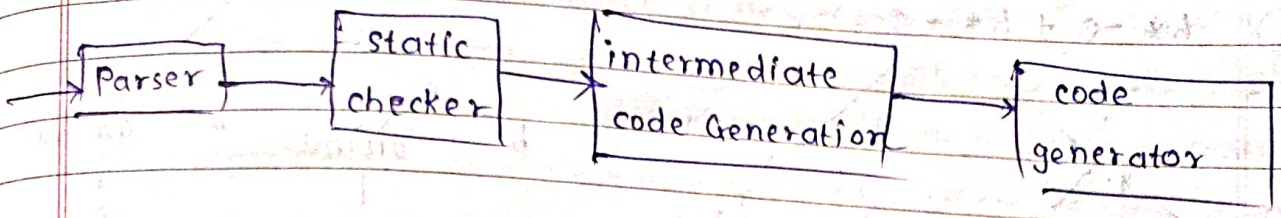
Intermediate Code Generation



Benifits :-

(1) Retargating is facilitated.

A compiler for a different machine can be created by attaching a backend for the new machine to an existing front-end

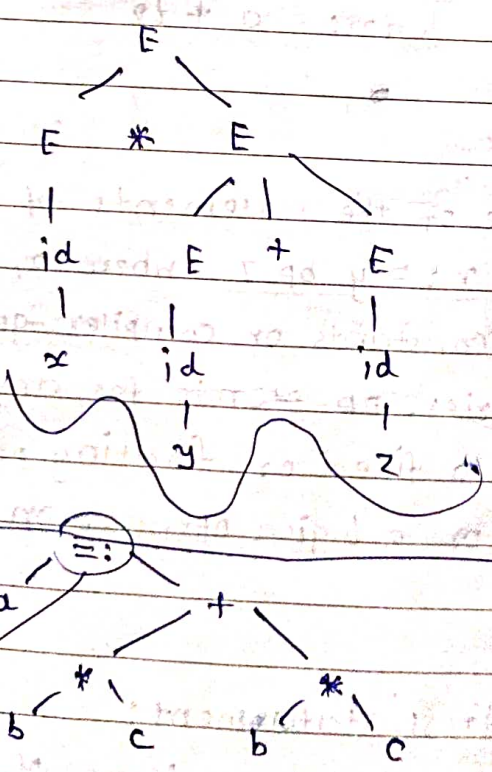
(2) A Machine Indipendent code optimizer can be applied to the intermediate representation.

$E \rightarrow E+E$

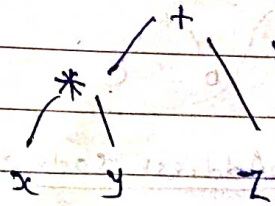
$E \rightarrow E*E$

$E \rightarrow id$

x * y + z



Syntex Tree :- DAG not possible Because there is no repetation.



$t_1 := x * y$

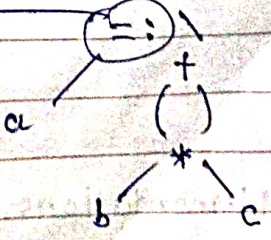
$t_2 := t_1 + z$

a := b * c + b * c

DAG is possible

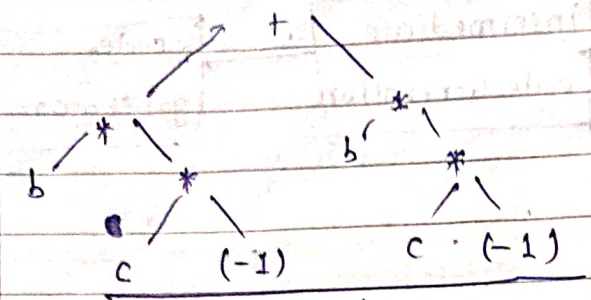
asign

Syntex Tree

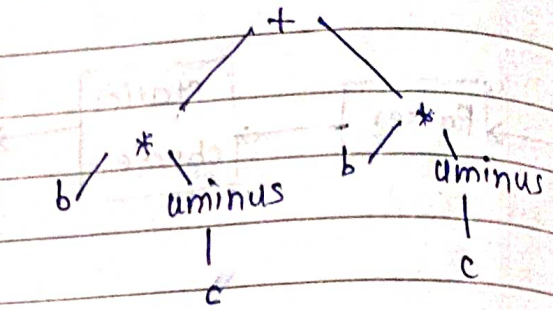


DAG

(*) b * -c + b * - c



X (wrong)



Syntax tree

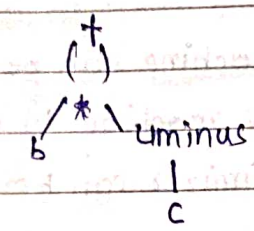
for synatax tree,

(iii) $t_1 = -c$ $t_3 = -c$

$t_2 = b * t_1$ $t_4 := b * t_3$

$t_5 := t_2 + t_4$ $a := t_5$

DAG :-



for DAG :- $t_1 = -c$, $t_2 = b * t_1$ $t_3 = t_1 + t_2$ $a := t_3$

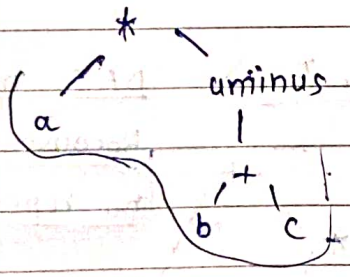
(*) Translate the arithmatic expression a * -(b + c) into

(i) A syntax tree

(ii) Postfix notation

(iii) Three Address code

(i)



(ii) a b c + uminus *

(iii) $t_1 := b + c$

$t_2 := -t_1$

$t_3 := a * t_2$

Three Address code :- Sequence of the statements of the general form $x := y$ op z where x, y & z are names, constants or compiler generated temporaries. op stands for any operator such as a fixed or floating point arithmetic operator or a logical operator on boolean valued data.

Implementations of three address statement :-

→ A three address statement is an abstract form of a intermediate code

→ In a compiler this statements can be implemented as a records

with fields for the operator & the operands.

→ Three such a representations are quadraple, triples & indirect triples.

(i) quadraples:- A quadraples is a record structure with four fields which we call op, arg1, arg2, result.

→ The op field contains an internal node for the operator

→ The Three address statement x := y op z is represent by placing y in arg1, z in arg2, & x in result.

x : y op z
↓ ↓ ↓
result arg1 arg2

→ statement in unary operators like x := -y & x := y

→ Do not use arg2.

→ The contents of fields arg1, arg2 & result are normally pointers to the symbol table entries for the names, represented by this fields so, temperary name must be entered into the symbol table as they are created.

Ex. a = b * -c + b * -c

t1 := -c
t2 = b * t1
t3 := -c
t4 = b * t3
t5 = t2 + t4
a := t5

Add.	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

(ii) triples:- To avoid entering temporary names into the symbol table we might refer to a temporary value by the position of the statement that computes it, so three address statement can be representated by records with only three fields: op, arg1, arg2

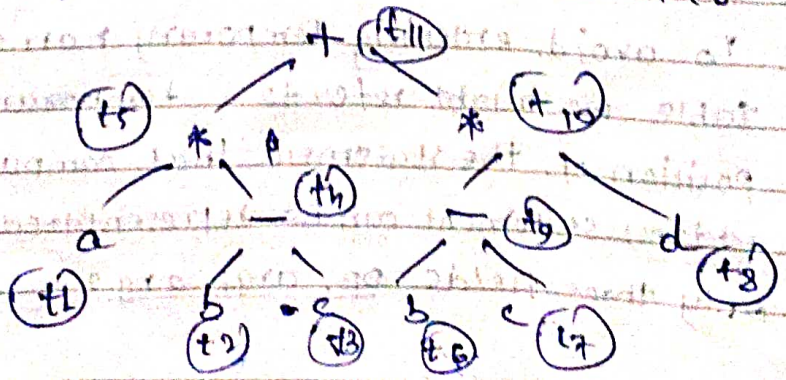
→ The fields arg1 & arg2 for argument op are either pointers to the symbol table or pointers into the tripple structure, since three fields are used this intermediate formate is known as tripples.

Add	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

(iii) Indirect triples:- another implementation of three address co is consider that listing pointer to triples rath listing the triples themsself. this implementa is called as "indirect triples"

	Statement	Add	op	arg1	arg 2
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	:=	a	(18)

(*) Translate the expressions unary minus into synotax tree, postfix notation & three address code. Ex:- a * (b - c) + (b - c) * d



$t_1 := leaf(id, entry-a)$
$t_2 := leaf(id, entry-b)$
$t_3 := leaf(id, entry-c)$
$t_4 = node('-', t_2, t_3)$
$t_5 = node('*', t_1, t_4)$
$t_6, t_7, t_8 \rightarrow leaf, b, c, d$
$t_9 := node('-', t_6, t_7)$
$t_{10} := node('*', t_9, t_8)$
$t_{101} := node('=', t_5, t_{10})$

$t_1 := b - c$
$t_2 = a * t_1$
$t_3 = b - c$
$t_4 = t_3 * d$
$t_5 := t_2 + t_4$

Postfix notation :- a b c - * b c - d * +

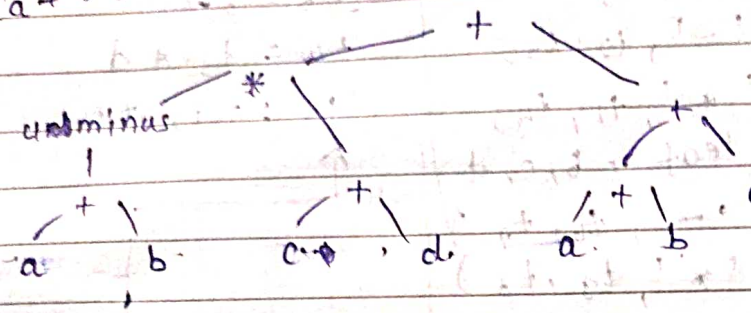
Three Address code :-

(*) Translate the expression into quadruples, triples and Indirect triples

(*) Translate the expression into a syntax tree, postfix notations, three address code, quadraplet triples and indirect triples.

