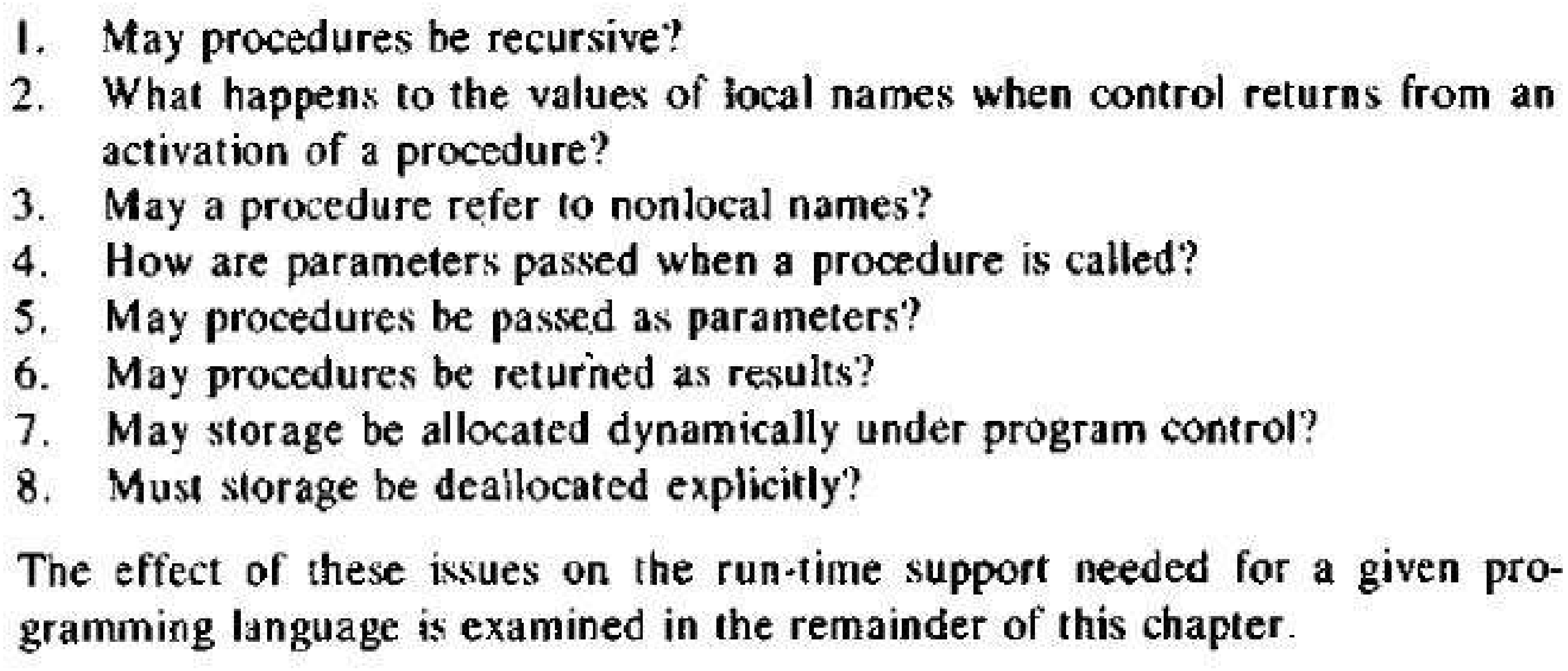
Run-Time Environments

CSE-4801 Compiler Design

Allocation and deallocation of data objects are managed by run-time environment packages.

Some questions need to be answered before designing run-time environment for a compiler



1. May procedures be recursive?
2. What happens to the values of local names when control returns from an activation of a procedure?
3. May a procedure refer to nonlocal names?
4. How are parameters passed when a procedure is called?
5. May procedures be passed as parameters?
6. May procedures be returned as results?
7. May storage be allocated dynamically under program control?
8. Must storage be deallocated explicitly?

The effect of these issues on the run-time support needed for a given programming language is examined in the remainder of this chapter.

