Unit - IV

Run time Environments

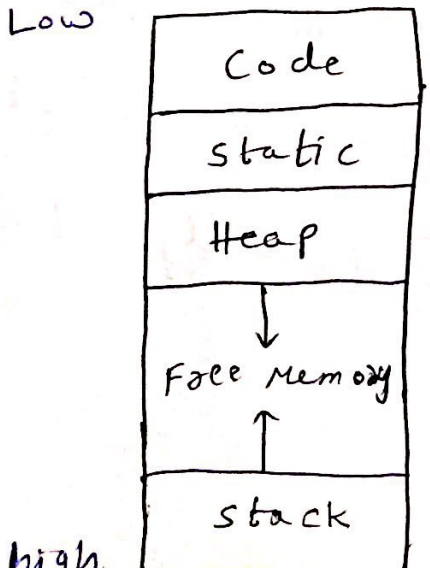
Storage organization

→ The management and orgnization of this logical address space is shared between the Compiler, operating system and target Machine.

→ The operating system maps the logical addresses into physical addresses, which are usually spread throughout Memory.

→ The runtime representation of an object program in the logical address space consists of data and program areas.

Typical Subdivision of run-time Memory into code and data areas



→ Runtime storage comes in blocks of contigous bytes, where a byte is the smallest unit of addressable Memory.

→ A byte is eight bits and four bytes form a machine word.

→ Multiple byte objects are stored in consecutive bytes and given the address of the first byte.

→ The storage layout for data objects is strongly influenced by the addressing constraints of the target Machine.

→ space left unused due to alignment considerations is referred to as padding.

→ The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area Code, (usually low end of memory)

→ Similary, the size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time and these data objects can be placed in another statically determined area called static.

→ One of the reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code.

→ To maximize the utilization of space at run time, the other two areas, stack and heap are at the opposite ends of remainder of address space.

→ These areas are dynamic, their size can change as the program executes.

→ These areas grow towards each other as needed.

→ The stack is used to store data structures called activation records that get generated during procedure calls.

→ stack grows towards lower addresses, heap towards higher.

Storage Allocation (Static vs Dynamic)

→ The layout and Allocation of data to Memory locations in the runtime environment are key issues in storage management.

→ storage allocation decision is static, if it can be made by compiler looking only at text of program, not at what program does when it executes.

→ storge allocation decision is dynamic, if it can be decided only while the program is running.

→ stack storage - Names local to a procedure are allocated space on stack. The stack supports the normal call/return policy for procedures.

→ Heap storage : Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage. The heap is area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.

storage Allocation strategies

→ There are three different type of storage allocation strategies based on the division of runtime storage. These are

(i) static Allocation - It is for all data objects at compile time

(ii) stack Allocation - In the stack allocation a stack is used to manage runtime storage

(iii) Heap Allocation - In heap allocation the heap is used to manage dynamic memory allocation.

(i) static Allocation

→ The size of data objets is known at compile time. The names of these objects

are bound to storage at compile time only and such an allocation of data objects is done by static Allocation.

→ The binding of name with the amount of storage allocated donot change at run time.

→ In static Allocation the compiler can determine the amount of storage required by each data object. Therefore it becomes easy for a compiler to find the addresses of these data in the activation record.

→ At compiler time compiler can fill the addresses at which the target code can find the data it operates on.

→ FORTRAN uses static Allocation strategy.

Limitations of static Allocation

→ The static allocation can be done only if the size of data object is known at compile time.

→ The data structures can not be created dynamically. The static allocation can not manage the allocation of memory at runtime.

→ Recursive procedures are not supported by this type of allocation.

(ii) Stack Allocation

→ Stack Allocation strategy is a strategy in which the storage is organized as stack. This stack is also called control stack.

→ As activation begins, the activation records are pushed onto the stack and on completion of this activation record the corresponding activation records are popped.

→ The locals are stored in each activation record. Hence locals are bound to corresponding activation record on each fresh activation.

→ The data structures can be created dynamically for stack allocation.

Limitations of stack allocation

→ The memory addressing can be done using pointers and index registers. Hence this type of allocation is slower than static Allocation.

Heap Allocation

→ If the values of non-local variables must be retained even after the activation record then such a retaining is not possible by stack allocation. This limitation of stack allocation is because of its last In First out nature. For retaining of such local variables Heap Allocation strategy is used.

→ The Heap allocation allocates the continous block of memory when required for storage of activation records (or other data object.

→ This allocated memory can be deallocated when activation ends.

→ The deallocated (free) space can be further reused by heap manager.

→ The efficient heap Management can be done by

(i) creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.

(ii) Allocate the most suitable block of memory from the Linked list (i.e) use best fit technique for allocation of block.

Stack Allocation of space

→ ~~All~~ → Almost all compilers for Languages that use procedures, functions (or) methods as units of user-defined actions manage atleast part of their runtime memory as stack.

→ Each time a procedure is called, space for its local variables is pushed onto a stack, and when procedure terminates, that space is popped off the stack

Activation Trees

→ stack allocation would not be feasible if procedure calls (or) activations of procedures, did not nest in time.

Eg: quick sort (recursive)

```
int   a[11];

void  readArray() { /* Read 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] so that a[m..p-1] are less than
v, a[p] = v, and a[p+1..n] are 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);
}
```

→ Sequence of calls that might result from an execution of the program.

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

→ In this example, as it is true in general, procedure activations are nested in time.

→ If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end.

There are three common cases:

(i) The activation of q terminates normally. Then the control resumes just after the point of p at which the call to q was made.

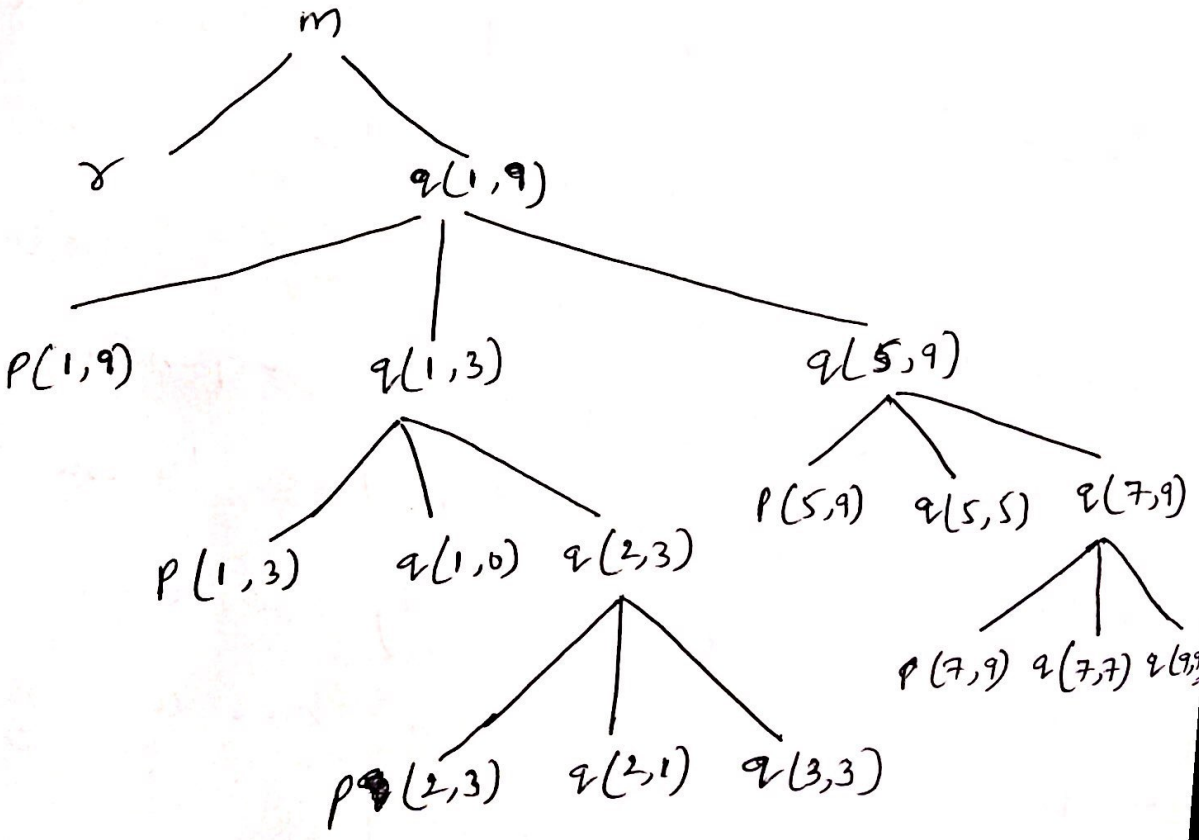
(ii) The activation of q, or some procedure q called, either directly or indirectly, aborts i.e it becomes impossible for execution to continue. In that case, p ends simultaneously with q.

(iii) The activation of q terminates because of an exception that q cannot handle. Procedure p may handle the exception, in which case the activation of q has terminated while the activation of p continues.

→ We represent the activations of procedures during the running of entire program by a tree, called Activation Tree.

→ Each node corresponds to one activation, the root is the activation of the "main" procedure that initiates execution of program.

→ At a node for an activation of procedure P, the children correspond to activations of the procedures called by this activation of P.