$-(a+b) * (c+d) + (a+b+c)$

→



$t_1 = a+b$
$t_2 = (\text{uminus}) - t_1$
$t_3 = c+d$
$t_4 = t_2 * t_3$
$t_5 = a+b$
$t_6 = t_5 + c$
$t_7 = t_4 + t_6$

Postfix notation :- $ab+ - cd+ * ab+c++$
↑ uminus

(i) Quadraplets :-

Add	op	arg1	arg2	result
(0)	+	a	b	t1
(1)	uminus	t1		t2
(2)	+	c	d	t3
(3)	*	t2	t3	t4
(4)	+	a	b	t5
(5)	+	t5	c	t6
(6)	+	t4	t6	t7

(ii) triples :-

Add	op	arg1	arg2
(0)	+	a	b
(1)	uminus	(0)	
(2)	+	c	d
(3)	*	(1)	(2)
(4)	+	a	b
(5)	+	(4)	c
(6)	+	(3)	(5)

(iii) indirect triples :-

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)
(6)	(20)

Add	op	arg1	arg2
(14)	+	a	b
(15)	uminus	(14)	
(16)	+	c	d
(17)	*	(15)	(16)
(18)	+	a	b
(19)	+	(18)	c
(20)	+	(17)	(19)

(*) x[i] = y ⇒ x = y[i] → ternary operation x = y[i]

i) quadruple :-

Add	op	arg1	arg2	result
(0)	[] =	x	i	t1
(1)	assign	y		t1

Add	op	arg1	arg2	result
(0)	=[]	y	i	t1
(1)	assign	t1		x

ii) triples :-

Add	op	arg1	arg2
(0)	[] =	x	i
(1)	assign	(0)	y

Add	op	arg1	arg2
(0)	=[]	y	i
(1)	assign	x	(0)

(*) z = x[i] + y[i]

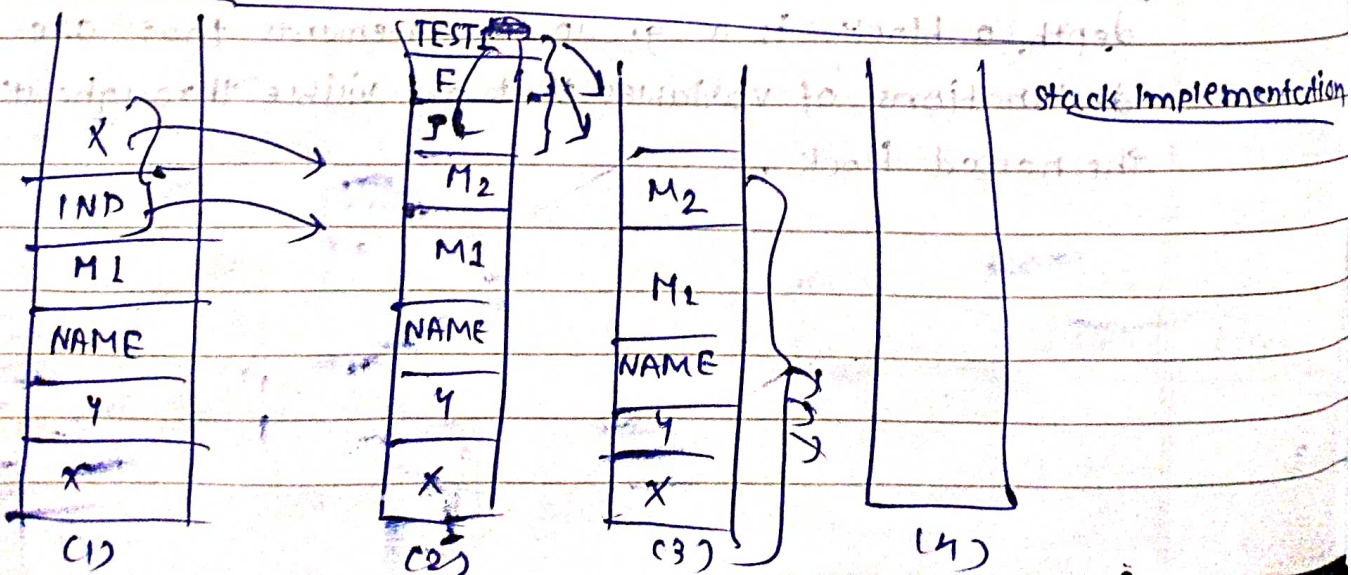
(*) Symbol Table Organization for blocked Structure languages

→ Block Structure Language comprise a class of high level language in which a program is made up of blocks which may include nested blocks as components. Such nesting being repeted to any depth, a block is a group of statements that are preceded by declarations of variables that are visible throughout the block & the nested block.

```
BBLOCK
    Real X; Y, STRING NAME ;

    M1 : PBLOCK ( INTEGER IND);
         INTEGER X;
         CALL M2 ( IND+1 );
    END M1

    M2 : PBLOCK (INTEGER J)
         BBLOCK;
         ARRAY INTEGER F(J);
         F(I);
         LOGICAL TEST1;
         END;
    END M2

         CALL M1 (X/Y);

END ;
```

b1, b2, b3, b4

A Trace showing the affect of Set & Reset operations.



Stack Implementation

	Symbol table contents	
operation	Active	Inactive
Set Block1	Empty	Empty
set Block2	M1, NAME, Y, X	Empty
Reset Block2	X, IND, M1, NAME, Y, X	Empty
set Block3	M2, M1, NAME, Y, X,	X, IND
Reset Block4	I, M2, M1, NAME, Y, X	X, IND
Reset Block4	TEST1, F, I, M2, M1, NAME, Y, X	X, IND
Reset Block3	I, M2, M1, NAME, Y, X	TEST1, F, X, IND
Reset Block1	M2, M1, NAME, Y, X ~~Empty~~	TEST1, F, X, IND
END OF COMPILATION	Empty	M2, M1, NAME, Y, X, I, TEST1, F, X, IND

```
BBLOCK;
    REAL X;
    INTEGER Y;

    SUB1 : PBLOCK (INTEGER I);
        ;
        BBLOCK;
            ARRAY STRING S(1+2);
            LOGICAL FLAG;
            INTEGER I;
        END;
```

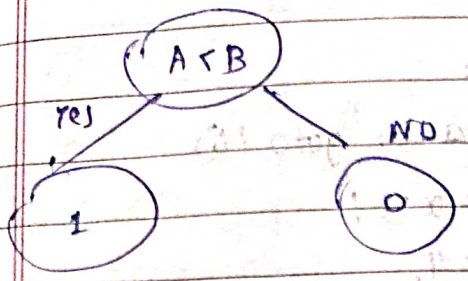
b1 b2 b3

```
SUB2 : PBLOCK ( REAN W, Z);
        REAL I, J;
        LOGICAL TEST 1, TEST 2, TEST 3;
    END ;
END ;
```

Symbol table contents

operation	Active	Inactive
Set Block1	Empty	empty
Set Block2	SUB1 , Y, Z	Empty
Set Block3	I, SUB1, Y, Z	Empty
Reset Block3	BY, FLAG, S, I, SUB 1, Y, Z	Empty
set Block4	SUB2, I, SUB1, Y, Z	Y, FLAG, S
Reset Block4		

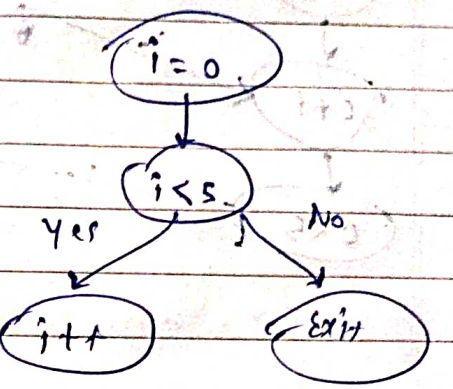
(*) If A < B then 1 else 0
(i) Syntax Tree (ii) postfix notation (iii) three address code.