Activation of records

Each execution of a function/procedure is referred as activation of that function/ procedure.

```
void main()
{
..
sort();      // → activation of function sort()
..
}
```

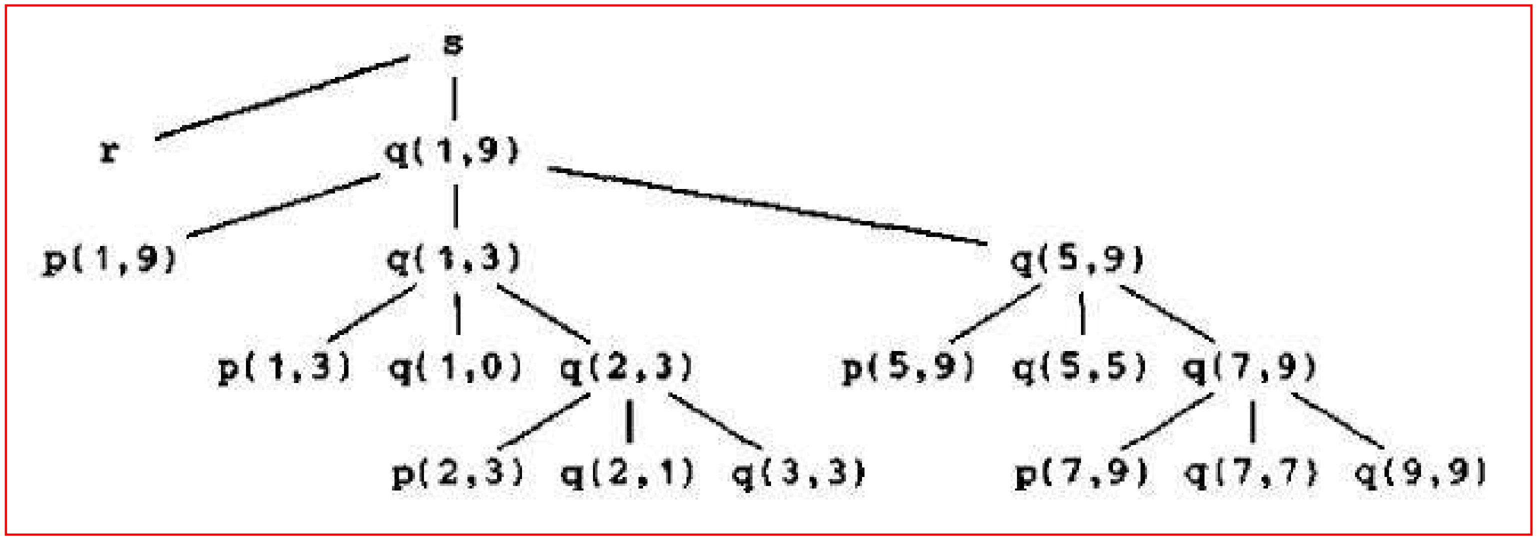
Flow of control between functions can be depicted as a tree.

A function for partition sort

```
program sort(input, output);
    var a : array [0..10] of integer;

    procedure readarray;
        var i : integer;
        begin
            for i := 1 to 9 do read(a[i])
        end;

    function partition(y, z: integer) : integer;
        var i, j, x, v: integer;
        begin ...
        end;

    procedure quicksort(m, n: integer);
        var i : integer;
        begin
            if ( n > m ) then begin
                i := partition(m,n);
                quicksort(m,i-1);
                quicksort(i+1,n)
            end
        end;

    begin
        a[0] := -9999; a[10] := 9999;
        readarray;
        quicksort(1,9)
    end.
```

```
Execution begins...
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)
Execution terminated.
```

