Unit-IV

Code Generation

Runtime Environment :-

A Runtime environment is a state of the target machine, which may include software libraries, environment variables etc; to provide services to the processes running in the system. It takes care of memory allocation and deallocation while the program is being executed.

Storage Organization :-

The target program runs in its own logical address space.

The logical address space is shared between compiler, operating s/m and target machine.

Code — The size of program

Static — The size of data objects

Heap — The size of program in runtime

↓

Free Memory

↑

Stack — Activation record

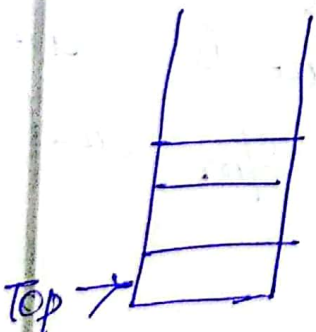
Static – If allocation decision can be made by the compiler looking at text of the program.

Dynamic – Decision can be made only when the program is running.

* Stack
* Heap.

① Storage Stack Allocation of Space:

Compilers uses procedures, functions (or) methods as units of user defined actions.




```
main()
{
function call();
}
void fun()
{
...
}
```

Activation:

The execution flow of procedure is called activation.

Activation Record:

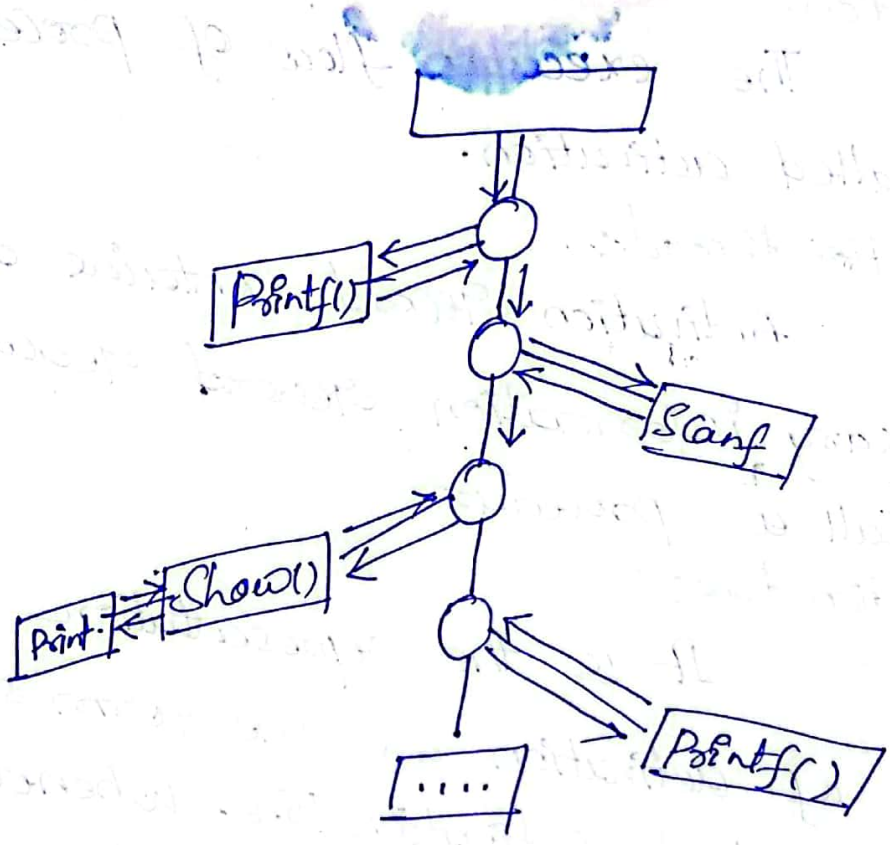
Activation record contains all the necessary information required to call a procedure.

Activation trees:

It is the representation of series of activation in the form of a tree is called Activation Tree. Whenever a procedure is executed its activation record is stored on the stack, known as control stack.

Ex:

```
Printf("Enter name");
Scanf("%s", username);
Show(username);
Printf("Enter any key:");
....
void show(char * user)
{
Printf("your name %s", user);
}
```



1. The Sequence of procedure calles corresponds to preorder traversal.
2. The Sequence of procedure calls corresponds to Postorder traversal.