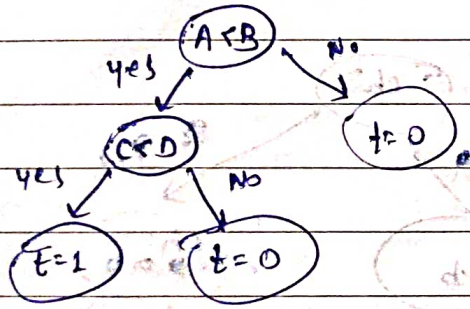
Postfix :- 1 0 A < B

Three address code:-
(1) if A < B goto (4)
(2) $T_1 = 0$
(3) goto (5)
(4) $T_2 = 1$
(5)

for (i = 0 ; i < 5 ; i++)



(*) if A<B and C<D then t=1 else t=0

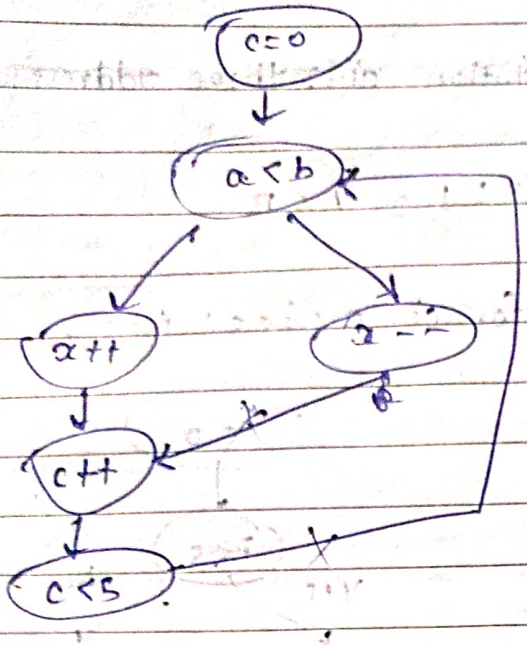


1 0 C<D 0 A<B

(1) If (A<B) goto (3)
(2) goto (4)
(3) If (C<D) goto (6)
(4) t=0
(5) goto (7)
(6) t=1
(7)

(i) If (A<B) goto (3)
(ii) t=0 goto (6)
(iii) if (C<D) goto (5)
(iv) t=0 goto (6)
(v) t=1
(vi)

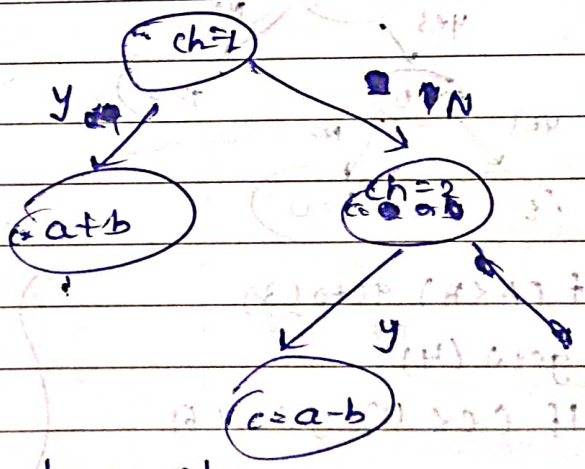
(*) c=0;
do
{ if (a<b) then
x++;
else
x--;
c++;
} while (c<5).



postfix notation:
c<5 c++ x++ x-- a<b c=

(1) c=0
(2) if (a<b) goto (6)
(3) $T_1 = x-1$;
(4) $x = T_1$
(5) goto (8)
(6) $T_2 = x+1$;
(7) $x = T_2$
(8) $T_3 = c+1$
(9) $c = T_3$
(10) if (c<5)
(11) goto (2)
(12)

* switch (ch)
{
case 1: c = a+b;
break;
case 2: c = a-b;
brack;
}

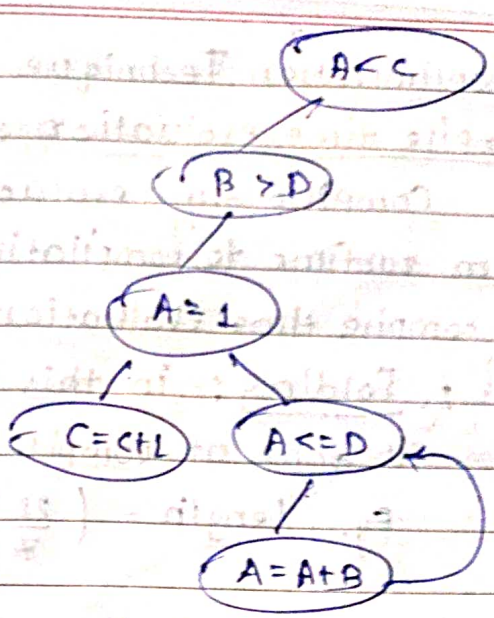


postfix notation:
c= a+b c=a-b ch=2 ch=1

(1) if ch=1 goto T_1
(2) if ch=2 goto T_4
(3) T_1 : $T_2 = a+b$
(4) $c = T_2$
(5) goto T_3
(6) T_4 : $T_5 = a-b$
(7) $c = T_5$
goto T_3

T_3 :

(*) while (A<C and B>D)
do
if A=1 then
C = C+1
else
while A<=D
do
A = A+B



Postfix notation

C=C+1 A=A+B A<=D A=1 B>D A<C

Code Optimization

Code optimization Techniques :-

(1) Compile time evaluation :-

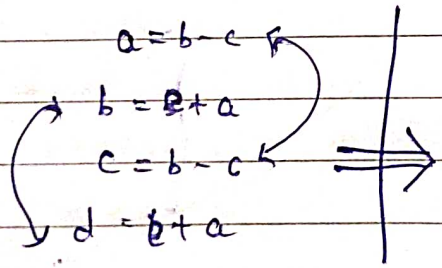
Compile time evaluation means shifting of computation from runtime to compilation. There are two methods to obtain the compile time evaluation.

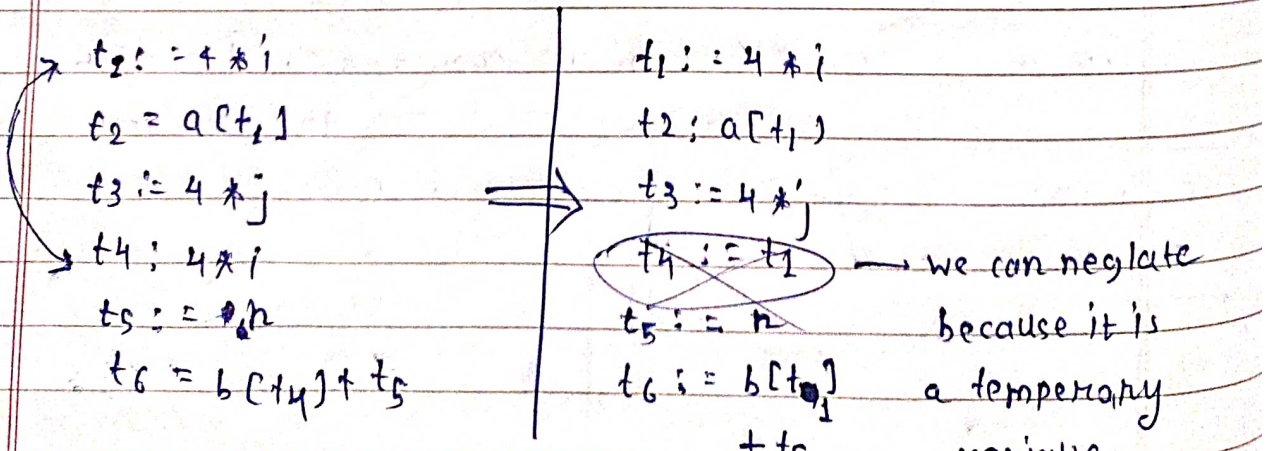
1. Folding :- In this technique the computation of constant is done at compile time instead of runtime.

   Eg. Length = $\left(\frac{22}{7}\right) * d$

2. Constant Propogation :- In this technique the value of variable is replaced and computation of an expression is done at compile time.

(2) Common sub-Expression Elimination :- Is an expression appearing repeatedly in the program, which is computed previously then if the operands of these sub expression do not get changed at all, then result of such sub expression is used instead of recomputing each time.





(3) Variable Propogation :- means use of one variable instead of another.

$x = p_i$

area = x * r * r $\Longrightarrow$ area = p_i * r * r

→ In previous example, t_5 will be placed in t_0

so, $t_6 := b[t_1, I] + n$

(4) Code Movement :- There are basic goals of code movement:-

1. To Reduce the size of code.
2. To Reduce the frequency of execution of a code.

Ex.

1.
```
for (i=0; i<=10; i++)
{
    x = y * 5;
    k = (y * 5) + 50;
}
```
$\Longrightarrow$
```
k = x + 50;
```

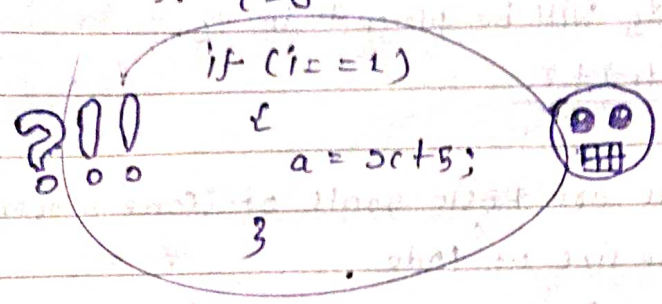
2.
```
x = max-1
```
Ex.
```
while (i<=(max-1)) {
    sum = sum + a[i];
}
```
Loop invarient optimization can be obtained by moving some amount of code outside the loop & placing it just before entering in the loop. therse method is called as code motion.