Activation tree

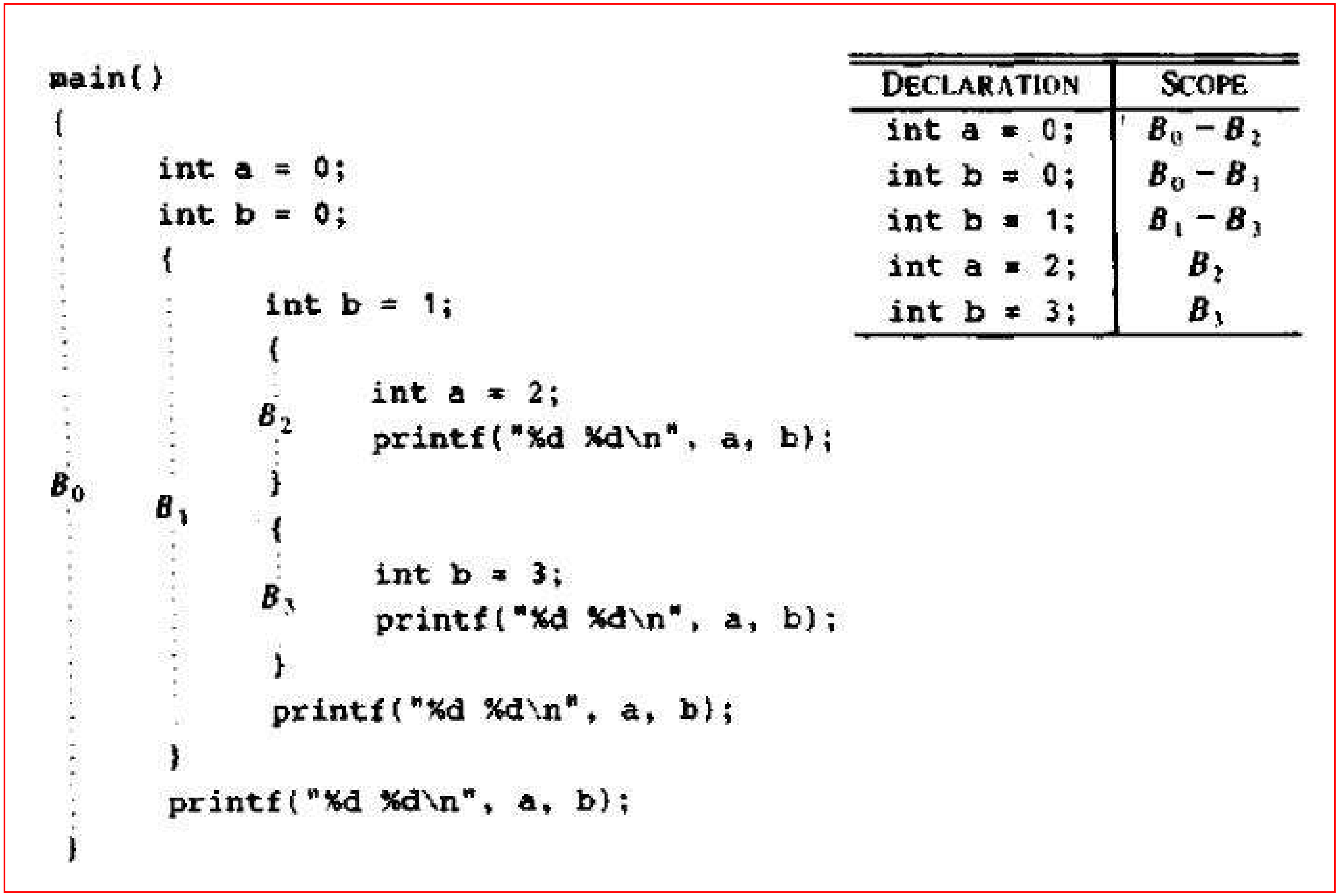


- Root represents activation of main program.
- Each node represents activation of a function.
- If control flows from ***function a*** to ***function b*** then ***node b*** will be child of ***node a*** in activation tree.
- ***Node a*** is at the left of ***node b*** if and only if the lifetime of ***function a*** occurs before ***function b***.

Control Stack

- We can maintain a stack, called control stack, to keep track of live function activations.

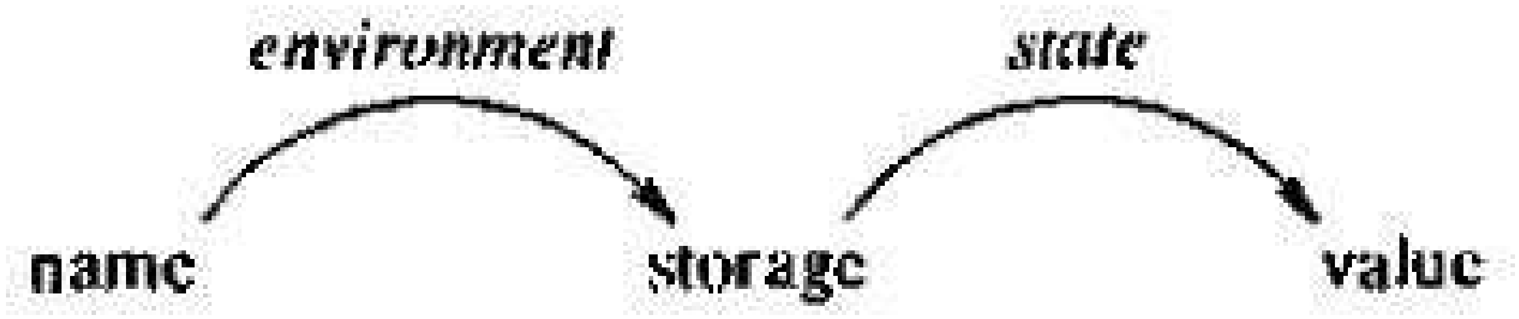
Scope of a declaration



```
main()
{
        int a = 0;
        int b = 0;
        {
                int b = 1;
                {
                        int a = 2;
                        printf("%d %d\n", a, b);
                }
                {
                        int b = 3;
                        printf("%d %d\n", a, b);
                }
                printf("%d %d\n", a, b);
        }
        printf("%d %d\n", a, b);
}
```

