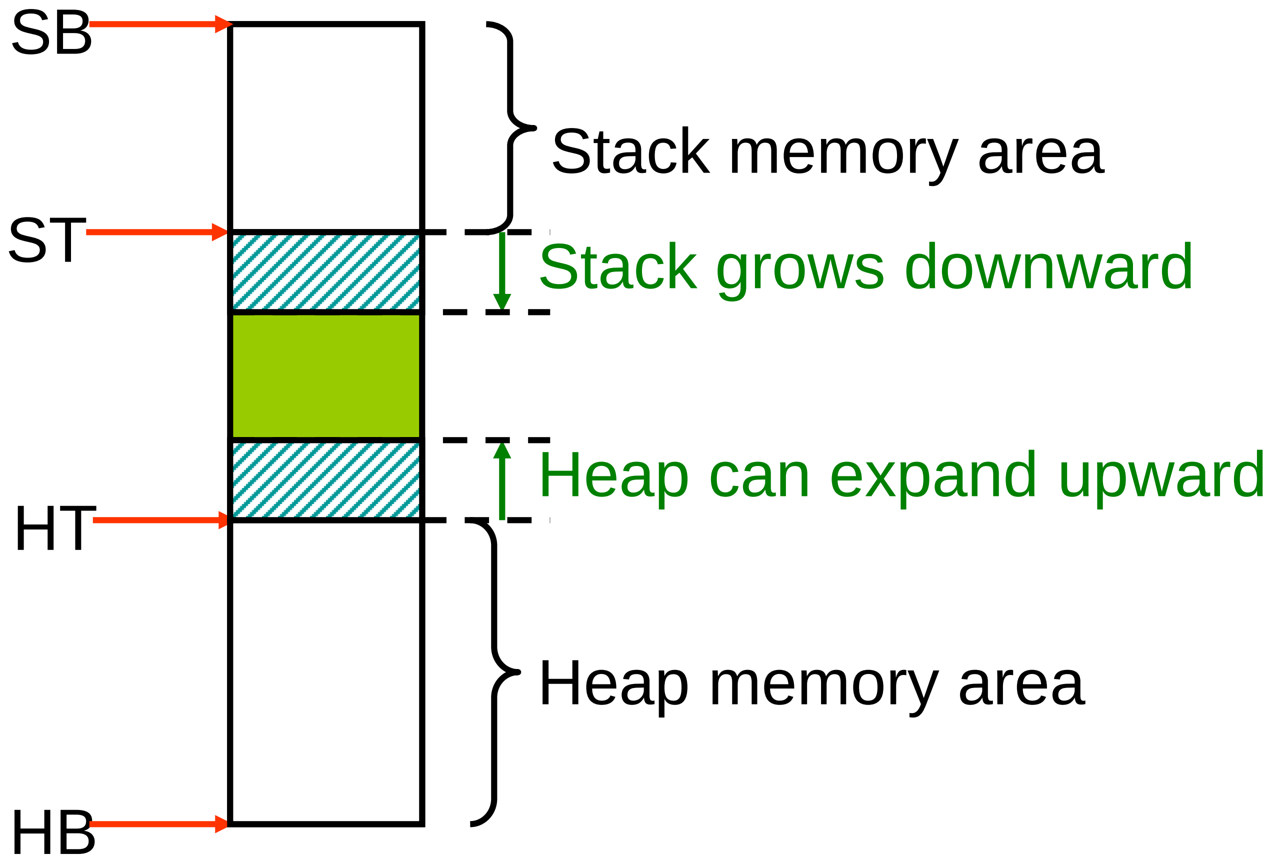
- Unlike a stack it is not a LIFO

  $\Rightarrow$more complicated to manage

- In a typical programming language implementation we will have both heap and stack allocated memory coexisting.

Where to put the heap?

Let both stack and heap share the same memory area, but grow towards each other from opposite ends!