Activation tree representing calls during an execution

→ The sequence of procedure calls correspond to a preorder traversal of the activation tree

→ The sequence of returns corresponds to a postorder traversal of the activation tree

Activation Records

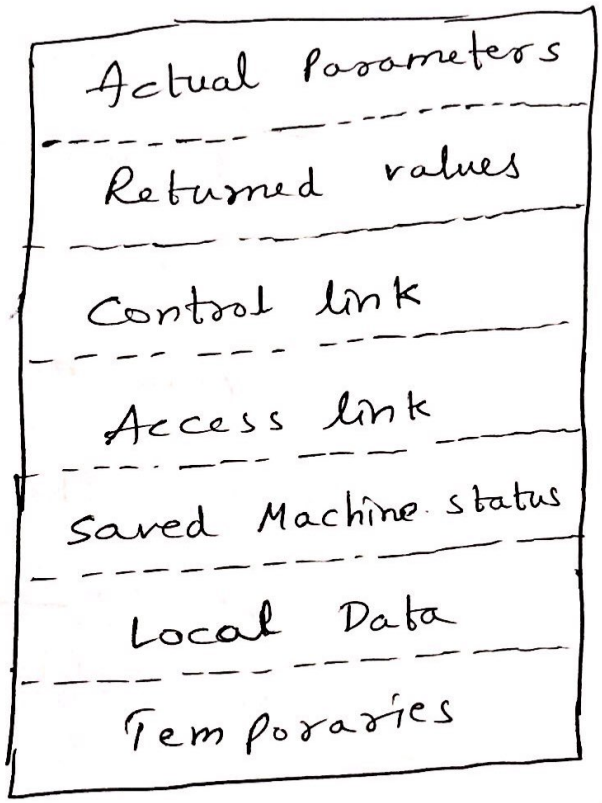
⇒ procedure calls and returns are usually managed by a runtime stack called the control stack.

→ Each live activation has an activation Record on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.

→ The latter activation has its record at the top of the stack.

→ The Activation Record is a block of memory used for managing information needed by a single execution of a procedure

→ The contents of activation records vary with the language being implemented.



A general Activation Record

→ Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

→ Local data belonging to the procedure whose activation record this is.

→ A saved Machine status, with information about the state of the machine just before the call to procedure. This information typically includes the return address and the contents of registers that were used by calling procedure and that must be restored when the return occurs.

→ An "access link" may be needed to locate data needed by the called procedure. This field refers to nonlocal data in another activation record. This field is also called static link field.

→ A "control link", pointing to the activation Record of the caller. It points to activation record of calling procedure. This link is also called dynamic link.

→ space for return value of the called function

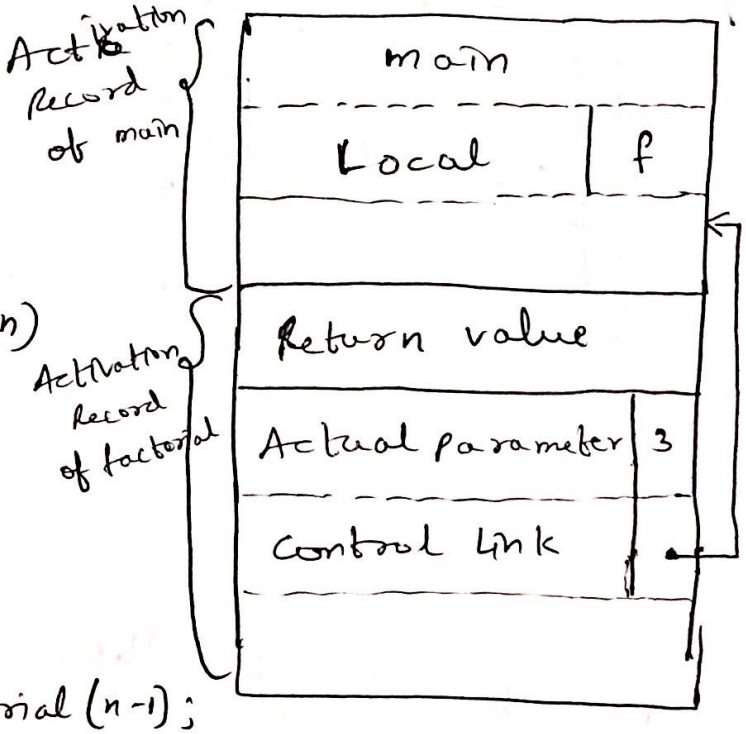
→ The actual parameters used by the calling procedure.

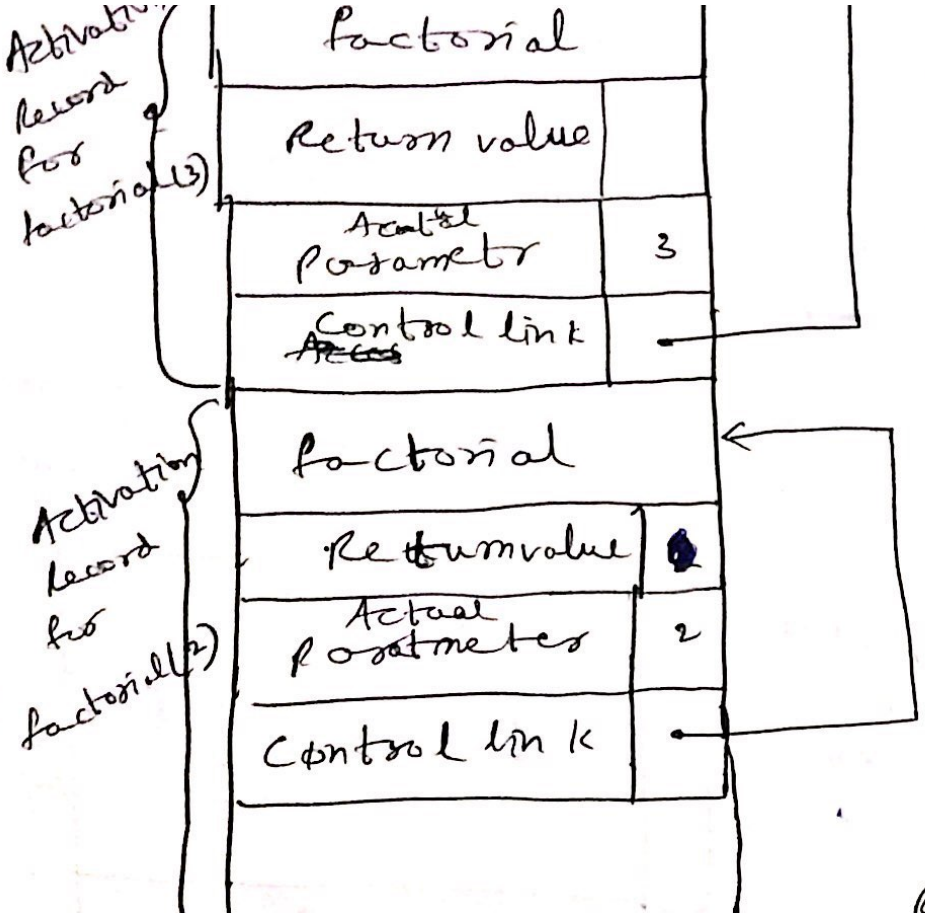
→ The size of each field of activation record is determined at the time when a procedure is called.

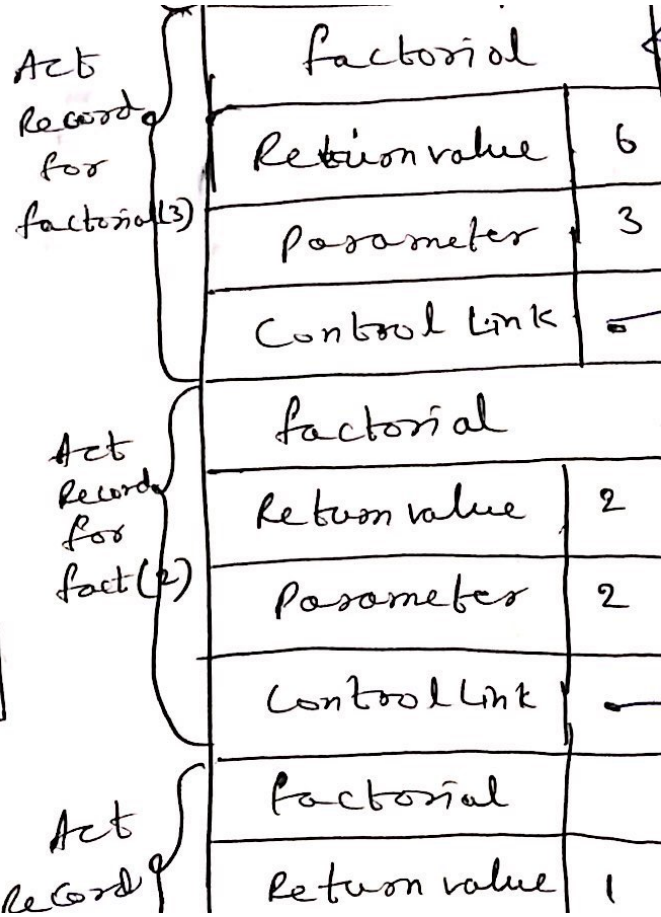
Eg:-

```
main()
{
int f;
f= factorial(3);
}
int factorial(int n)
{
if (n==1)
 return 1;
else
 return n* factorial(n-1);
}
```

First call Activation Record







Eg:- Suppose a procedure A calls the procedure B (callee procedure). ~~begins after the array~~

→ Procedure A has two arrays x & y.