DECLARATION	SCOPE
int a = 0;	$B_0 - B_2$
int b = 0;	$B_0 - B_1$
int b = 1;	$B_1 - B_3$
int a = 2;	B_2
int b = 3;	B_3

Binding of names



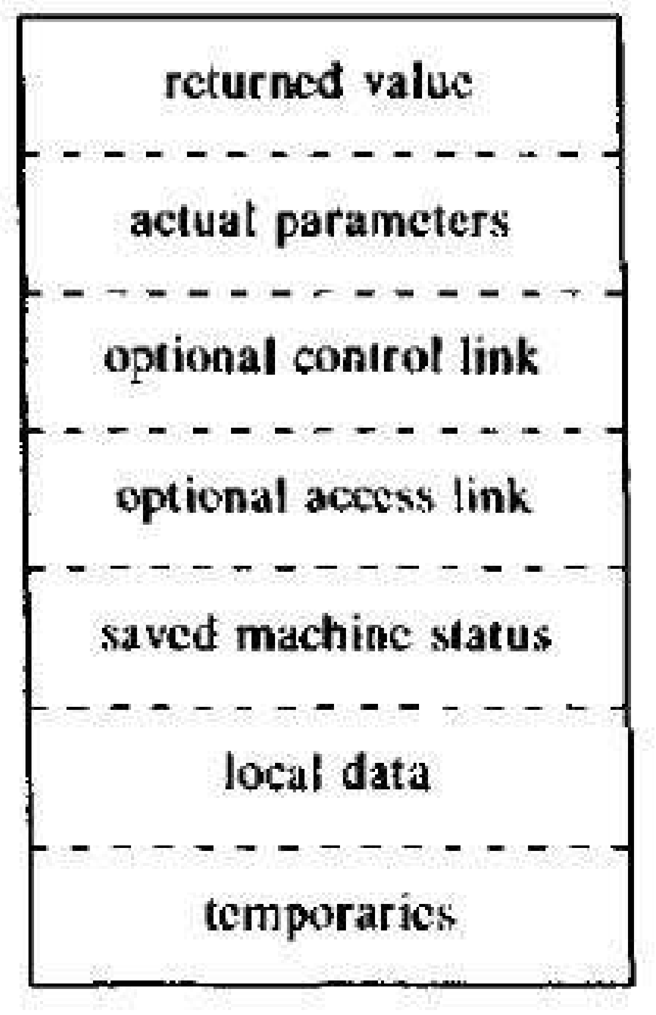
Activation Records

Information needed by a single execution of a function is managed using a contiguous block of records called and ***activation record*** or ***frame***.

It might contains:

- Temporary values
- Local data
- Machine status data
- Access link (optional: to access data stored in other activation records)
- Control link (optional: points to the activation record of the caller function)
- Field for actual parameters
- Field for returned value