(5) Strength Reduction :- Strength of certain operators is higher than others. Eg. * > + In this technique the higher strength operators can be replaced by lower strength operators.

Eg.
```
for (i=1; i<=50; i++)
{
    count = i * 7
}
```
$\Longrightarrow$
```
count = 0;
count = count + 7
```

(6) Dead-Code Elimination :- The variable is said to be dead at a point in a program, if the value contained into it is never been used.

Eg. i=0



Loop-Optimization Techniques :-

(1) Code Motion :- } → previously covered

(2) Strength Reduction :- }

(3) Loop Unrolling :- In this method no. of jumps and test can be reduced by writing the code two times,

```
int i=1
while (i<=100)
{
  A[i] = b[i];
  i++;
}
```

```
int i=1
While (i<=100)
{
  A[i] = b[i]
  i++;
  A[i] = b[i]
  i++;
}
```

(4) Loop fusion :- In loop fusion method several loops are merged into one loop.

Eg. for (i=1 to n do
  for j=1 to m do
    A[i,j] = 10

⇒

for i=1, j=1 to n*m do
  A[i,j] = 10

X

(*)

```
for (i=0; i<300; i++)
{
  a = a * i;
}
```

```
for (i=0; i<300; i++)
{
  b = b * i;
}
```

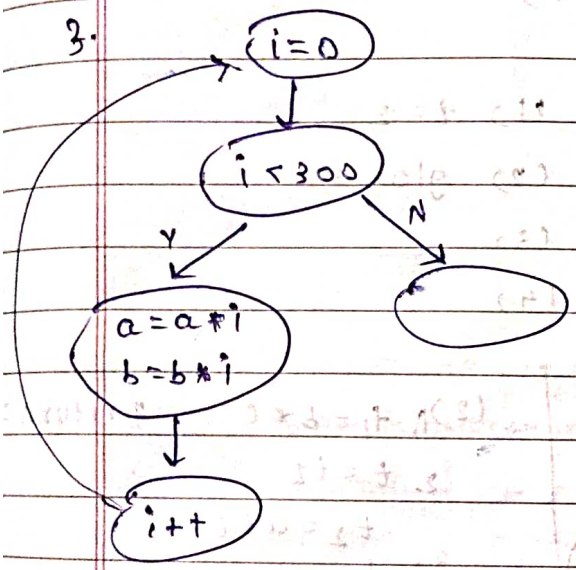
```
for (i=0 ; i<300 ; i++ )
{
    a = a*i;
    b = b*i;
}
```

(*) 1. Generate the target machine code
2. Generate the three address statements
3. Construct a flow graph from the three address code.

for perious example:-

1.

3.



2.

(1) i = 0
(2) if (i<300) goto (4)
(3) goto (7)
(4) a = a * i
(5) b = b * i
(6) i++ goto (2)
(7)

2. (1) i = 0
(2) if (i<300) goto (4)
(3) goto (16)
(4) $t_1 = a * i$
(5) $a = t_1$
(6) $t_2 = b * i$
(7) $b = t_2$
(8) $t_3 = i + 1$
(9) $i = t_3$
(10) goto (2)
(11)

(1) optimize the code
(2) After optimization generate 3 address code and flow graph of this

```
int glob;
void f()
{
    int a;
    a = 1;
    glob = 1;
    glob = 2;
    return;
    glob = 9;
}
```

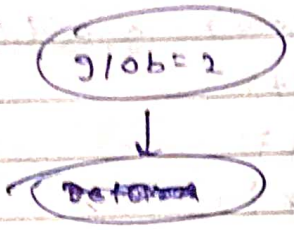
dead-code

(1)
```
int glob;
void f()
{
    int a = 1;
    glob = 2;
    return;
}
```

dead-code
variable propogation

(2) (1) t = 2
(2) glob = t
(3) —
(4)

(3)

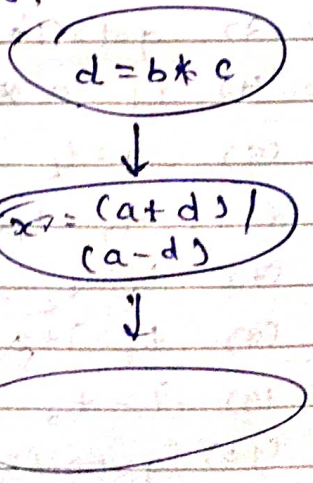



```
int a, b, c, x;
int main()
{
    x = (a + (b*c)) / (a - (b*c));
    return x;
}
```

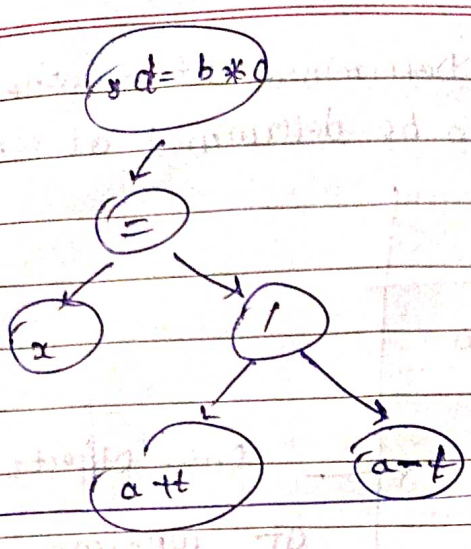
(2) 1. $t_1 = b * c$
2. $t = t_1$
   $t_2 = a + t$
3. $t_3= a - t$
4. $t_4= t_2 / t_3$
5. $x = t_4$
6. 7. return x

(1)
```
int a, b, c, x;
int main()
{
    int d = b*c;
    x = (a+d) / (a-d);
    return x;
}
```

(3)



or

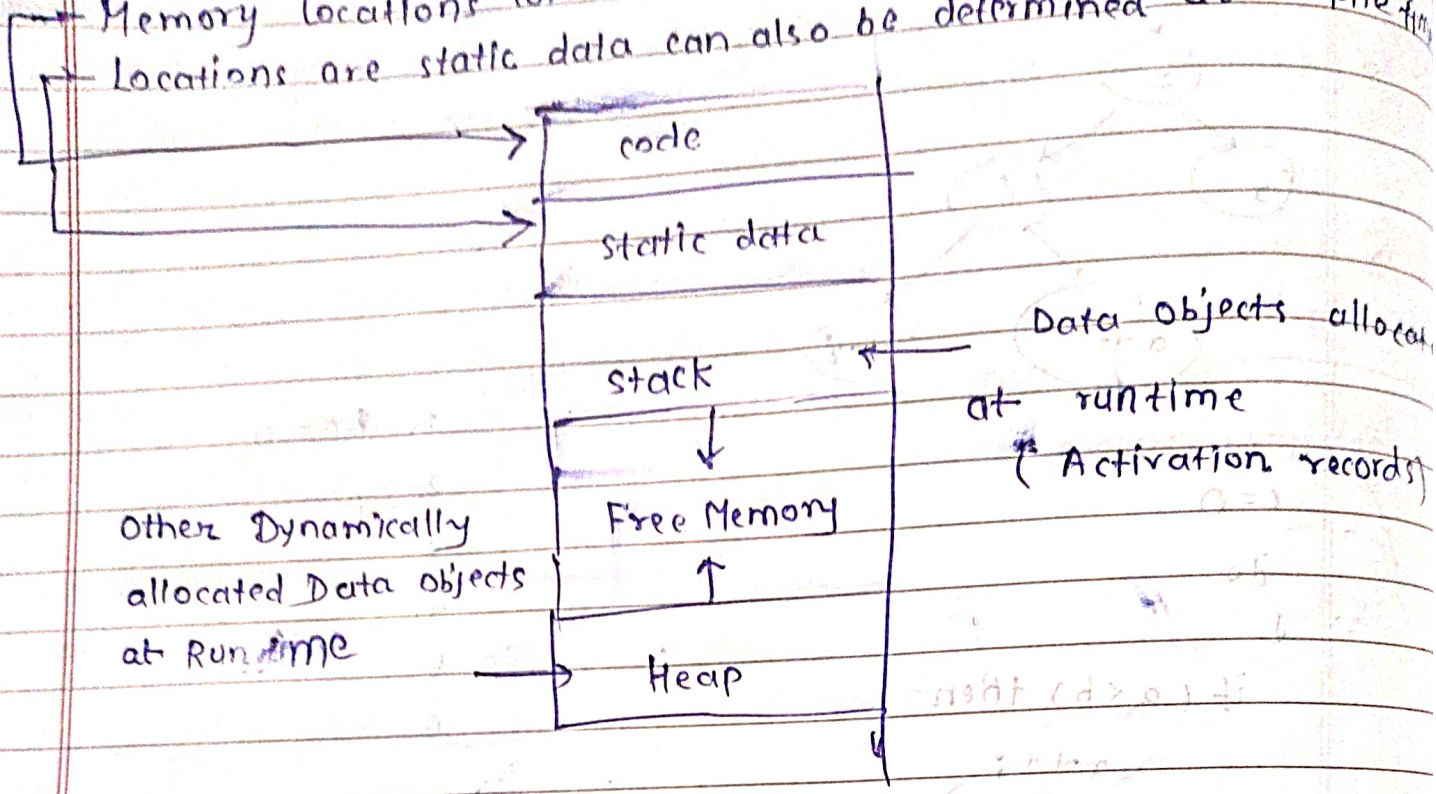



```
c=0
do
{
  if (a<b) then
        x++;
  else
      x--;
      c++;
} while (c<5)
```

1. ! c=0
2. !
3.
4.
5.
6.