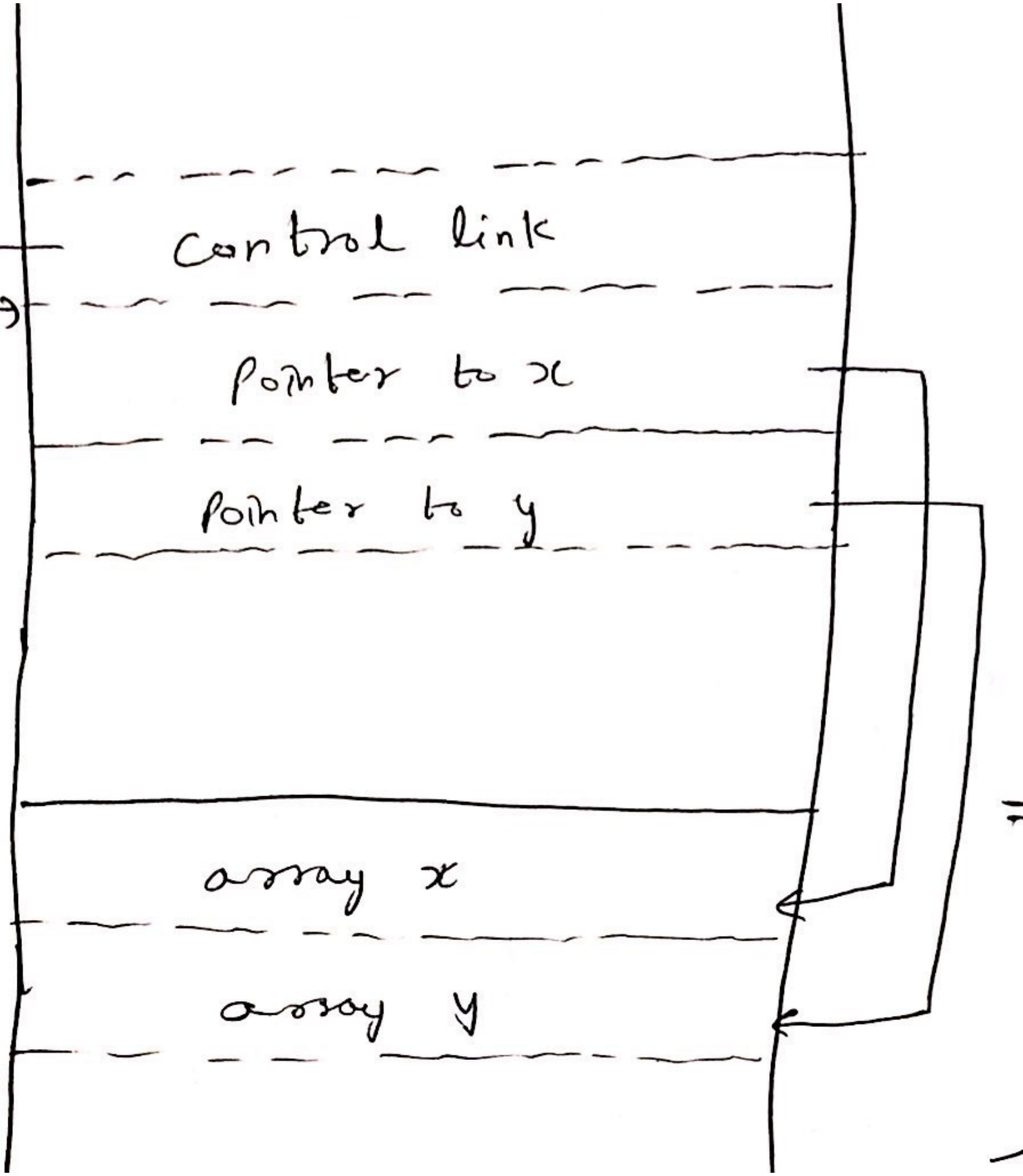
The storage of these arrays is not the part of activation ~~record~~ record of A.

→ In the activation record of A only the pointers to the beginning of x and y are appearing we can obtain the relative addresses of these arrays at compile time.

→ The activation Record ~~of~~ for procedure B begins after the arrays of A. Suppose the procedure B has variable length arrays p and q. Then after the activation record of B the arrays for procedure B can be placed.

→ Two pointers can be maintained top and top-sp to keep track of these records.

→ The top points to actual top of the stack and top-sp points to end of some field in the activation record.



Access to Non-Local Data on the stack

→ The storage Allocation can be done for two types of data variables.

(i) Local data

(ii) Non-Local data

→ The local data can be handled using activation record whereas non local data can be handled using scope information

→ The block structured storage allocation can be done using static scope or) lexical scope.

→ The non block structured storage allocation can be done using dynamic Scope.

Local data

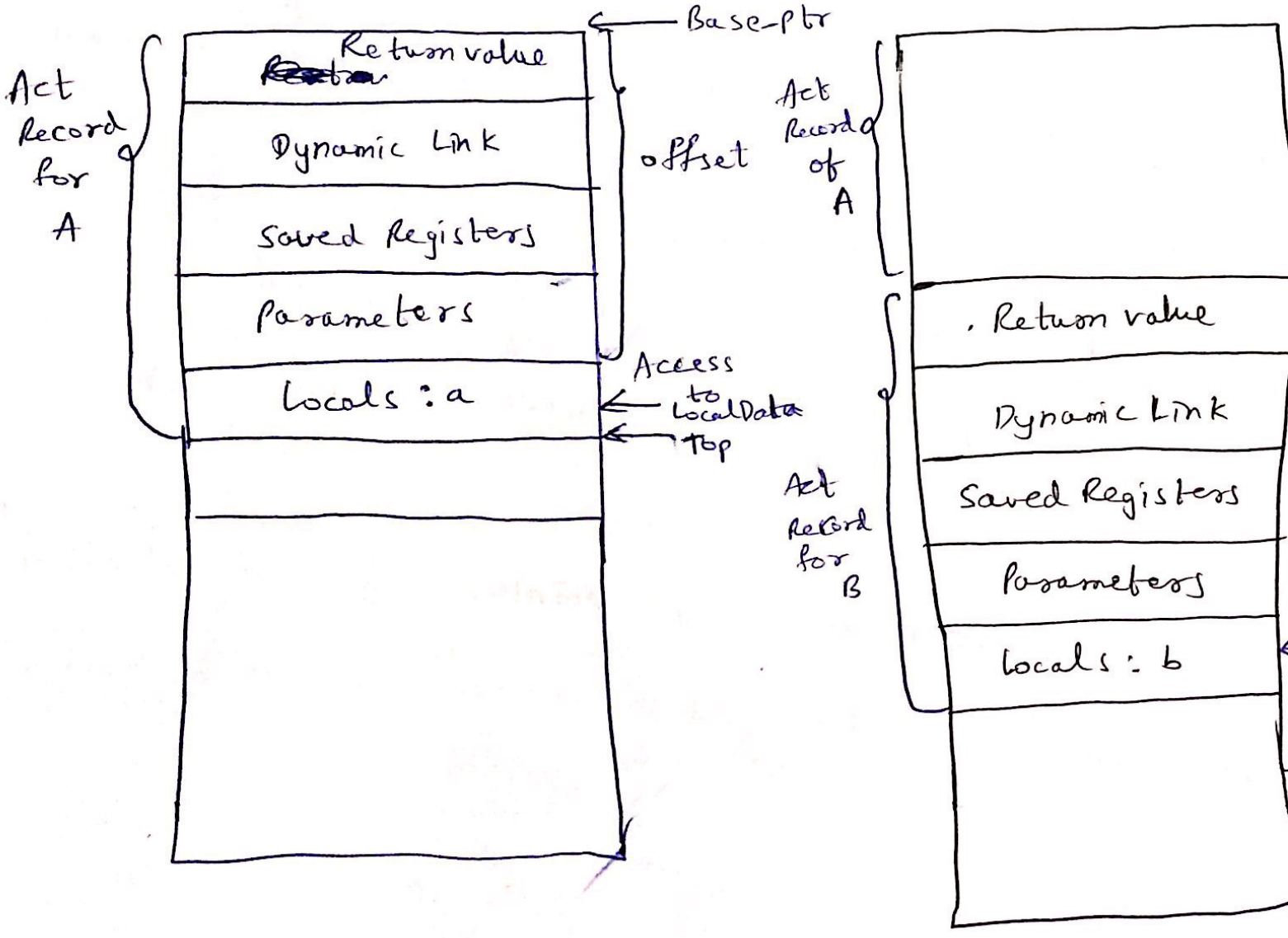
→ The Local data can be accessed with the help of activation record.

→ The offset relative to base pointer of an activation record points to local data variables within activation record.

Eg :-

```
Procedure A
    int a;
    Procedure B
        int b;
        body of B;
    body of A;
```