Activation Record

Actual Parameters

Returned values

Control link

Access link

Saved machine states

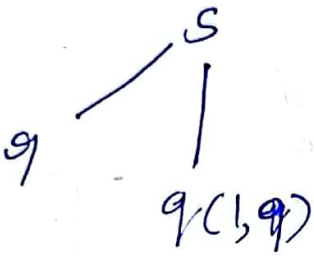
Local data

Temporaries.

Position in Activation Tree

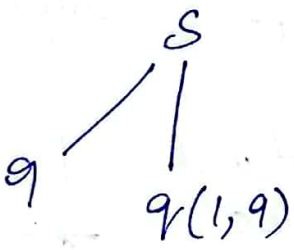
Activation Record on Stack

S

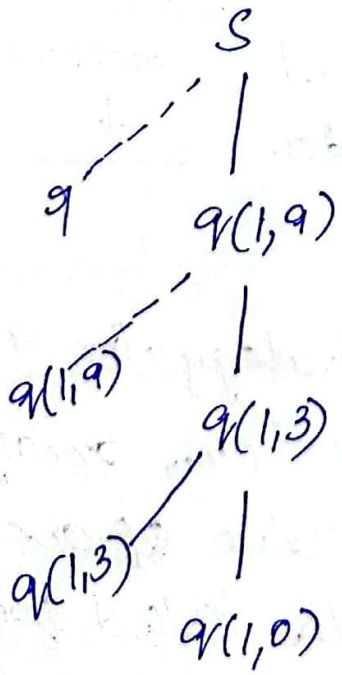


S
a:array

S
a:array
q
i=integer



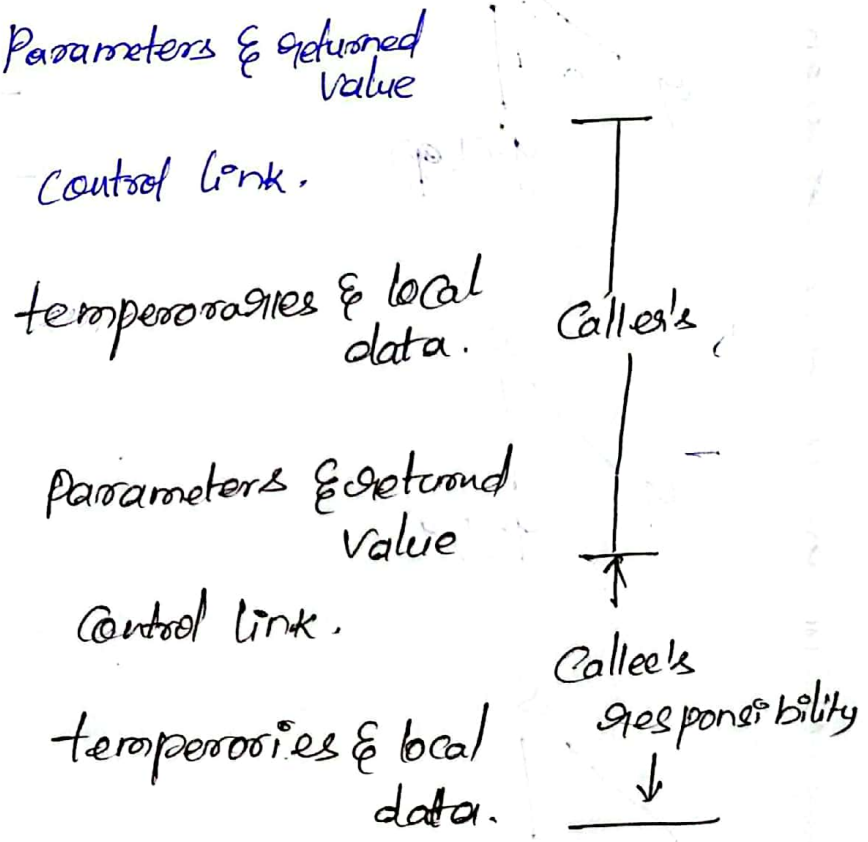
S
a:array
q(1,9)
i=integer



S
a:array
q(1,9)
i:integer
q(1,3)
i:integer
q(1,0)
i:integer.