Runtime Enviornment

Memory locations for code are Determined at compile time
Locations are static data can also be determined at compile tim



Storage Organization :-

Sub division of run-time memory the compiler obtains a block of storage from the OS for the compile program to run. This runtime storage might be sub divided;

The generated target code, data Objects a counter part of the control stack to keep track of procedure activations.

(*) Typical Subdivision of Runtime Memory into code & data areas

```
1  Program sort (input, output);
2        var a : array [0... 10] of integer;
3  Procedure readarray ;
4        var i: integer ;
5        begin
6            for i: 1 to 9 do read (a[i])
7        end;
8  Function Partition (y, z: integer) : integer
9        var i, j, x, v: integer
10       begin ....
11       end;
12 Procedure quicksort ( m, n:integer ) ;
13       var i :=integer;
14       begin
15           if ( n > m ) then begin
16               i:= ~~func~~ Partition (m, n) ;
17               quicksort ( m, i-1 ) ;
18               quicksort (i+1, n);
19           ~~end;~~ end;
20       end;
21 begin
22     a[0] = -9999; a[10] = 9999;
23     readarray ;
24     quicksort [ 1, 9 ) ;
25 end;
```

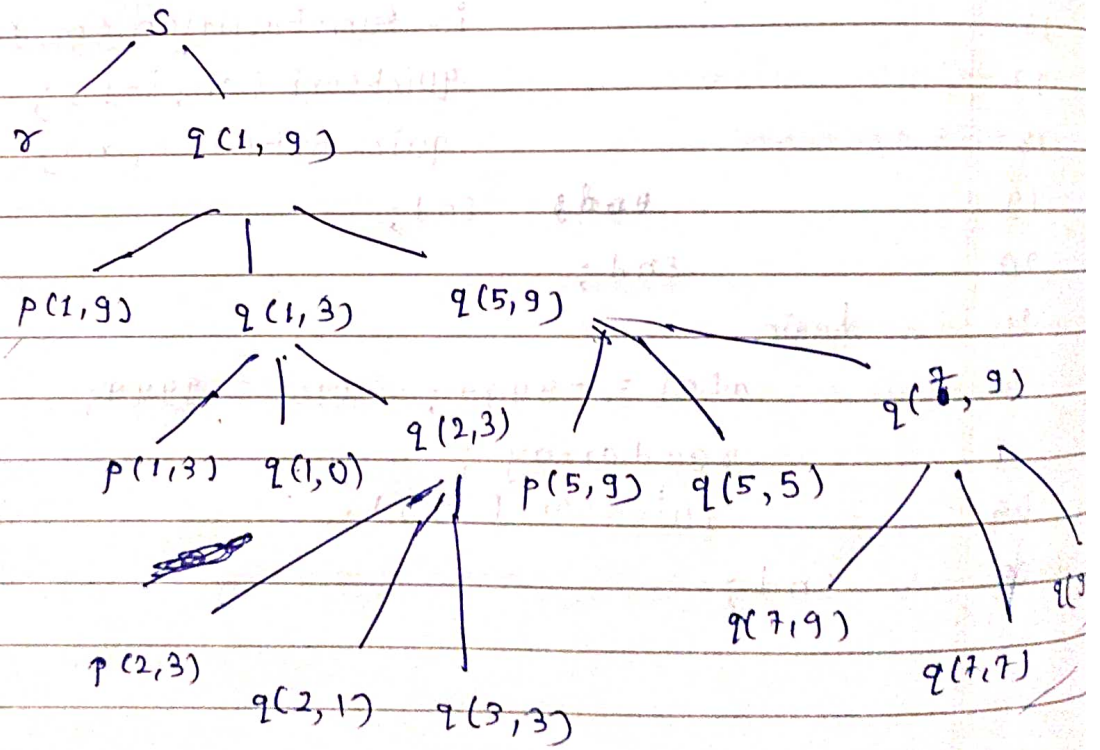
How to find Activation trees

Step 1 Activation of procedure

- 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

Pivot (4)

step 2 Activation Tree



Control links :-

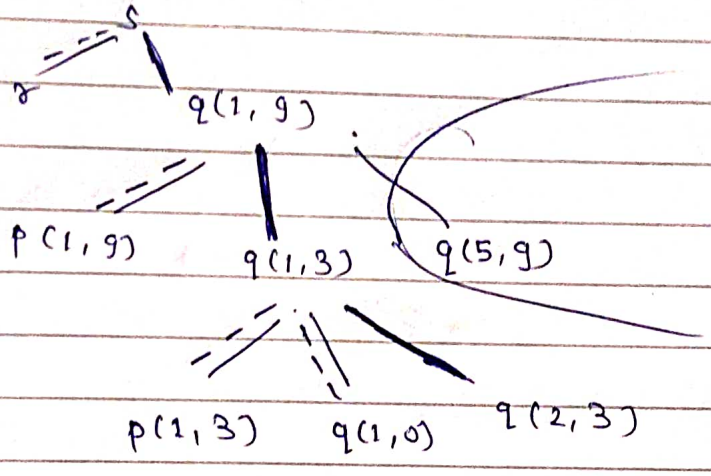
→ You can use a stack which is called as control stack to keep a track of life procedure activation.

→ The idea is to push the node for an activation on to the control stack as the activation begins and to pop the node when the activation ends.

→ Then the contents of the control stack are related to parse to the root of the activation tree.

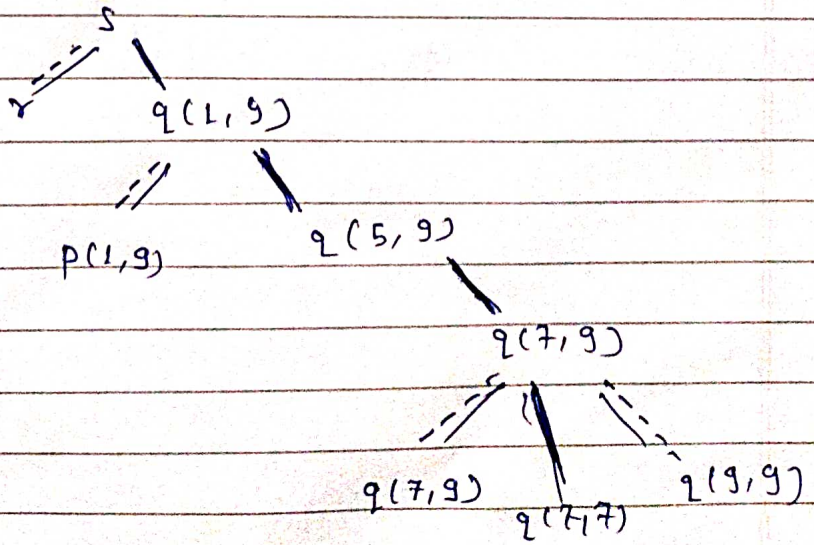
→ Then node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.

* Activation with levels are p(1, 9), p(1, 3), q(1, 0), q(2, 3)



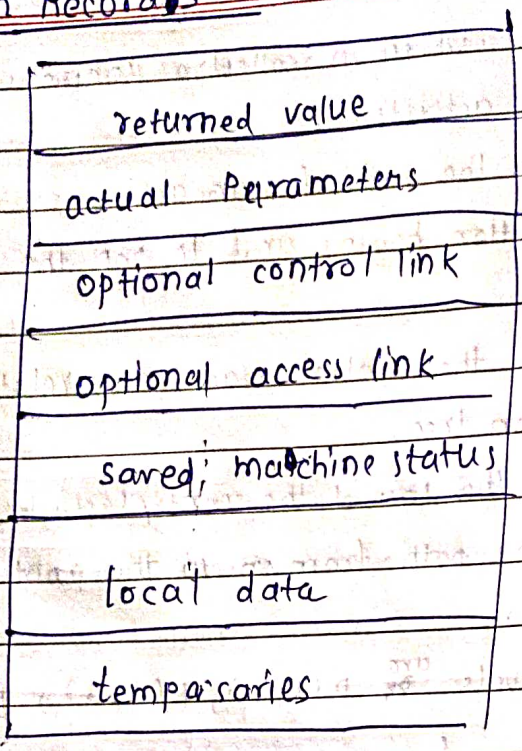
p(1, 9),

partitioning of p(5, 9), q(7, 9) When control enters activation represent by q(7, 7)



Date
Page

Activation Records



→ Information needed by a single execution of a procedure is managed using a continuous block of storage called as activation record or frame.

→ (1) Temporaries is those arising in the evaluation of expression are stored in field for temporaries.

(2) local data which holds data that is local to an execution of a procedure

(3) Saved machine status which holds information about the state of the machine just before the procedure is called. ex. prog. counter.

(4) optional access link :- which refer to non-local data holding other

(5) option cont which points activation records of the

(6) actual parameters which is used by the calling procedure to supply parameters to the called procedure.