The contents of stack along with base pointer a offset

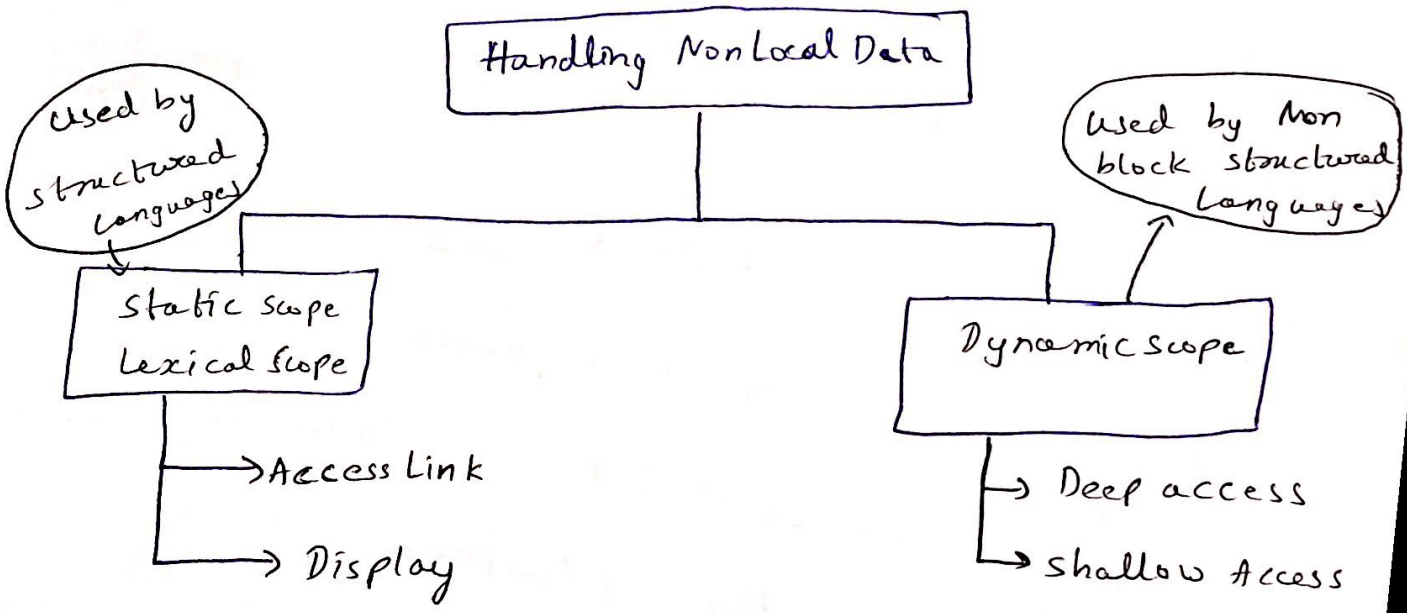


Access to Non Local Names

→ A procedure may sometimes refer to variables which are not local for it. such variables are called as Non local variables.

→ For the Non-Local Names there are two types of scope rules that can be defined

(i) static
(ii) dynamic



(i) Static Scope Rule

→ It is also called as lexical scope. In this type the scope is determined by examining the program text.

→ PASCAL, C and ADA use static scope rule.

→ These Languages are also called as Block Structured Languages.

(ii) Dynamic Scope rule

→ For non block structured Languages this dynamic scope allocation rules are used.

→ The dynamic scope rule determines the scope of declaration of names at runtime by considering the current activation.

Eg: LISP and SNOBOL use Dynamic Scope rule.

Static Scope (or) Lexical Scope

→ Block :- It is a sequence of statements containing the Local data declarations and enclosed with in the delimiters.

Eg:
```
{
Declaration statements;
.....
}
```

→ The delimiters mark the beginning and end of the Block. The Blocks can be in nesting fashion that

means block B_2 completely can be inside the Block B_1.

→ The scope of declaration in a Block structured Language is given by most closely nested loop (or) static rule.

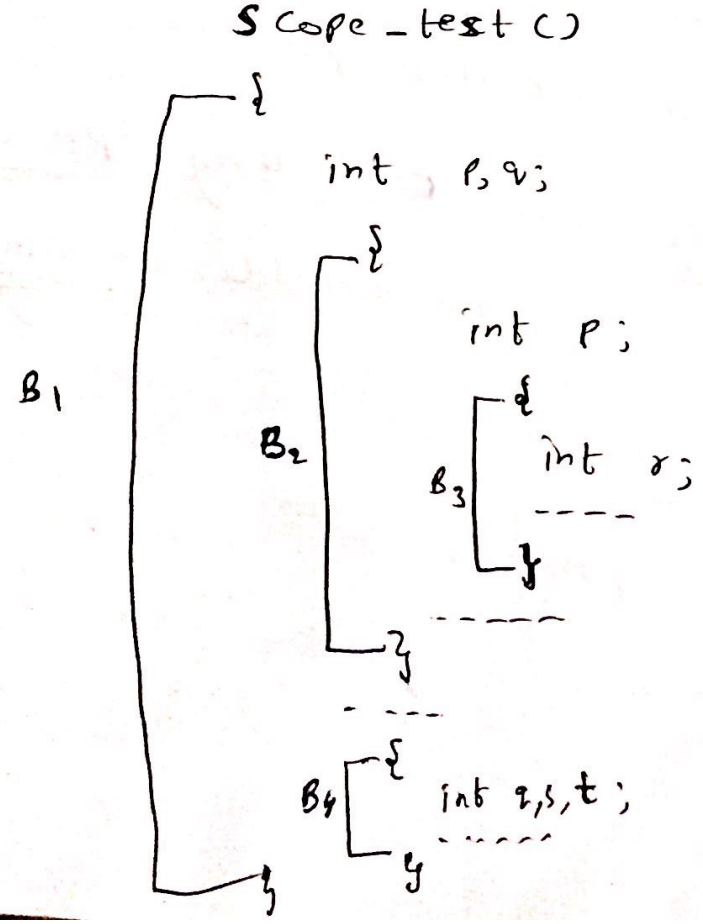
→ The declarations are visible at a program point are:

(i) The declarations that are made locally in the procedure.

(ii) The names of all enclosing procedures

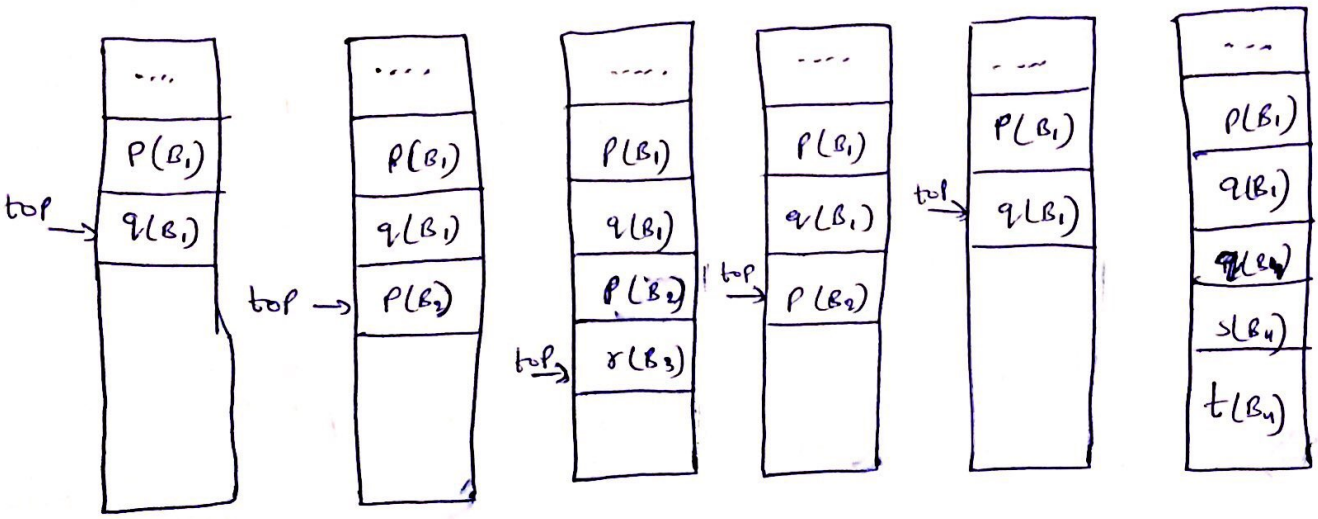
(iii) The declarations of names made immediately within such procedures.

Eg:-



→ The storage can be allocated for a complete procedure body at one time.

→ The storage for the names corresponding to particular block can be as shown below



Lexical Scope for Nested procedure

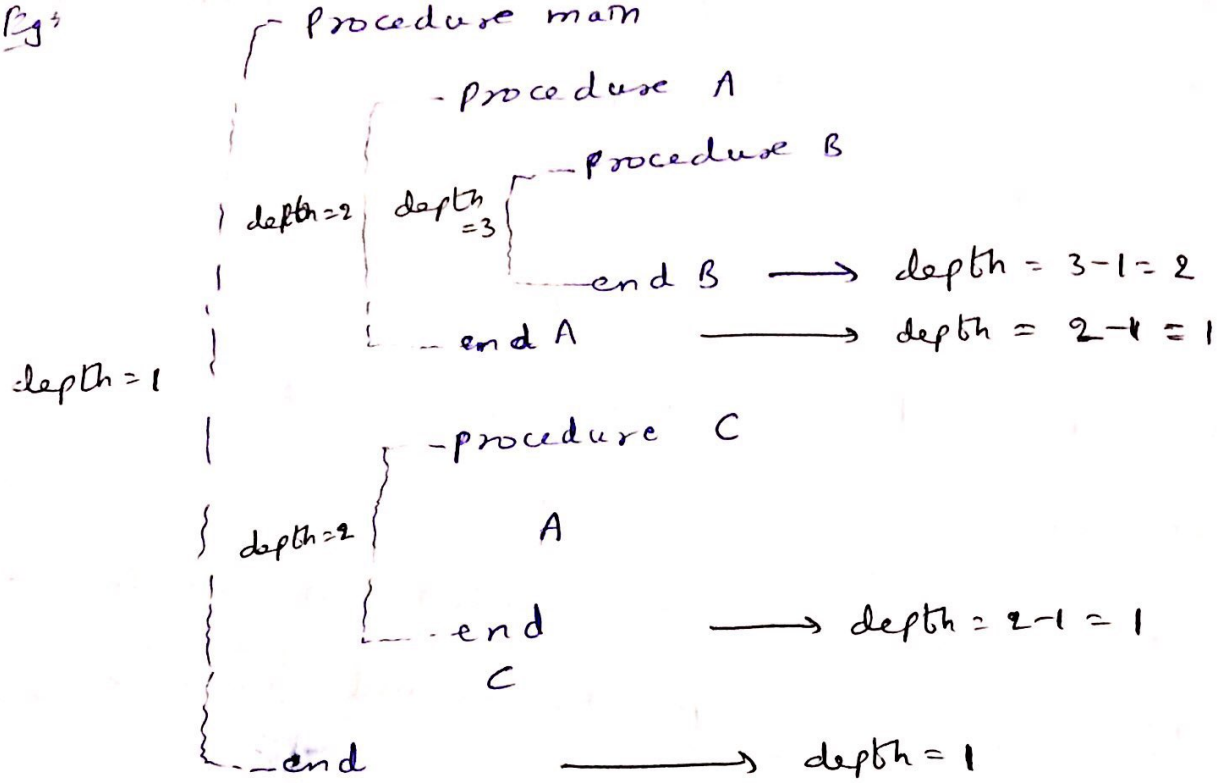
→ Nested procedure is a procedure that can be declared within another procedure.