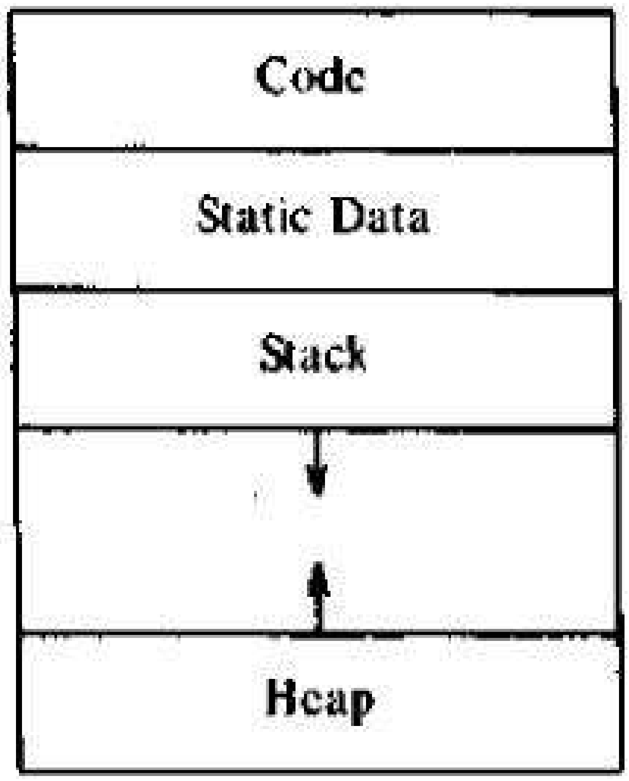
A general activation record



Subdivision of runtime memory for a function

Generated target code
Data objects
Counterpart of control stack

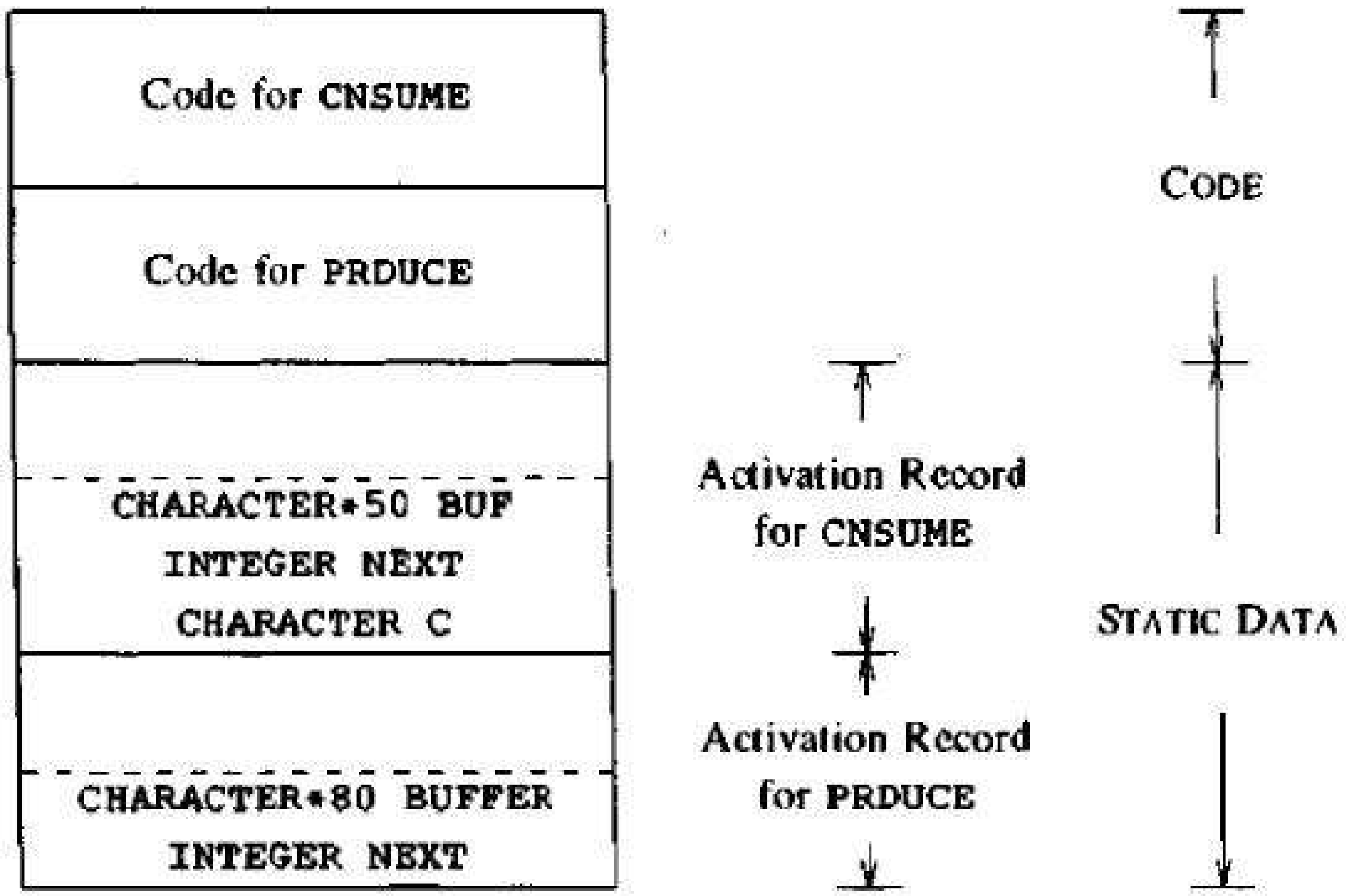
Typical subdivision of run-time memory of a function