(7) Returned values which is used by the called procedure to return the value to the calling procedure

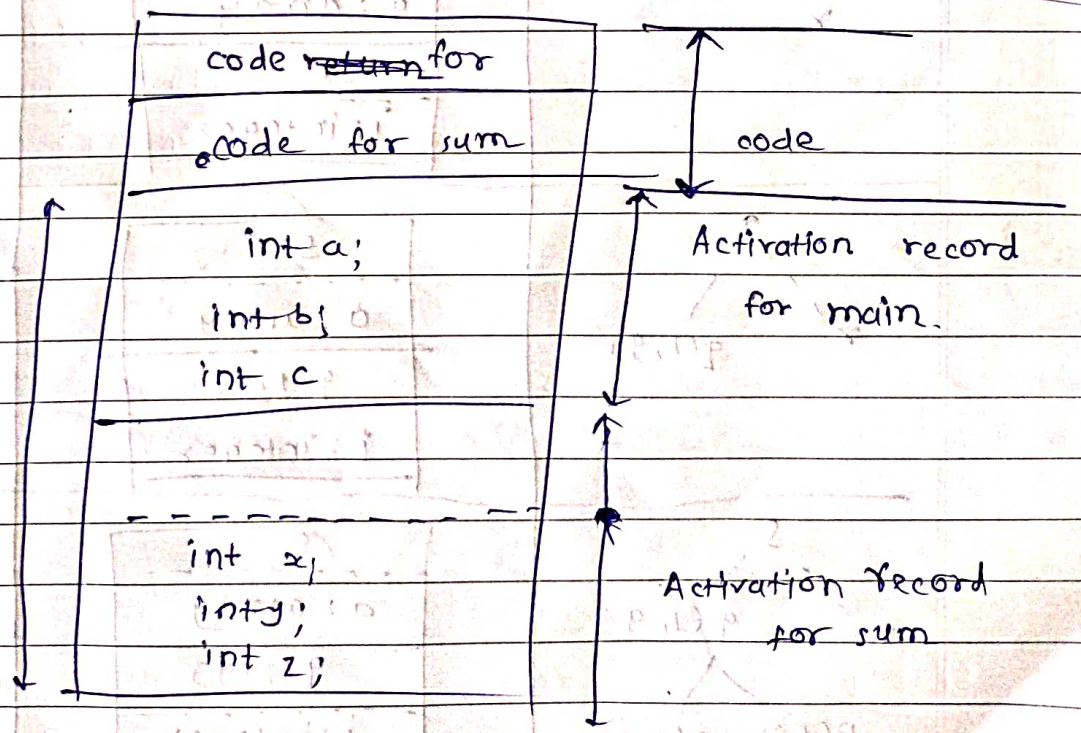
A different storage strategy is used in each of the three data areas in the organization.

(1) Static Allocation
(2) Stack Allocation
(3) Heap Allocation

(1) Static Allocation :-
names are bound to storage during compile time position of activation record, address of names are allocate at compile time.

Ex.

```
void main(){
    int a=100;
    int b=120;
    int c;
    c=sum(a,b)
    printf("%d",c);
}

int sum (int x, int y)
{
    int z;
    z = x+y;
    return z;
}
```



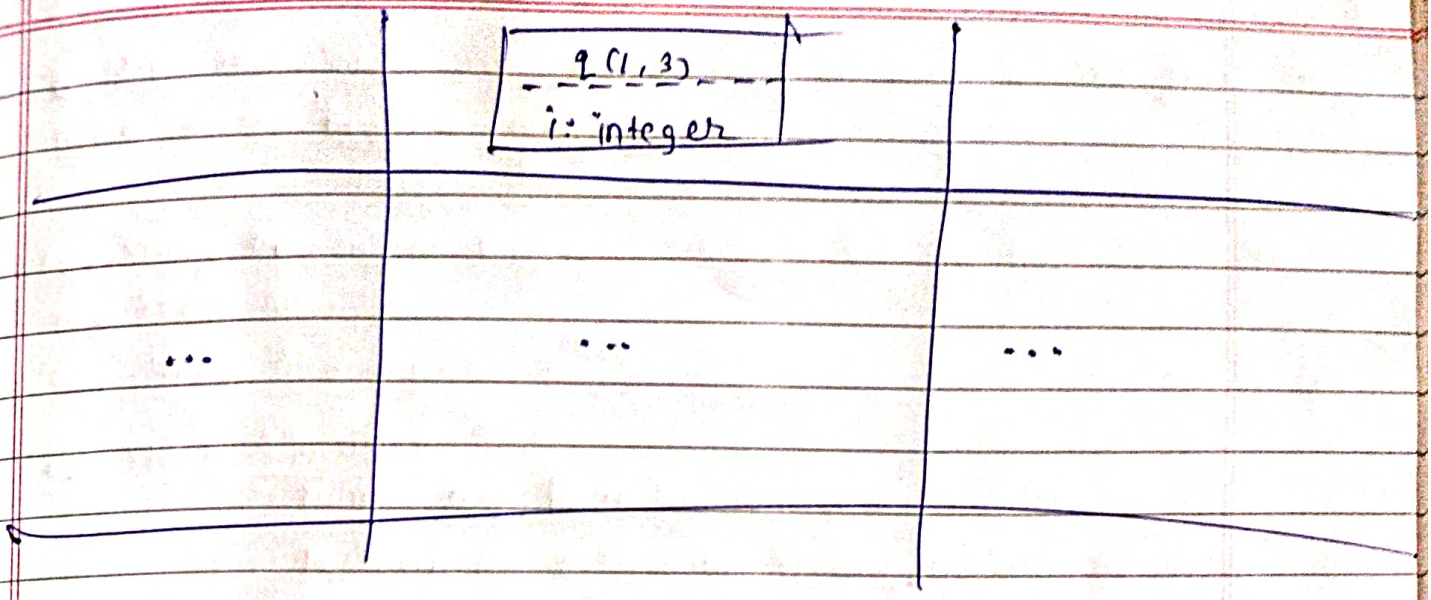
limitations of static allocation

1. size of data object and constraints on its position in memory must be known at compile time.
2. Data structure can not created dynamically, since there is no mechanism for storage allocation at run time.
3. Recursive procedures are restricted because all activations of a procedure used in same binding for local names

(2) Stack Allocation :- is based on the idea of a control stack storage is organized as a stack and activation records are pushed and poped as activation begin and end.

① Downward Growing Stack-allocation

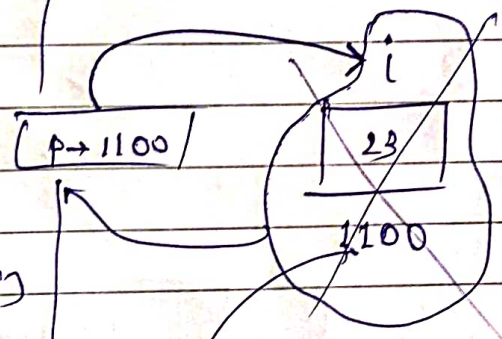
Position in Activation tree	Activation Records on a stack	Remarks
s	s a : array	Frame for s
s, r	s a : array r i : integer	r is activated
s, r, q(1,9)	s a : array q(1,9) i : integer	Frame for r has been popped and q(1,9) pushed
s, r, q(1,9), p(1,3), q(1,3)	s a : array q(1,9) i : integer	Control has just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



Dangling Reference :-

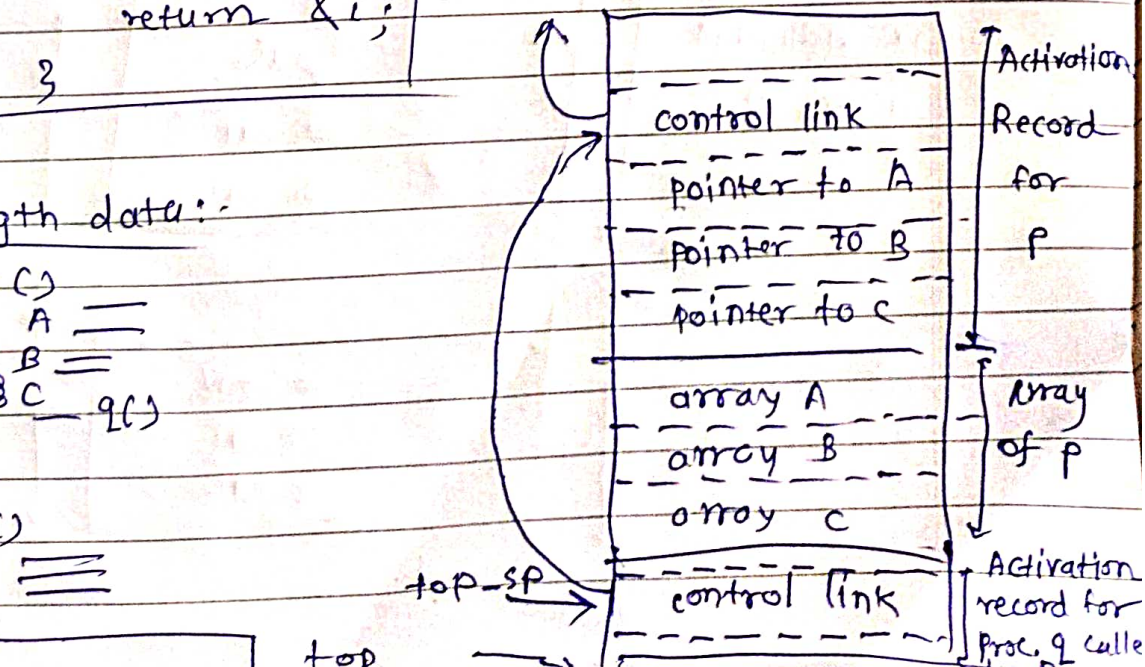
→ Whenever storage can be deallocated the problem of dangline reference arises. The dangling reference occures when there is a reference to storage that has been deallocated.

```
main()
{
  int *p;
  p = dangle();
}
int *dangle()
{
  int i = 23;
  return &i;
}
```