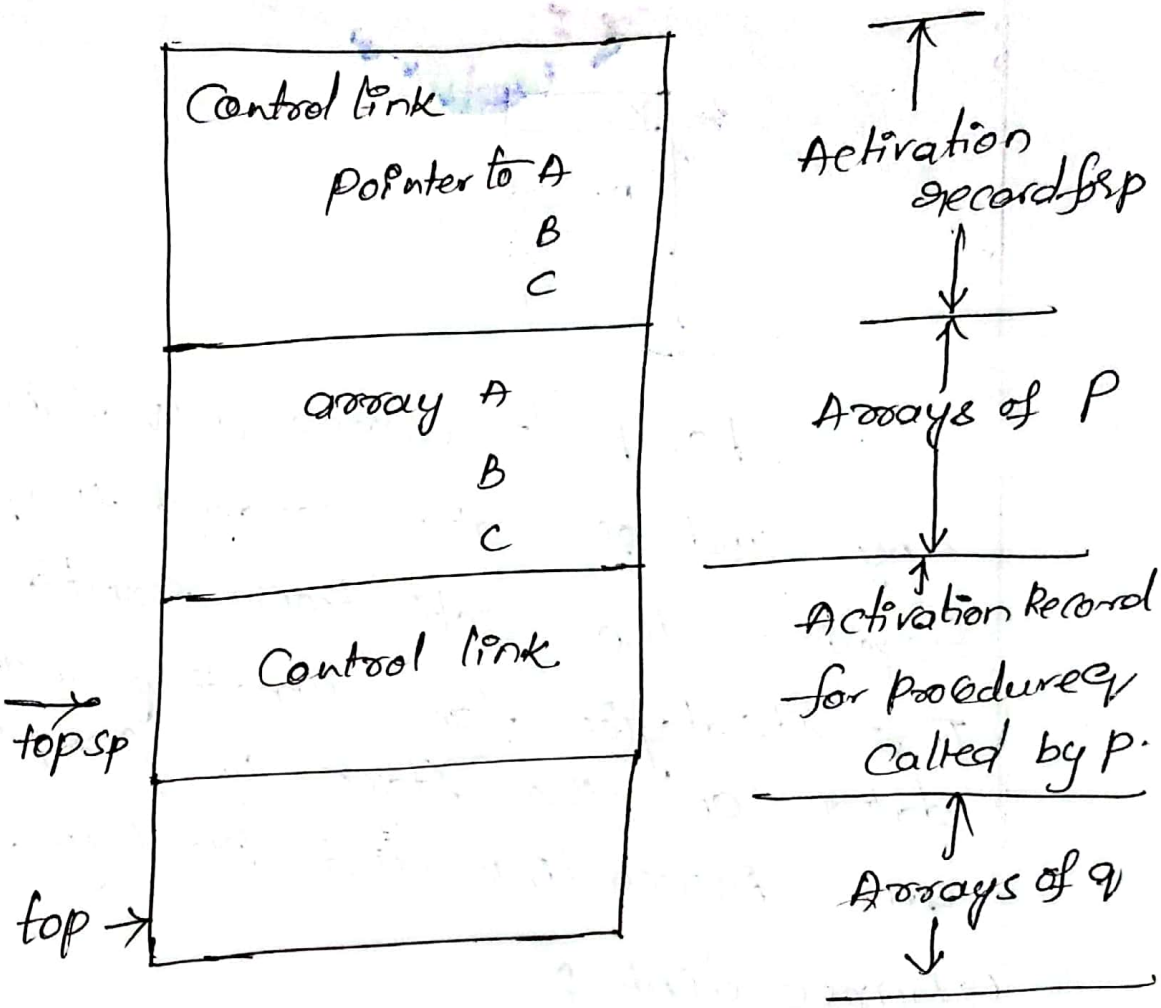
Call Sequences:

Procedure calls are implemented by what are known as Calling Sequence which consist of code that allocates an activation record on the stack and enters information into its fields.



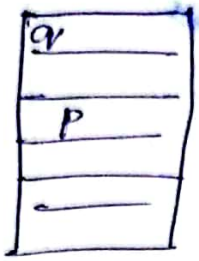
Variable length Data:

A common strategy for allocating variable length arrays (ie., arrays whose size depends on the value of one (or more parameters of the called procedure) and it has a size that depends on the parameters of the call.