→ A procedure Pi, can call any procedure that is its direct ancestor or older siblings of its direct ancestor.

→ The Nested procedures can be as shown

```
Procedure main
Procedure P1
    //    P2
    //    P3
    //    P4
```

Eg:



→ Nesting depth - Nesting depth of a procedure is used to implement lexical scope. The Nesting depth can be calculated as follows:

(i) The nesting depth of main program is 1

(ii) Add 1 to depth each time when a new procedure begins

(iii) Subtract 1 from depth each time when you exit from a nesting procedure.

(iv) The variable declared in specific procedure is associated with nesting depth.

→ The lexical scope can be implemented using Access links and Displays

Access Link

→ The implementation of lexical scope can be obtained by using pointer to each activation record.

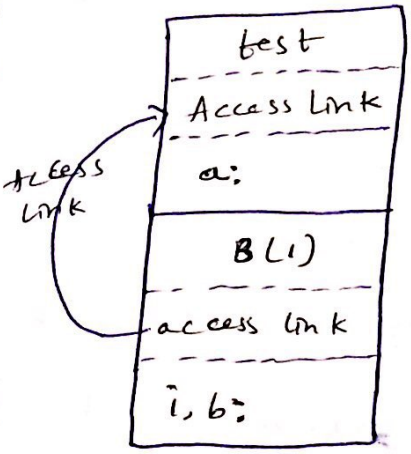
→ These pointers are called Access links.

** If a procedure p is nested within a procedure q then access link of p points to access link of most recent activation record of procedure q.

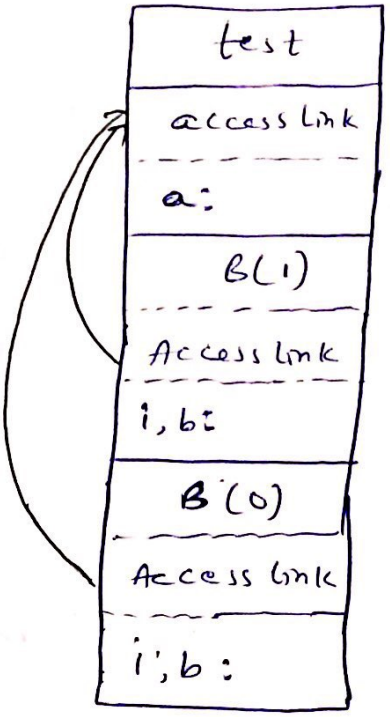
Eg:

```
program test;
    var a: int;
    procedure A;
        var d: int;
        begin a:=1, end;
    procedure B (i: int);
        var b: int;
        procedure C;
            var k: int;
            begin A; end;
        begin
```

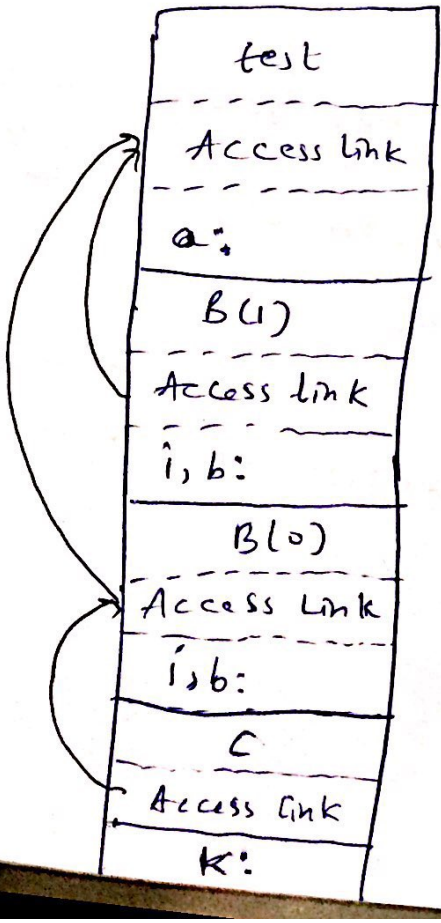
```
if (i<>0) then B(i-1)
else C;
end
begin B(1); end;
```



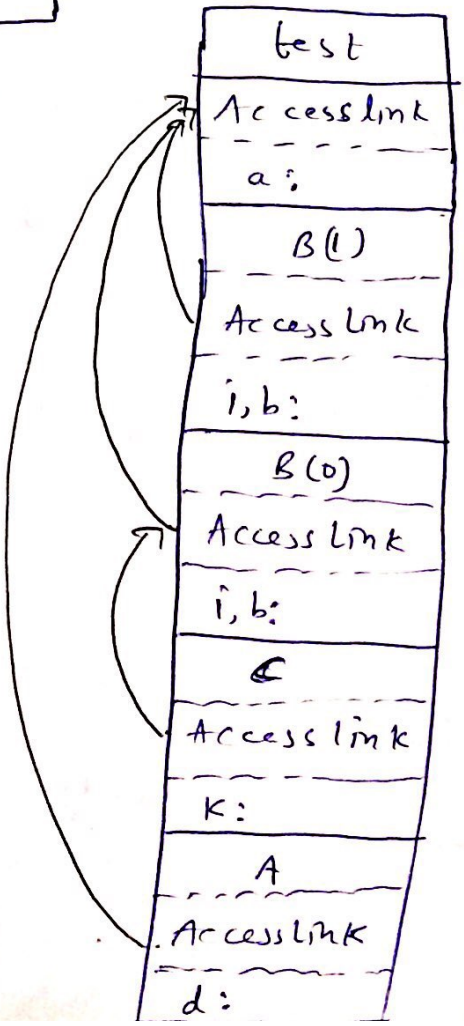
(a)



(b)



(c)



→ To set up the access links at compile time:

(i) if procedure A at depth n_A calls procedure B at depth n_B then

case 1: if $n_A < n_B$, then B is enclosed in A and $n_A = n_B - 1$

case 2: if $n_A \geq n_B$, then it is either a recursive call or calling a previously declared procedure.

(ii) The access link of activation record of procedure A points to activation record of procedure B where the procedure B has procedure A nested within it.

(iii) The activation record for B must be active at the time of pointing.

(iv) If there are several activation records for B then the most recent activation record will be pointed.

→ Thus by traversing the access links, Non Locals can be correctly accessed.

A simple Code Generator

→ The algorithm that generates code for a single basic block.

→ It considers each three address instruction in turn, and keeps track of what values are in what registers so it can avoid generating unnecessary loads and stores.

→ Primary issue during code Generation is deciding how to use registers to best advantage.

→ The four principal uses of registers.

(i) In most Machine architectures, some (or) all the operands of an operation must be in registers in order to perform the operation.

(ii) Registers make good temporaries - places to hold the result of subexpression while a larger expression is being evaluated.

(iii) Registers are used to hold (global) ~~variables~~ values that are computed in one basic block and used in other blocks.

(iv) Registers are often used to help with runtime storage management.

Eg:- To manage Runtime stack

→ The algorithm assumes that some set of registers are available to hold the ~~variable~~ values that are used in the block.

→ Typically, this set of registers doesnot include all registers of the machine, since some registers are reserved for global variables and managing the stack.

Register and Address Descriptors

→ Code generation Algorithm considers each Three-Address Instruction in turn and decides what load are necessary to

get the needed operands into registers.

→ After generating the loads, it generates operation itself.

→ Then, if there is a need to store the result into a memory location, it also generates that store.

→ In order to make needed decisions, we require data structure that tells us what program variables currently have their value in a register, and which register can registers, if so.

→ The data structure has following descriptors

(i) For each available register, a Register Descriptor keeps track of the variable names whose current value is in that register.

(ii) For each program variable, an Address descriptor keeps track of location (or) locations where the current value of that variable can be ~~found~~ found.

→ The location might be a register, a memory address, a stack location, (or) some set of more than one of these.

The Code Generation Algorithm

→ An essential part of the algorithm is a function getReg(I), which selects registers for each memory location associated with the three address instruction-(I).

→ Function getReg has access to register and address descriptors for all the variables of basic blocks, and may also have access to certain useful data-flow information such as variables that are live on exit from the block.

→ In a three address instruction such as $x = y + z$, we shall treat '+' as Generic operator and ADD as equivalent machine instruction.

Machine Instructions for operations

→ For a three-address instruction such as $x = y + z$

(i) Use getReg ($x = y + z$) to select registers for x, y and z. Call them R_x, R_y, R_z

(ii) If y is not in R_y (according to register descriptor R_y), then issue an instruction LD R_y, y' where y' is one of memory locations for y.

(iii) Similary, if z is not in R_z, issue an instruction LD R_z, z' where z' is a location for z.

(iv) Issue the instruction ADD R_x, R_y, R_z

Machine Instructions for copy statements

→ Three address copy statement of the form $x = y$.

→ Assume getReg($x = y$) will always choose same register for both x and y.

→ If y is not already in that register R_y, then generate the machine instruction LD R_y, y.

→ If y was already in R_y, we do nothing

→ It is also necessary that we adjust the ~~address~~ register descriptor for R_y so that it includes x as one of the values found there.