Storage Allocation Strategies

1. Static Allocation: lays out all data objects at compile time.
2. Stack Allocation: manages the run-time storages as a stack.
3. Heap Allocation: Allocate and deallocate data storage as needed at run-time from a data area known as Heap.

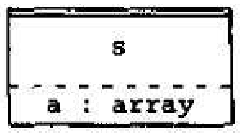
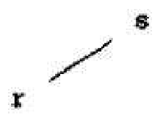
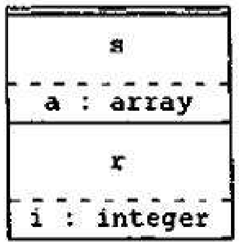
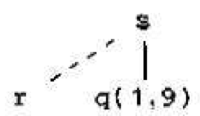
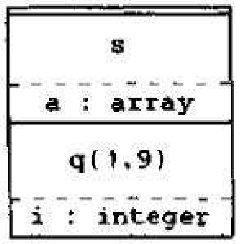
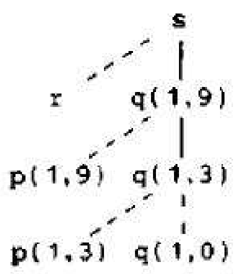
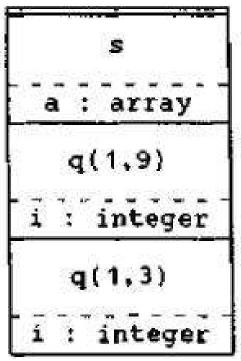
Static Allocation



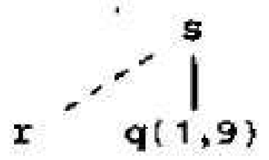
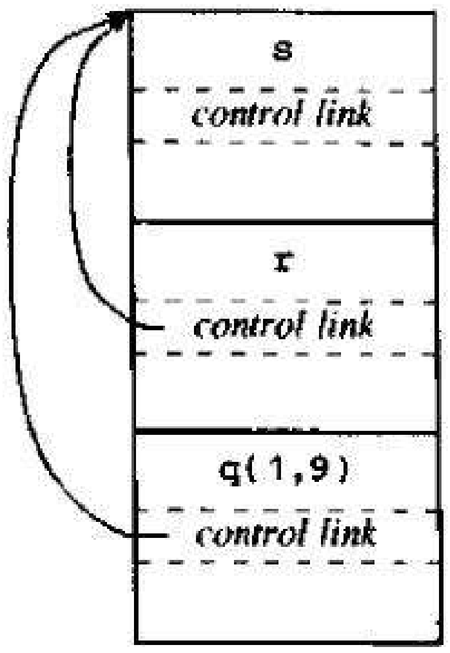
Static Allocation Limitations

1. Size of all data objects must be known at compile time.
2. Do not support true recursion.
3. Data structures can not be created dynamically.

Stack Allocation

Position in Activation Tree	Activation Records on the Stack	Remarks
s		Frame for s
		r is activated
		Frame for r has been popped and q(1,9) pushed
	 s a : array q(1,9) i : integer q(1,3) i : integer	Control has just returned to q(1,3)

Heap Allocation

Position in the Activation Tree	Activation Records in the Heap	Remarks
	 s control link r control link q(1,9) control link	Retained activation record for r

Parameters

- Actual parameter
- Formal parameter

```
.
main(){
.
 int result= fact(5);
.
}

int fact(n){
.
.
}
```

Parameter passing

- Call by value

  When we pass a parameter by value, it just copies the value within the function parameter and whatever is done with that variable within the function that doesn't reflect the original variable in caller function

- Call by reference/ call by address/ call by location

  When we send the parameters by reference, it copies the address of the variable which means whatever we do with the variables within the function, is actually done at the original memory location of the variable. Thus it also changes value of the variable in caller function.