(2) Access to Non local Data on Stack:

When a language allows nested procedures, access becomes difficult. For example, in the figure below, p is inside q. So at compile time, it is impossible to determine the position of each activation record. If either of p and q (or) both are recursive, then there will be a series of activation records.



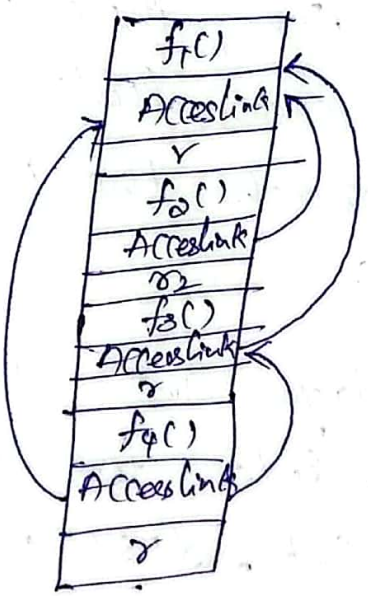
For p to access a variable (or) any kind of data declared in q, it requires the activation runtime information. Two strategies for accessing non local data on the stack in compiler design are Access link and Displays.

Access link :

Access link in one activation record are links that are used to refer to the non local data in other activation records and provide access to it. A chain of access links is formed on the top of stack and represent the program scope (or) static structure.

The chain a sequence of forth activations with progressively lowering depth.

The function currently being executed is at the end of chain and has access to data and procedures of all activations




```
f1()
{
 f2()
 f3()
{
 f4()
}}
```

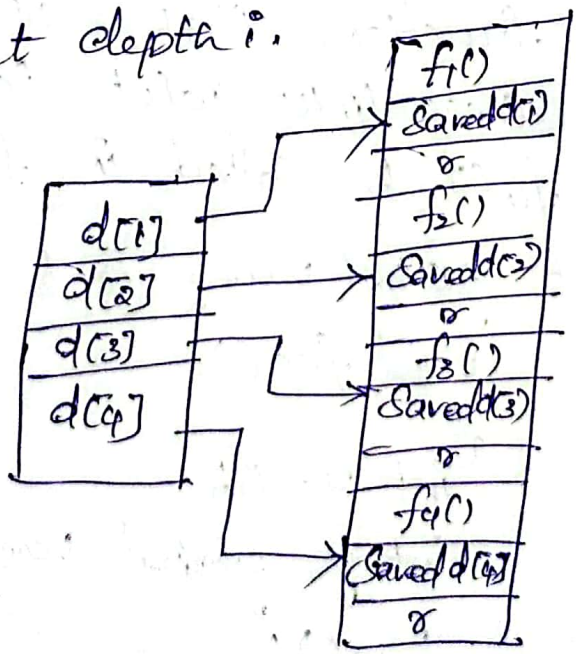
In this, $f_1()$ is the function calling all other functions that is why all other function's access links point to its access link. Then $f_4()$ is called in $f_3()$.

Ex:
```
int global var = 5;
void foo()
{int x=1;
 int y = global_var;
 bar();
}
void bar()
{int z=3
 z= x+z; }
```

Displays:

One of the issues of the access links is that if there are many nested calls of functions, there will be a very long chain access links that we must follow to reach the non-local data we need.

Displays are auxiliary arrays that contain a pointer for every nesting depth. At any point, d[i] is the pointer to the activation record which is highest in the stack for a procedure which is at depth i.



(3) Heap Management :—

Defination :—

Heap Management refers to the process of allocating and deallocating memory dynamically during program execution. Unlike stack memory, which automatically managed by a compiler, heap memory is managed automatically by the programmer.

Importance of Heap Management :

→ Heap memory provides flexibility as it allows for the allocation and deallocation of memory blocks at runtime, enabling creation of complex data structures.

→ Heap memory involves allocating memory blocks of varying sizes from the heap and deallocating them when they are no longer needed.

→ It requires tracking and organizing the allocated memory blocks to ensure efficient memory utilization and prevent memory leaks.

Advantages:

* Dynamic Memory Allocation: Heap memory enables the creation of dynamic data structures such as list, trees and graphs which are essential in many applications.
* Memory Reusability: Heap management facilitates the reuse of memory blocks that are no longer needed. This recycling of memory improves overall memory utilization and prevents memory fragmentation.
* Support Dynamic languages: Compilers Design for dynamic language like Python, JavaScript heavily relies on heap management.