Managing Register and Address Descriptors

→ As code generation algorithm issues load, store and other machine instructions, it needs to update the register and address descriptors.

The rules are as follows:

(i) For the instruction LD R, x

(a) change the register Descriptor for register R so it holds only x.

(b) change the address descriptor for x by adding register R as an additional location.

(ii) For the instruction ST x, R

change the address descriptor for x to include its own memory location.

(iii) For an operation such as ADD R_x, R_y, R_z

(a) change the register descriptor for R_x so that it holds only x

(b) change the address descriptor for x so that its only location is R_x. Note that the memory location for x is not now in address descriptor for x.

(c) Remove R_x from the address descriptor

of any variable other than x.

(iv) When we process a copy statement x = y, after generating the load for y into register R_y, and after managing descriptors as for all load statements (Rule 1)

(a) Add x to register descriptor for R_y

(b) Change the address descriptor for x so that its only location is R_y.

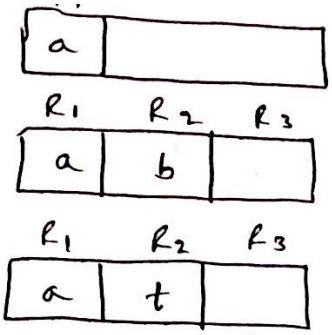
Example

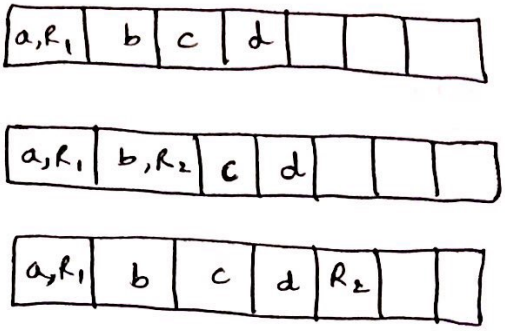
```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

Code generated

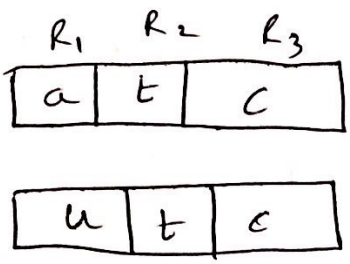
```
LD R1, a
LD R2, b
SUB R2, R1, R2
LD R3, c
SUB R1, R1, R3
ADD R3, R2, R1
LD R2, d
ADD R1, R3, R1
exit
ST a, R2
ST d, R1
```

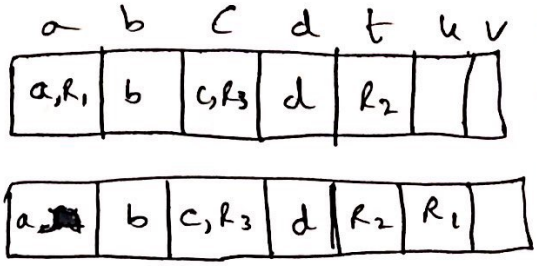
```
LD R1, a
LD R2, b
SUB R2, R1, R2
```



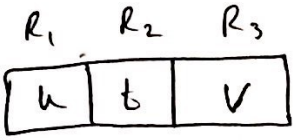


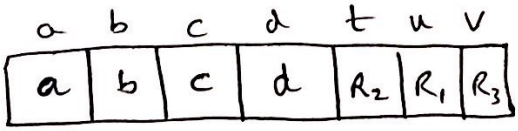

```
u = a - c
LD R3, c
SUB R1, R1, R3
```



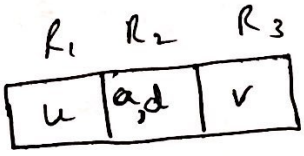


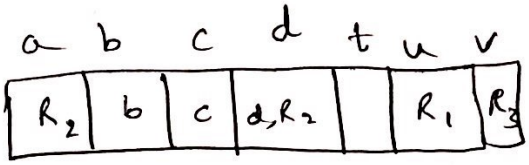

```
v = t + u
ADD R3, R2, R1
```



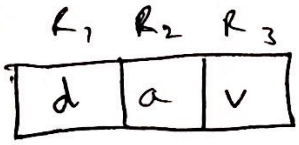


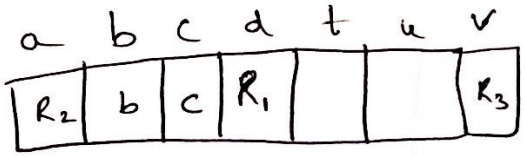

```
a = d
LD R2, d
```





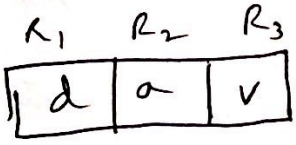

```
d = v + u
ADD R1, R3, R1
```

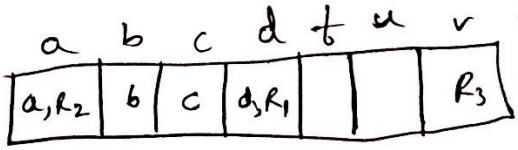




exit