Heap Management

- Allocation: When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size
- Deallocation: The memory manager returns de-allocated space to the pool of free space

- Problem: Heap is fragmented into too many small and noncontiguous chunks

Allocation

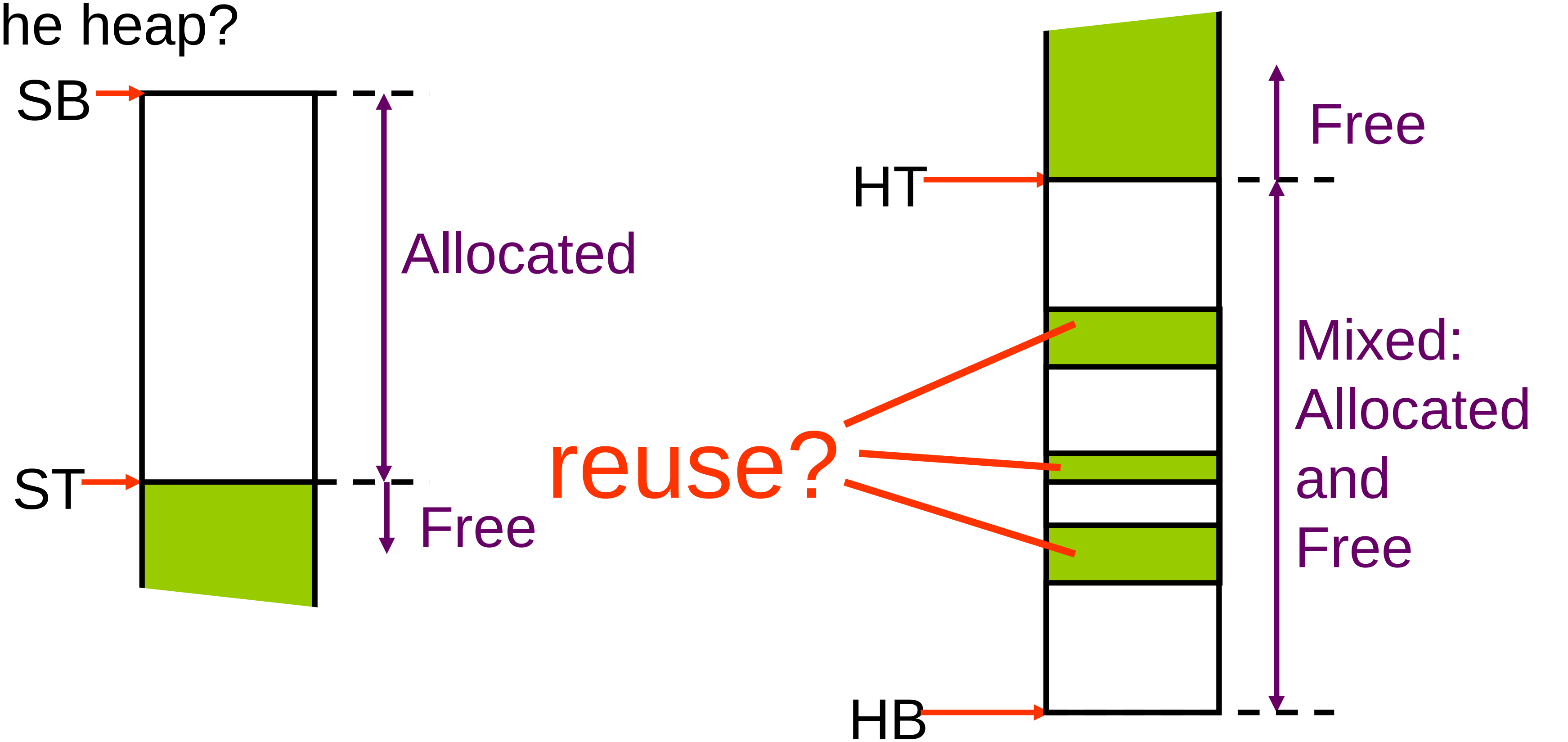
- Best-fit algorithm: allocates the requested memory in the smallest available chunk that is large enough.