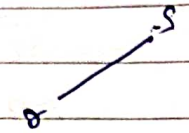
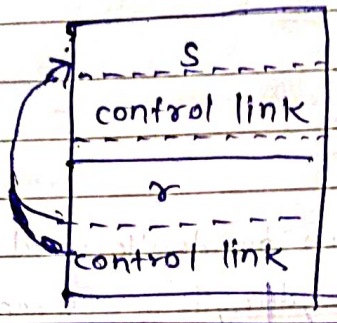
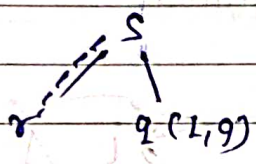
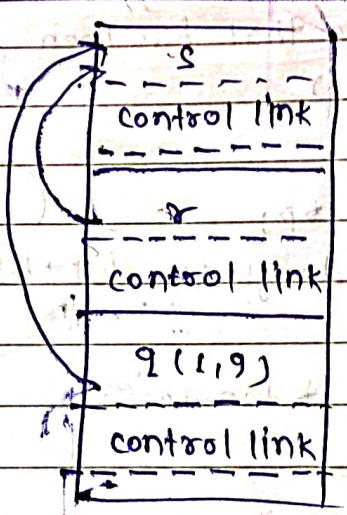
Variable length data :-

```
p()
{ A ==
  B ==
  C  q()
}
q()
{ ==
}
```



(3) Heap Allocation :-

Heap allocation is a piece of continuous storage as needed for activation records orth other objects. peices may be deallocated in any order so the heap will consist alternate areas that are free and in use.

Position of the Activation Tree	Activation Records in the heap	Remarks
		
		retained r

- Access to non-local Heaps :-

(1) Blocks

(2) Lexical Scope without nested procedure (not in syllabus)

(3) Lexical Scope with nested procedure (require accessing / access link)

(1) Blocks :- A block is a statement containing it's own local data declarations.

{ declaration statements }

```
main()
{
    int a=0;
    int b=0;
    {
        int b=1;
        {
            int a=1;
            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);
}
```

Step 1 :- Declarations & scope of variable

Declarations	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

Step 2 :- Resultant values of a & b :

2	1
0	3
0	1
0	0

Step 3 :- storage of variable

a_0
b_0
b_1
a_2, b_3

as each block avilable complementry at run time

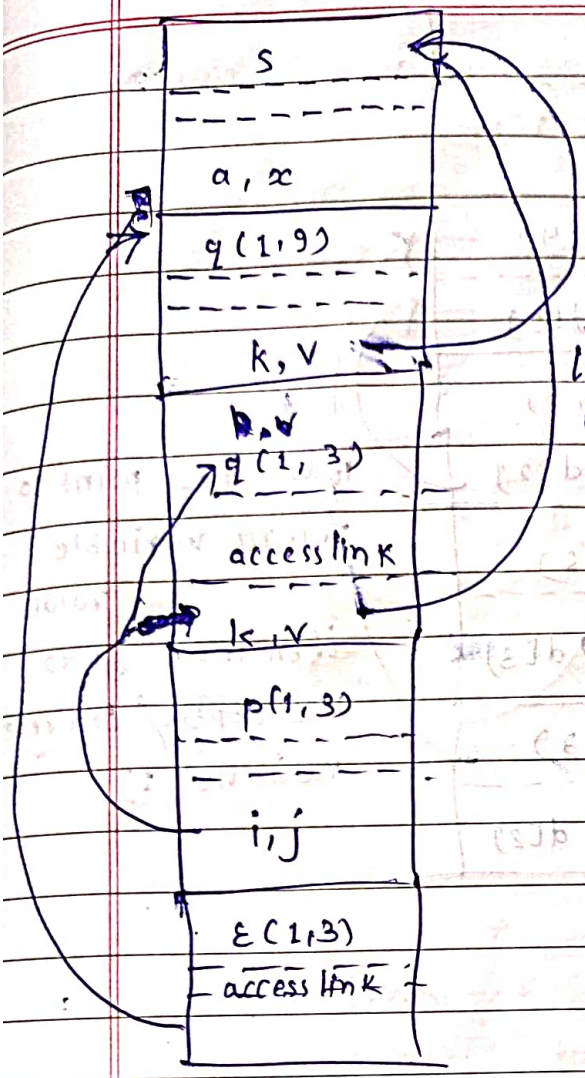
Lexical Scope with nested procedure

```
Program sort (input, output)
var a: array [0...10] of integer;
    x: integer
Procedure readarray;
    var i: integer
        begin .... a ..... end {readarray};
    Procedure exchange (i, j: integer);
        begin
            x := a[i]; a[i] := a[j]; a[j] := x
        end {exchange};
    Procedure quicksort (m, n: int)
        var k, v := integer;
        function partition (y, z: integer): integer;
            var i, j: integer;
            begin ... a ....
                ... v ...
                exchange (i, j);
            end {partition};
            begin end {quicksort};
    begin end {sort};
```

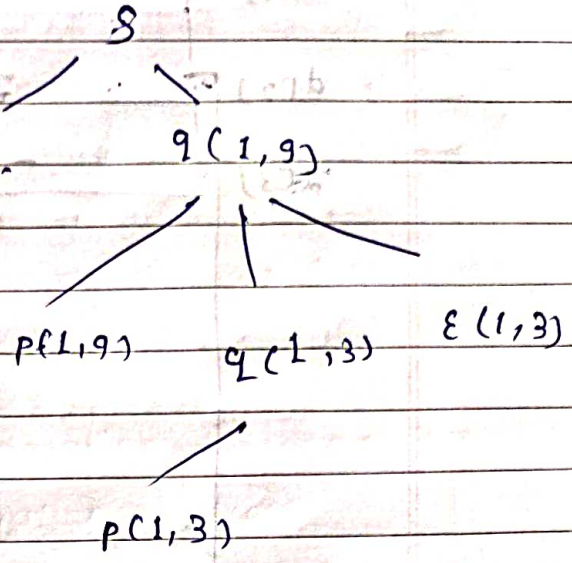
Step 1:- nesting procedures **Step 2:-** nesting depth (n_d)

- sort — 1
  - readarray — 2
  - exchange — 2
  - quicksort — 2
    - partition — 3

Step 3:- access link for finding storage for non-locals.



lower to upper depth we can go.



(*) Displays

faster access to non-locals than with access links can be obtained using an array d of pointers to activation records, which is called as Displays.

We mainatin the displays so that storage for a non-local at nesting depth (i) is in the activation record pointed by display element d[i]

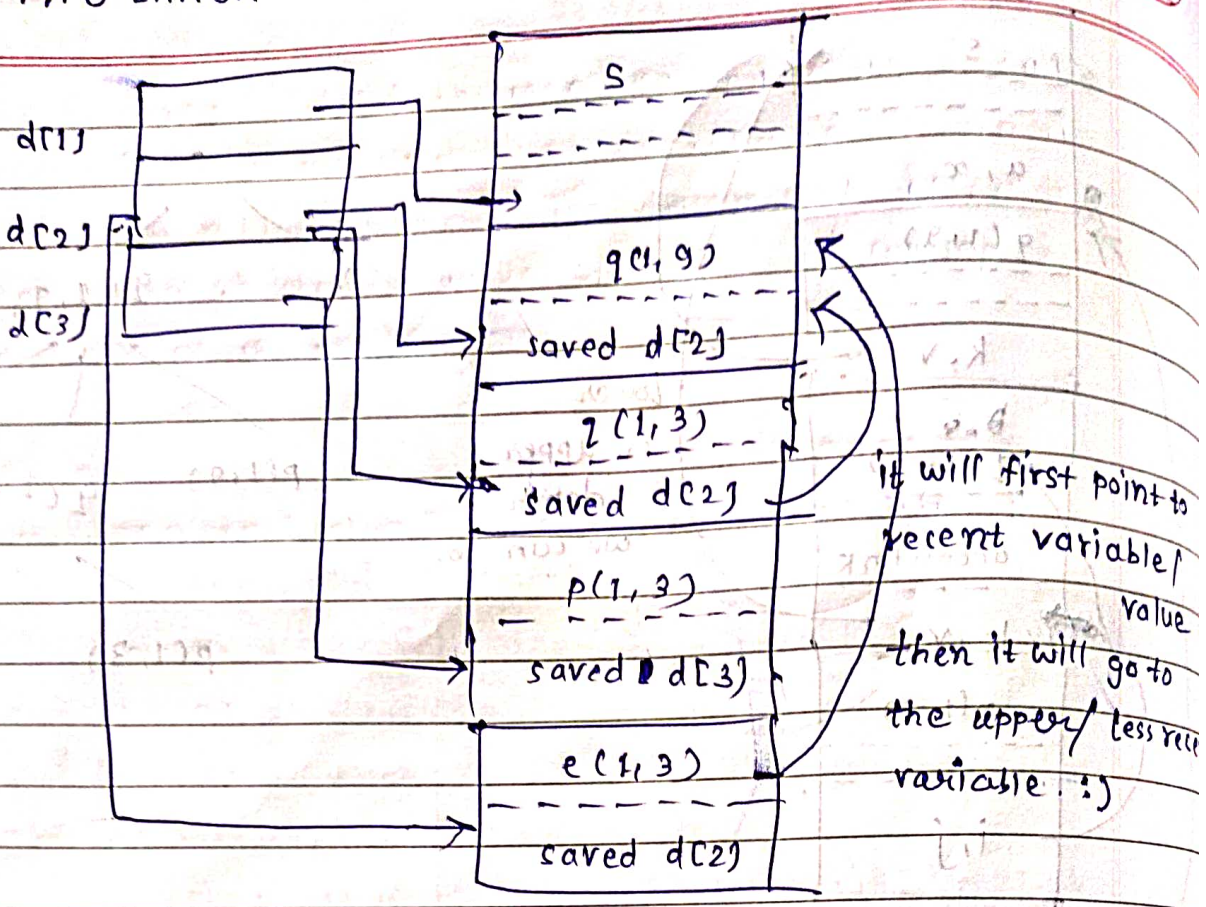
→ When a new activation record for a procedure at nesting depth i is set up then, so