```
ST a, R2
ST d, R1
```





Design of function getReg

→ Eg:- $x = y + z$

→ First we pick a register for y and register for z.

→ We shall concentrate on picking register R_y for y. The rules are as follows:

(i) if y is currently in a register, pick a register already containing y as R_y. **(Dont issue load instruction).

(ii) If y is not in a register, but there is a register that is currently empty, pick one such register as R_y.

(iii) The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse.

→ Let R be a Candidate Register and suppose V is one of the variables that the register

descriptor for R says is in R.

→ we need to make sure that v's value either is not really needed or that there is somewhere else we can go to get the value of R.

The possibilities are:

(a) if the address descriptor for v says that v is somewhere besides R, then we are (ok).

(b) if v is x, the value being computed by Instruction I, and x is not also one of the other operands of Instruction I (z in this example), then we are [ok]. The reason is that in this case, we know this value of x is never going to be used again, so we are free to ignore it.

(c) Otherwise, if v is not used later, then we are [ok].

(d) If we are not ok by one of the first two cases, then we need to generate the store instructions ST v,R to place a copy of v in its own memory location. This operation is called a spill.

→ Now consider the selection of register R_x. The issues and options are almost as for y. So we shall mention only the differences.

1) Since a new value of x is being computed, a register that holds only x is always acceptable choice for R_x. This statement holds even if x is one of y and z, since our machine instructions allows two registers to be the same in one instruction.

2) If y is not used after instruction I, in the sense described for variable v in (3c) and R_y holds only y after being loaded

Statement	Code Generation	Register Descriptor
$t_1 = a - b$	LD R_0, a SUB R_0, R_0, b	R_0 contains t_1
$t_2 = a - c$	LD R_1, a SUB R_1, R_1, c	R_0 contains t_1 R_1 contains t_2
$t_3 = t_1 + t_2$	ADD R_0, R_0, R_1	R_0 contains t_3 R_1 contains t_2
$d = t_3 + t_2$	ADD R_0, R_1, R_0	R_0 contains d

Register Allocation and Assignment

→ Instructions involving only register operands are faster than those involving memory operands.

→ Therefore, efficient utilization of registers is vitally important in generating good code.

→ There are various strategies for deciding at each point in a program what values should reside in registers (Register Allocation) and in which register each value should reside (Register Assignment)

→ One approch to Register Allocation and Assignment is to assign specific values in the target program to certain registers.

Eg:- Base addresses to one group of registers

Arithemetic compatations to another group

Top of stack to ~~one~~ fixed register

Advantage

→ It simplifies the design of code generator.

Disadvantage

→ It uses registers inefficiently, certain registers may go unused.

Global Register Allocation

→ Registers are assigned to frequently used variables and those registers are kept consistent across block boundaries (globally)

→ Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop.

→ One strategy for global register allocation is to assign some fixed no. of registers to hold the most active values in each inner loop.

→ The selected values may be different in differ loops.

→ Registers not already allocated may be used to hold values local to one block.

→ This approach has a disadvantage that fixed no. of registers is not always the right number to make available for Global Register Allocation.

Usage Counts

→ By keeping variable x in a register for a duration of loop L is one unit of cost for each reference to x if x is already in a register.

→ However, if code generation Algorithm is used there is a good chance that after x has been computed in a block it will remain in a register if there are subsequent uses of x in that block.

→ Thus, we count savings of one for each ~~block in loop 'L' for which x is live on exit~~

→ Also two units are saved if we can avoid a store of x at the end of block.

** If x is allocated to a register, we count a savings of two for each block in loop L for which x is live on exit and in which x is assigned a value.

→ on debit side, if x is live on entry to the loop header, we must load x into its register just before entering loop L. This load has '2' units.

→ For each exit bolock B of loop L at which x is live on entry to some successor of B outside of L, we must store x at a cost of two.

→ As loop is iterated many times, we may neglect these debits since they occur only once we enter the loop.

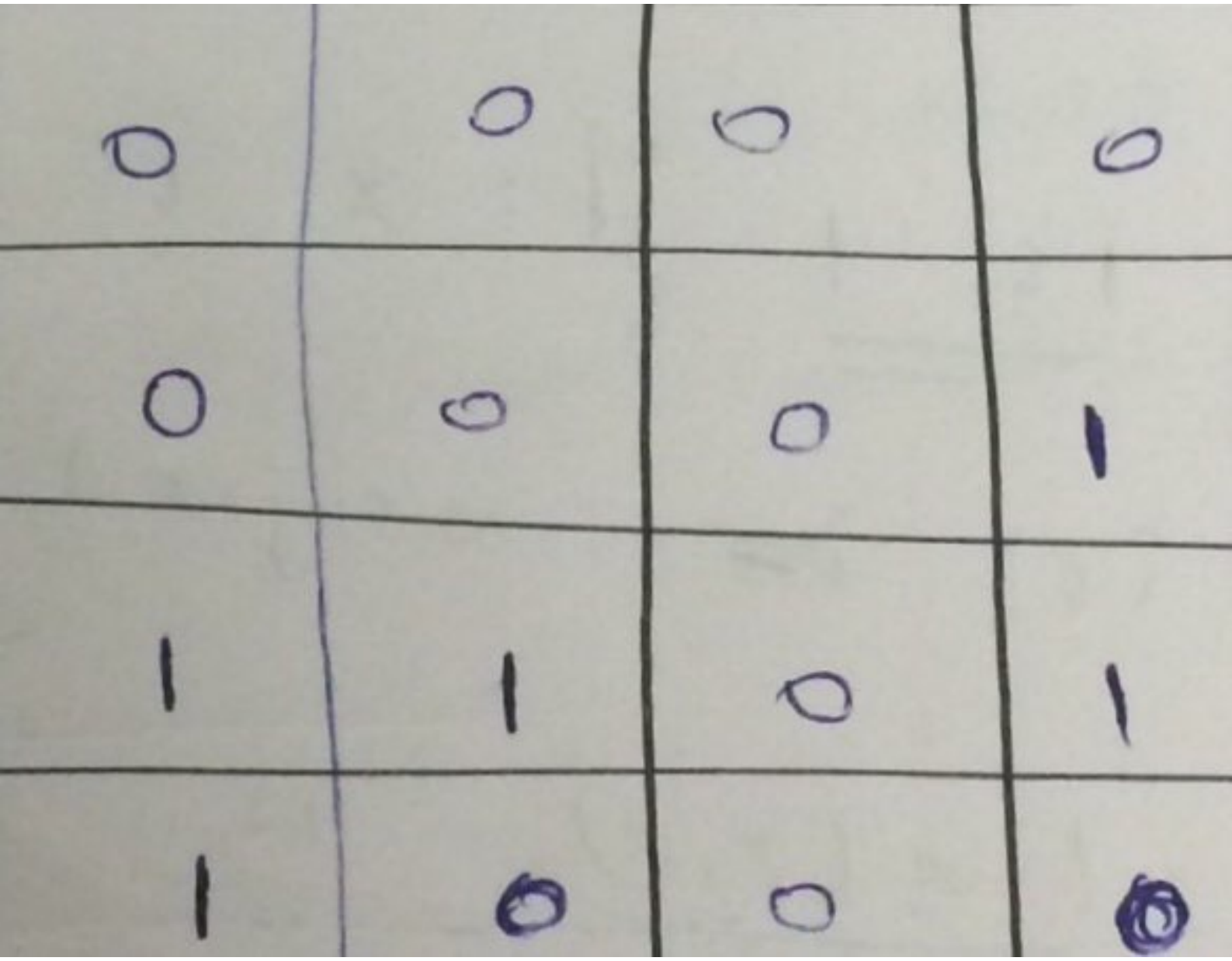
→ Formula for the benifit to be realized from allocating a register x within the loop L is

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B)$$

→ use (x, B) is no of times x is used in B prior to any definition of x.

→ live (x, B) is 1 if x is live on exit from B and is assigned a value in B

→ 0 otherwise live (x, B) is 0.



$b = use(b, B_1) + 2 * live(b, B_3) + 2 * Live(b, B_4)$

$b = 2 + 2 * 1 + 2 * 1$

$\boxed{b = 6}$

$c = use(c, B_1) + use(c, B_3) + use(c, B_4)$

$c = 1 + 1 + 1 = 3$

$\boxed{c = 3}$

$d = use(d, B_1) + use(d, B_2) + use(d, B_3) + use(d, B_4) + 2 * Live(d, B_1)$

$= 1 + 1 + 1 + 1 + 2 * 1$

$\boxed{d = 6}$

$e = 2 * Live(e, B_1) + 2 * Live(e, B_3)$

$= 2 * 1 + 2 * 1 = 4$

$\boxed{e = 4}$

$f = 2 * Live(f, B_2) + use(f, B_1) + use(f, B_3)$

$= 2 * 1 + 1 + 1 =$

$\boxed{f = 4}$

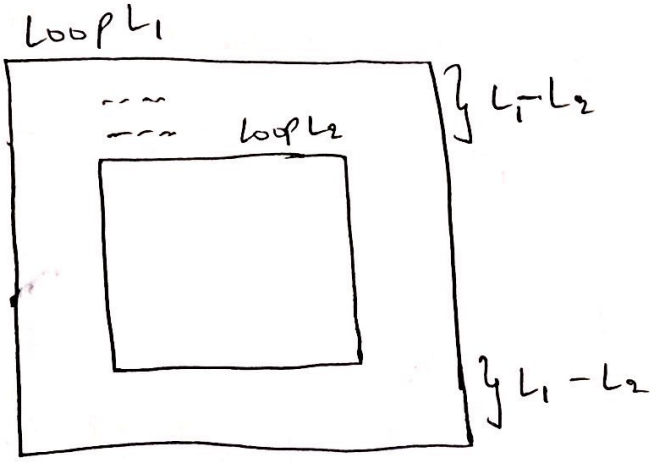
→ we may select to allocate registers for variables with more count. //

Register Assignment for outer loops

→ If an outer loop L_1 contains an inner loop L_2, the names allocated registers in L_2 need not be allocated registers in $L_1 - L_2$.

→ If we choose to allocate x a register in L_2 but not L_1, we must load x on entrance to L_2 and store x on exit from L_2.

→



→ The following criteria should be adopted for register assignment for outer loop

(i) if a is allocated in loop L_2 then it should not be allocated in $L_1 - L_2$

(iii) If a is allocated in L_1 and it is not allocated in L_2 then store a on entrance to L_2 and load a while leaving L_2

(iii) If a is allocated in L_2 and not in L_1 then load a on entrance of L_2 and store a on exit from L_2.

Graph coloring for Register Assignment

(or)

Register Allocation by Graph Coloring

→ When a register is needed for a computation but all available registers are in use, the contents of the used registers must be stored (spilled) into a memory location in order to free up register.

→ Graph coloring is a simple, systematic technique for allocating registers and managing register spills.

→ In this method two passes are used.

→ In the first, target-Machine instructions are selected as though there are an infinite number of symbolic registers; In effect names used in Intermediate code become names of registers and three address instructions become machine-language Instructions.

→ In the second, for each procedure a Register-Interference-graph is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where other is defined.

→ Eg:-

```
a = b + c
d = d - b
e = a + f
```

Block B_1

→ In Block B_1, a is live at the second statement, which defines d, therefore in the graph there would be an edge between the nodes for a and d.

→ An attempt is made to color to register interference-graph using k-colors, where k is number of assignable registers.

→ A graph is said to be colored if each node has been assigned a color in such a way that no two adjacent nodes have same color.

→ A color represents a register, and a color makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register.

Peephole optimization

→ A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter (or) faster sequence, whenever possible.

→ peephole optimization can also be applied directly after intermediate code generation to improve the intermediate code representation.

→ peephole is a small, sliding window on a program.

→ This technique is applied to improve the performance of program by examining a short sequence of instructions in a window (peephole) & replace the instructions by a faster (or) short sequence of instructions.

→ Program transformations that are characteristic of peephole optimizations:

(i) Eliminating redundant loads & stores

(ii) Eliminating Unreachable code

(iii) Flow of control optimizations

(iv) Algebraic simplifications

(v) Use of Machine idioms.

(i) Elimination of redundant loads & store

```
LD   a, R0
ST   R0, a
```

→ we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been loaded into Register R0.

(ii) Eliminating Unreachable code

→ An unlabeled instruction immediately following an unconditional jump may be

removed.

→ This operation can be repeated to eliminate a sequence of instructions.

Eg: if debug == 1 goto L_1

goto L_2

L_1 : print debugging information

L_2 :

→ one obvious peephole optimization is to eliminate jumps over jumps.

→ Thus no matter what the value of debug the code sequence above can be replaced by

if debug != 1 goto L_2

print debug information

L_2 :

→ If debug is set to 0 (zero) at the beginning of the program, constant propagation would transform sequence into

if 0 != 1 goto L_2

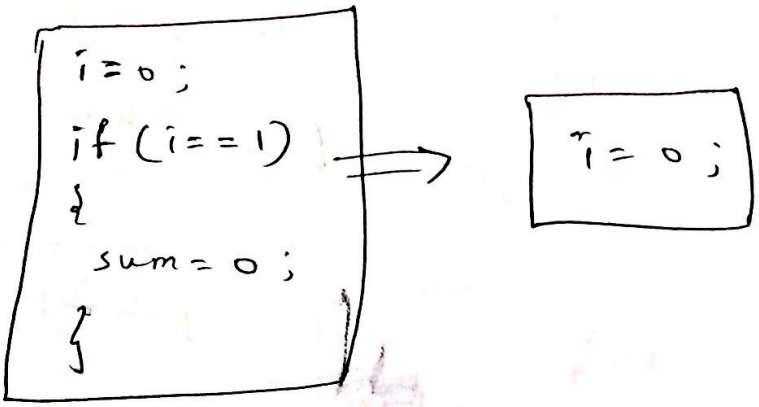
print debug information

L_2 :

→ Now the argument of the first statement always evaluates to true, so that statement can be replaced by goto L_2.

→ Then all statements that print debugging information are unreachable and can be eliminated one at a time.

Eg:



Eg:-