Explicit Heap Management:

This technique requires the programmer to manualy allocation and deallocation of memory using specific functions (or) keywords provided by programming language Example includes malloc() and free. Explicit heap management gives the programmer full control over memory.

Ex:
```
int * ptr = malloc (sizeof(int));
if (ptr != NULL)
  {
*ptr = 40;
  printf("%d", *ptr);
  free (ptr);
  }
```

Implicit Heap Management :

Implicit heap management is performed automatically by the compiled (or) runtime without explicit instructions from the programmer. Garbage collection is common example of implicit heap management, where the runtime system periodically identifies and deallocates memory that is no longer reachable.

Ex:
```
my_list = [1,2,3]
print(my_list)
my_list = none
```

Hybrid Heap Management:—

Hybrid heap management combines elements of explicit and implicit techniques. It involves a combination of manual memory management and garbage collection.

```
Eg; List<Integer> list = new ArrayList<>();
    System.Out.println(list);
    list = null;
```

(4) Introduction to Garbage Collection:—

Garbage collection is the process of automatically recovering computer memory storage. A garbage collector consist of compiler and a runtime system. A mark and sweep collector (or) a copying collector are viable options, but even a conservative collector is acceptable.

Design Goals of Garbage Collectors:

Garbage collection is the process of restoring chunks of storage space that a program could no longer access. The Garbage Collector locates the inaccessible

objects and return them to memory manager which keeps track of the available space.

Requirements :

Type Safety : Not all languages are suitable for automatic garbage Collection. A garbage collector must determine whether every given data element (or) component of data element is (or) may be used as a pointer to a piece of allocated memory space to function correctly.

1. Garbage collection is important for avoiding fragmentation and making for the most available memory.
2. Garbage Collections are known for causing programs, the mutators, to halt for an unusually long time when garbage Collection occurs without warning.
3. Real time application are specific scenarios in which certain computations must be finished with in a certain amount of time.
4. We can't evaluate a garbage collectors speed based on its running time.

Characteristics of Garbage Collection:

* Garbage collectors outperform systems when it compared to Explicit deallocation.
* Automatic deallocation relieves a programmer of memory management concerns, improving system reliability and reducing development time and costs.
* It is required for functional languages
* It is generally so expensive that, it avoid memory leaks, many prominent languages have yet to implement it.
* It can take a long time and does not significantly increase an application's total execution time.

Algorithms:

1. Classical Algorithm
    1. Reference Counting.
    2. Mark & Sweep
    3. Mark-Compact
    4. Copying Collection.
    5. Non Copying Implicit
2. More Creative Algorithm:
    1. Incremental Tracing Collection
    2. Generational Collection.

⑤ Trace Based Collection :—

It allows the programmers to define and use efficient data structures on the fly, instead of being restricted to predefined types like arrays or linked lists. It runs periodically to find unreachable objects and reclaim their space.

Mark and Sweep collector :

It is a straight forward method, find all unreachable object and put them on the list of free space.

Algorithm :

1. add each obj referenced by the rootset to list unScanned and set its reached-bit to 1;
2. while (unScanned ≠ 0) {
3. remove some object O from unScanned;
4. for (each object O' referenced in O) {
5. if (O' is unreached) {
6. Set the reached bit of O' to 1;
7. put O' in unScanned
8. Free = ∅

9. for (each chunck of memory O in Heap) {
10. if (O is unreached, i.e., its reached bit is 0) add O to Free;
11. else set the reached bit of O to 0;

Free – holds objects known to free.
Unscanned – reached, but successors are not yet considered.

Mark and Compact Garbage Collector:

It uses two step process.
The first step is to mark all the live objects and second step is to compact all dead objects. It is faster than mark and sweep collector.

Copying Collectors:

The copying collectors moves all live objects to new location and then frees up the old space. The copying collector is very fast algorithm for moving data b/w memory ranges.

Incremental Collectors:

It is a powerful optimization technique that can help to improve performance for many classes of programs. The major benefit of this collection is that it allows the compiler to collect more data from the program than it would otherwise be able to do, without incurring any extra runtime overhead.

Partial Collectors:

The train / partial algorithm is used to collect garbage in generational fashion. It also can be used for both single and multiple object collectors.