(i) Save the value of d[i] in the new activation record and

(ii) Set d[i] to point the new activation record.

→ Just before an activation ends d[i] is reset to the saved values.

TYPO-ERROR



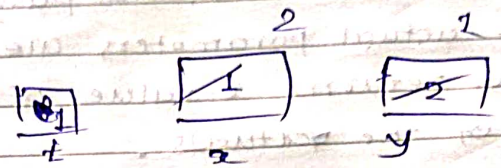
Parameter Passing

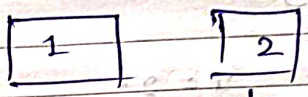
(1) Call by value
(2) call by Reference
(3) Copy & Restore
(4) call by name

Actual Parameters :- Variables specified in a function call.
Formal Parameters : Variables declared in called function.
l-value : Refer to the storage
r-value :- Value contained in the storage

(1) Call By Value :- caller evaluates the actual parameters.
→ Passes r-values of actual parameter to formals.

```
void swap ( int x, int y )
{
    int t;
    t = x;
    x = y;
    y = t;
}
void main ()
{
    int a = 1, b = 2;
    swap (a, b);
    printf ( "a = %d, b = %d", a, b );
}
```

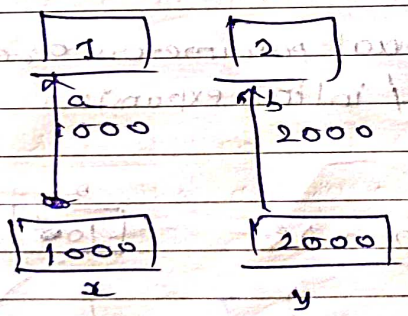




Q) Call By Reference (Call By Address) :-

→ Caller evaluates actual parameters.

→ Passes L-values to the formals.



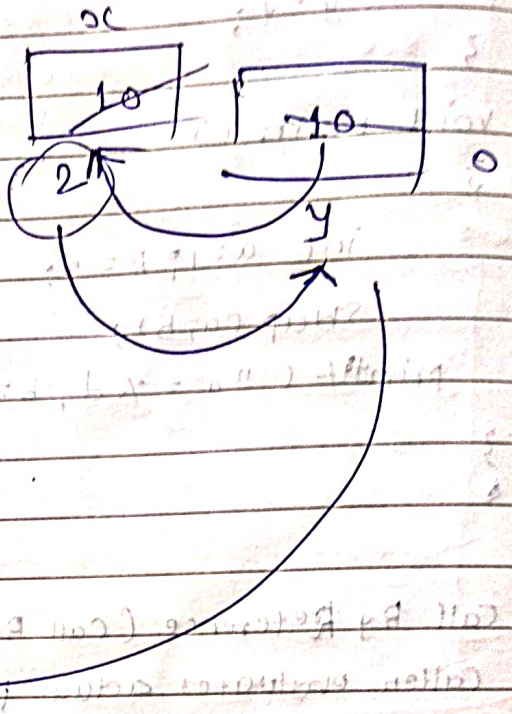
Change:

swap (&a, &b) ← caller.

```
void swap (int *x, int *y)
{
    int t;
    t
```

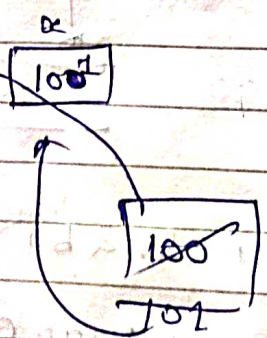
(3) Copy Restore :- Hybrid between call by value & call by Ref
→ Caller Evaluates actuals, and passes r-value to formals,
→ L-value of actual parameters are determined before the call,
→ When control returns r-values of formals r copied back into
L-values of the actuals.

```
int y;
copyrestore (int x)
{
    x = 2;
    y = 0;
}
main()
{
  y=10;
  copyrestore(y);
    print(y);
}
```



(4) Call by name : actual parameters are substituted for the for
→ macro expansion / inline expansion

```
int i=100;
void callbyname(int x)
{
    print(x);
    print(x)
}
main()
{
  callbyname(i=i+1);
}
```



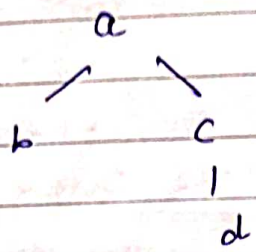
• as many time print

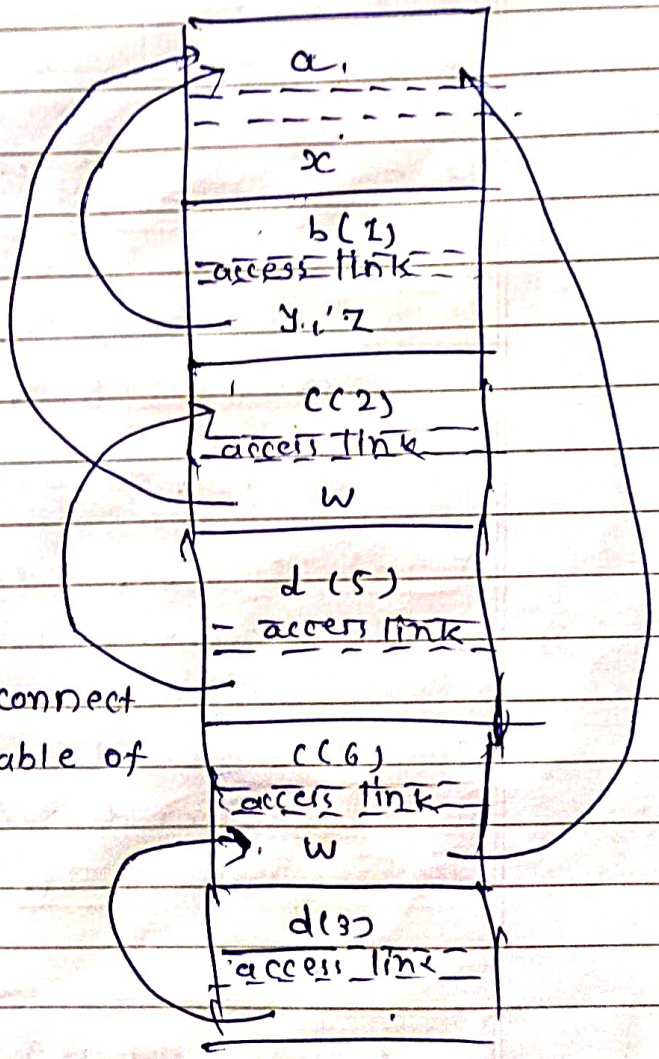
(*)

```
fun a(f)
    int x;
    fun b(i)
        int y, z;
        fun c(j)
            int w;
            fun d(k)
```

```
b(1)
c(2)
d(5)
c(6)
d(3)
```

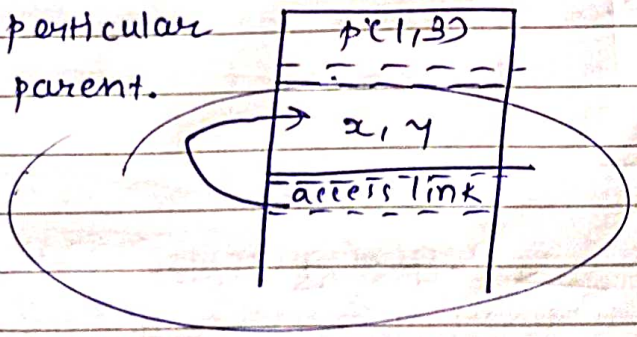
find the access-link ?



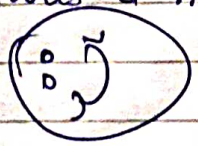


IMP NOTE:

In access-link topic we have to connect link (arrow) access-link to variable of particular parent.



→ ~~In all e~~ Previously in was a mistake - ~~Correct it~~



THE END