```
int add (int a, int b)
{
  int c=a+b;
  return c;
  printf ("%d", c);
}
```

→ Eliminate

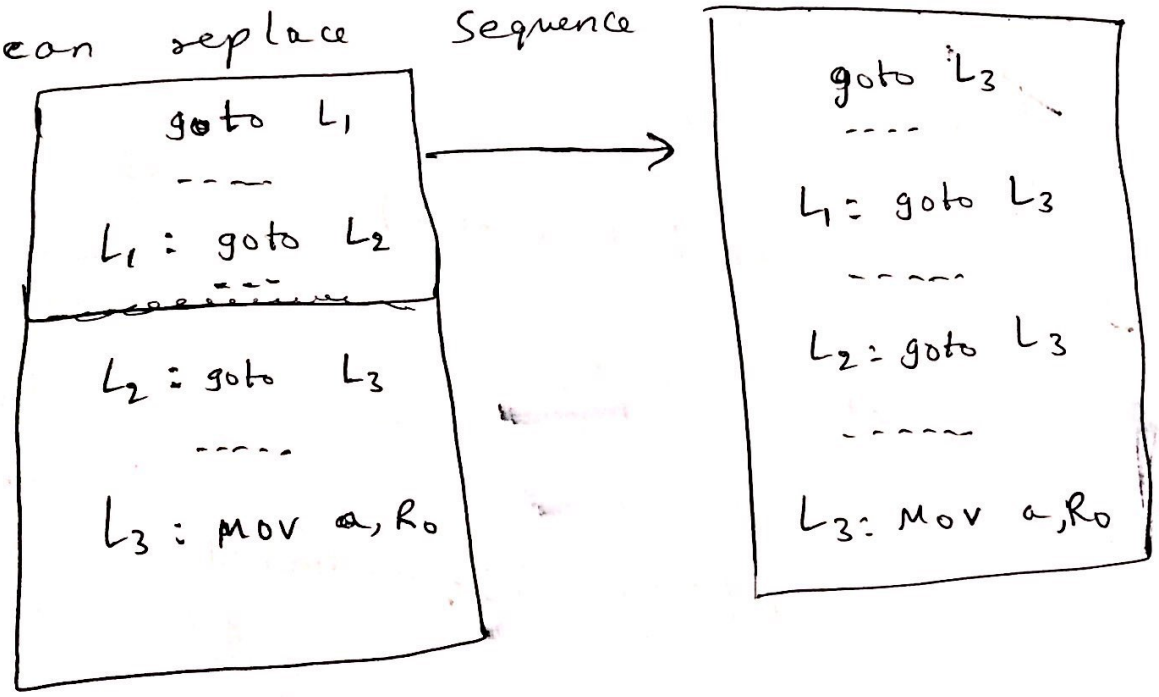
(iii) Flow-of-control optimizations

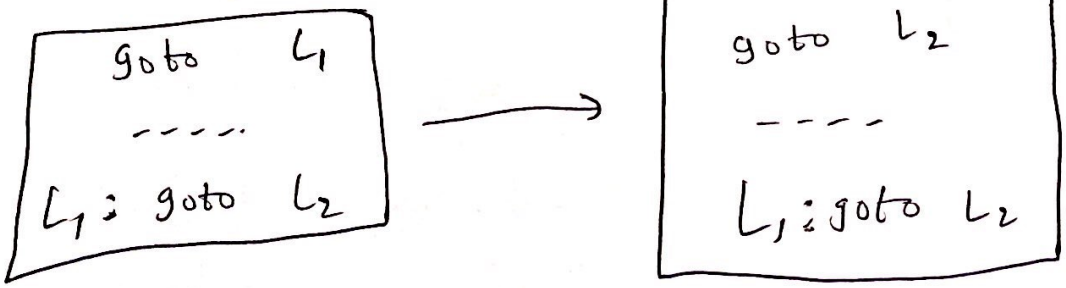
→ Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, (or) conditional jumps to jumps.

→ These unnecessary Jumps can be eliminated in either the intermediate code (or) the target code by following types of peephole optimizations.

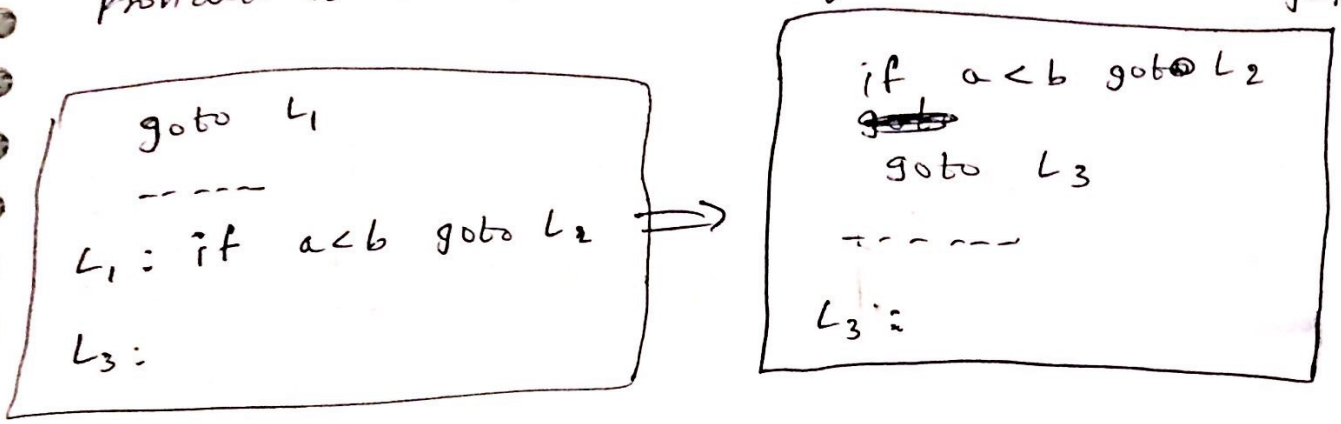
→ we can replace Sequence

Eg:





→ The statement L_1 : goto L_2 can be eliminated provided it is preceded by an unconditional jump.



(iv) Algebraic Simplifications and Reduction in strength

→ The algebraic identities used by a peephole optimizer to eliminate three-address statements such as

$x = x + 0$ (or) $x = x * 1$ in a peephole.

→ The above statements can be eliminated because by executing those statements the result 'x' won't change.

→ Reduction in strength transformations can be applied in a peephole to replace expensive

operations by equivalent cheaper ones on the target machine.

Eg: $a = x^2$ replaced with $a = x * x$

$b = y/8$ replaced with $b = y >> 3$

(V) Use of Machine Idioms

→ The target machine may have hardware instructions to implement certain specific operations efficiently.

→ Detecting situations that permit the use of these instructions can reduce execution time significantly.

→ For example, auto inc/dec features can be used to inc/dec variable

Eg:

$a = a + 1$	→	Inc a
$a = a - 1$	→	DEC a

```
LD   R, a
ADD  R, R, #1
ST   a, R
```

⇒ INC a

```
LD   R, a
SUB  R, R, #1
ST   a, R
```

⇒ DEC a

A Basic Mark and sweep Collector

→ Mark-and-sweep garbage collection algorithms are straight forward, stop-the world algorithms that find all the unreachable objects, and put them on the List free space

→ Algorithm ~~marks~~ "visits" and "marks" all reachable objects in the first tracing step.

→ "Sweeps" the entire heap to free up unreachable objects.

Algorithm : Mark-and-sweep garbage collection.

Input: ~~A~~ A Root set of objects, A heap, and a freelist, called Free, with all unallocated chunks of the heap.

All chuncks of space are marked with boundary tags to indicate their free/used status and size.

ouput: A modified ~~Free~~ Free list after all the garbage has been removed.

METHOD:

→ The algorithm uses several simple data structures.

→ List Free holds objects known to be free.

→ A list called unscanned, holds objects that we have determined are reached, but whose successors we have not yet considered. i.e, we have not scanned these objects to see what other objects can be reached through them.

→ The unscanned list is empty initially.

→ Additionally, each object includes a bit to indicate whether it has been reached (the reached bit).

→ Before the Algorithm begins, all allocated objects have the reached bit '0'.

Marking Phase

1) set the reached bit to 1 and add to list unscanned each object referenced by the root set;
2) while (unscanned ≠ ϕ) {
3) remove some object o from unscanned;
4) for (each object o' referenced in o) {
5) if (o' is unreached; i.e it reached bit is 0) {
6) set the reached bit of o' to 1;
7) put o' in unscanned;
8) }
9) }
10) }

Sweeping Phase

11) Free = ϕ

12) for (each chunck of memory O in heap) {

13) if (O is unreached, i.e its reached bit is 0)

add O to Free;

14) else set reached bit of O to zero

15) }

→ In Line (1) of Marking phase, we initialize unscanned list by placing these all the objects referenced by the root set.

→ The reached bit for all these object is also set to 1.

→ Lines (2) to (7) are in loop, in which we, in turn, examine each object O this is ever placed in Unscanned List.

→ The for-loop of Lines (4) to (7) implements scanning of objects O.

→ we examine each object O' for which we find a reference within O.

→ If O' has already been reached (if reached bit is 1), then there is no need to do anything about O'; it has either been scanned previously, or its is on unscanned list to be scanned later.

→ If O' was not reached already, then we need to set its reached bit to 1 in Line (6) and add O' to the unscanned list in Line (7).

→ Lines (11) to (14), the sweeping phase, reclaim the space of all the objects that remain unreached at the end of the marking phase.

→ Note that these will include any objects that were on the FreeList originally.

→ Because the set of unreached objects cannot be enumerated directly, the algorithm sweeps through entire heap.

→ Line (13) puts free and unreached objects on the Free List, one at a time.

→ Line (14) handles the reachble objects we set their reached bit to zero (0), mord to maintain the proper predictions for the next execution of the garbag collection Algorithm.