- First-fit algorithm: places an object in the first (lowest-address) chunk in which it fits
  - takes less time than best-fit to place objects
  - overall performance is worse than best-fit

Managing Free Space

Stack is LIFO allocation => ST moves up/down everything above ST is in use/allocated. Below is free memory. This is easy! But …

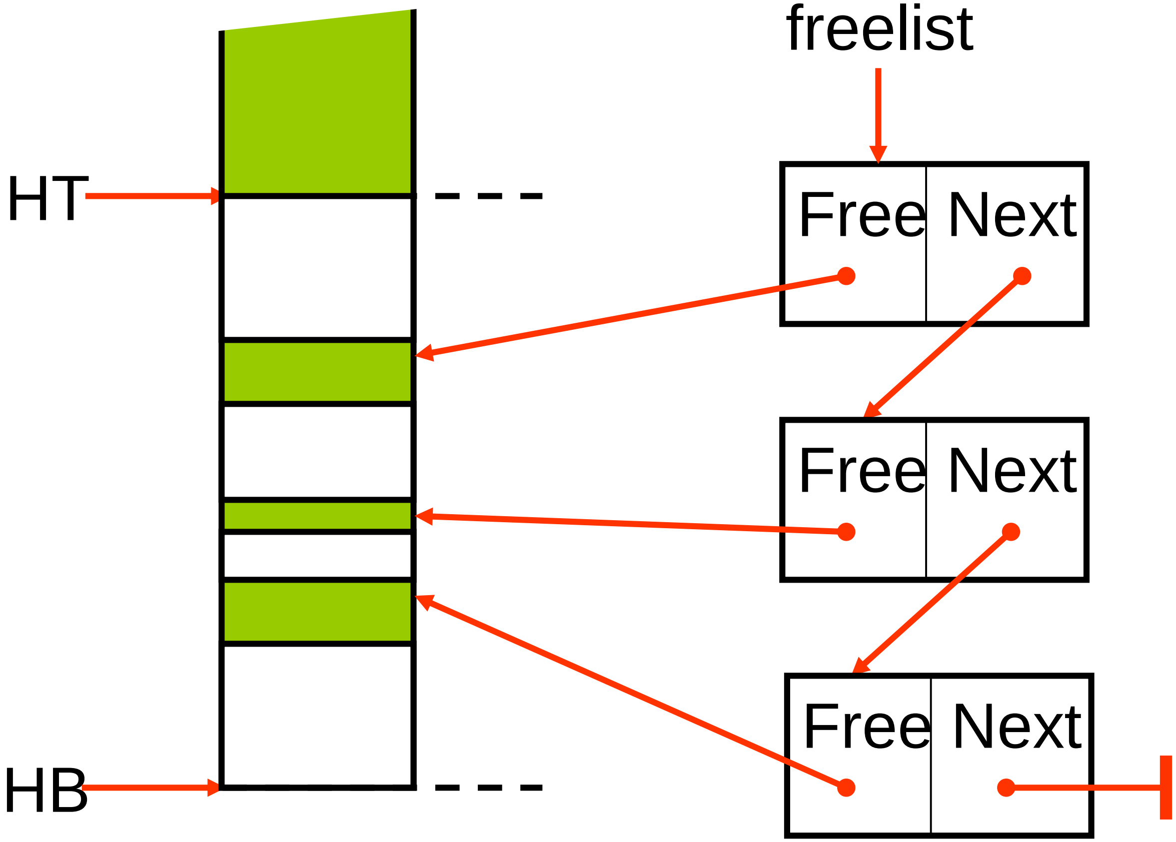
Heap is not LIFO, how to manage free space in the "middle" of the heap?



Managing Free Space

How to manage free space in the "middle" of the heap?

=> keep track of free blocks in a data structure: the "free list". For example we could use a linked list pointing to free blocks.



But where do we store this data structure?

How to keep track of free memory?

Where do we find the memory to store a freelist data structure?
=> Since the free blocks are not used by the program, memory manager can use them for storing the freelist itself.