- Copy/restore

  Call by copy-restore is similar to call by reference except that the values of actual parameters are changed when the called function ends, unlike call by reference

- Call by name

  In call by name formal parameters will be evaluated when they are going to be used inside function body, not during function call.

Symbol Tables

- A compiler uses a symbol table to keep track of scope and binding information of names.
- Symbol table is searched every time a name is encountered in the source text.
- Changes to the table occur if a new name or new information about an existing name discovered.

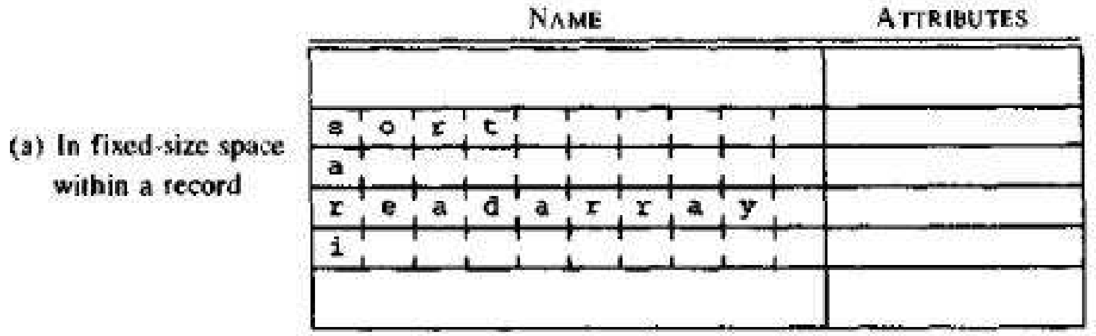
Symbol table implementation

- Linear list
  - Array
  - Linked list
- Hash table

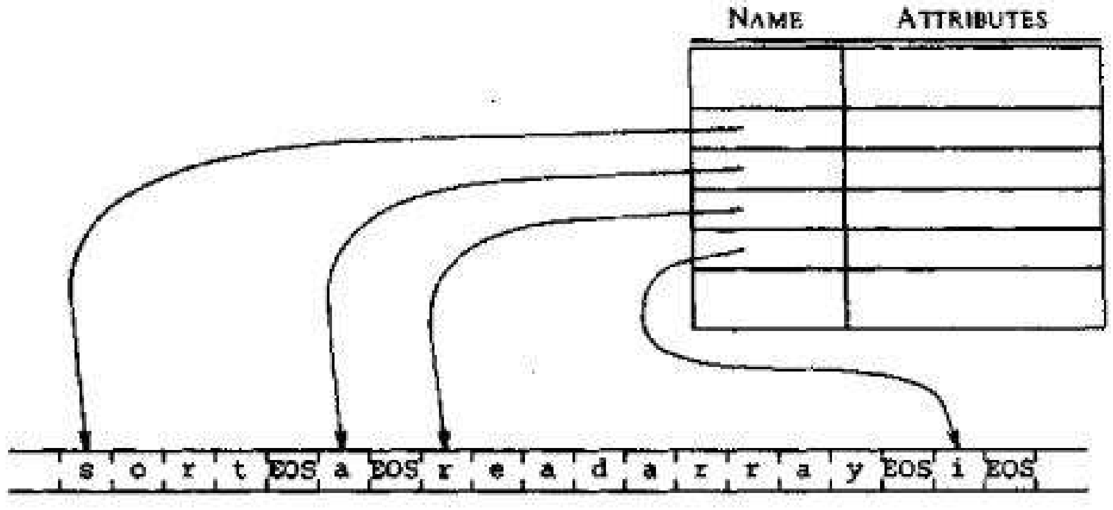
Symbol table entries

- Each entry in the table is for a name
- Does not have to be uniform
- Keywords and system function names will be added initially
- Other name may added whenever encountered first time.

Names in symbol table



(a) In fixed-size space within a record



(b) In a separate array

Dynamic memory allocation

- **Explicit allocation** - Explicit memory allocation occurs through pointers, and calls to the memory management functions.

- Implicit allocation - Implicit memory allocation is memory that is allocated, typically on the system stack, by the compiler. This occurs when you create a new variable:

- Dangling reference - A **dangling reference** is a **reference** to an object that no longer exists.

- Garbage/ Garbage Collection - Garbage is an object that cannot be reached through a **reference**. **Dangling references** do not exist in garbage collected languages because objects are only reclaimed when they are no longer accessible (only garbage is collected).