Syntax Directed Tree :- (Unit-3 part).

The value of an attribute at any node of parse tree is defined by a semantic rule associated with the production used as the node.

There are 2 diff. attributes :

(i) Synthesized att. (ii) Inherited attr.

* The value of a synthesized attribute at any node is computed from the values of the attribute at the child nodes of that node in the parse tree.
* The type of Inherited attribute is computed from the type of the attribute at the siblings and the parent nodes of that node.

Ex.

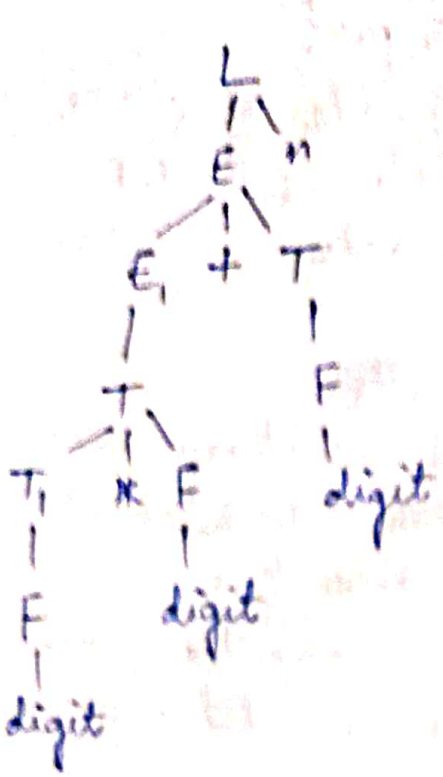
note: A syntax directed definition that uses only synthesized attributes is said to be 's'-attributed definition.

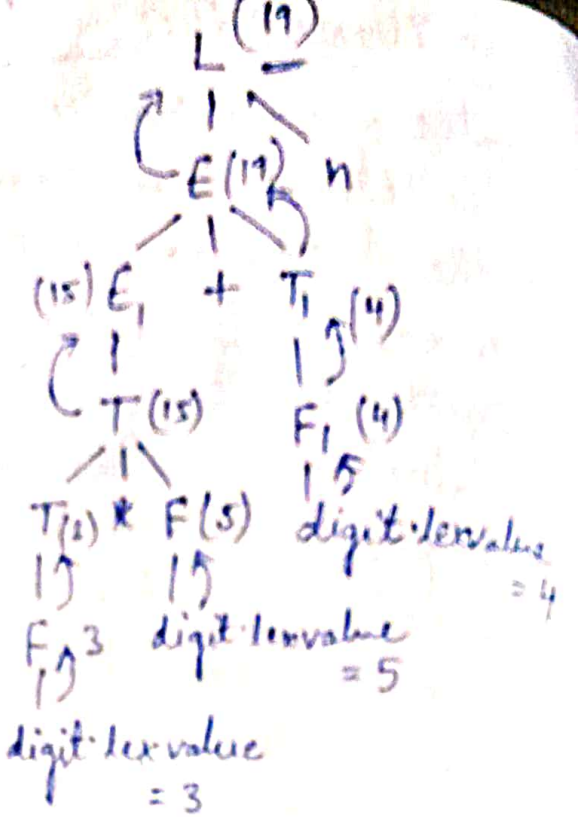
	SDD / Attributed grammar
$L \rightarrow E.n$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexvalue$

A parse tree representing the values of its attribute using SDD is known as annotated parse tree.

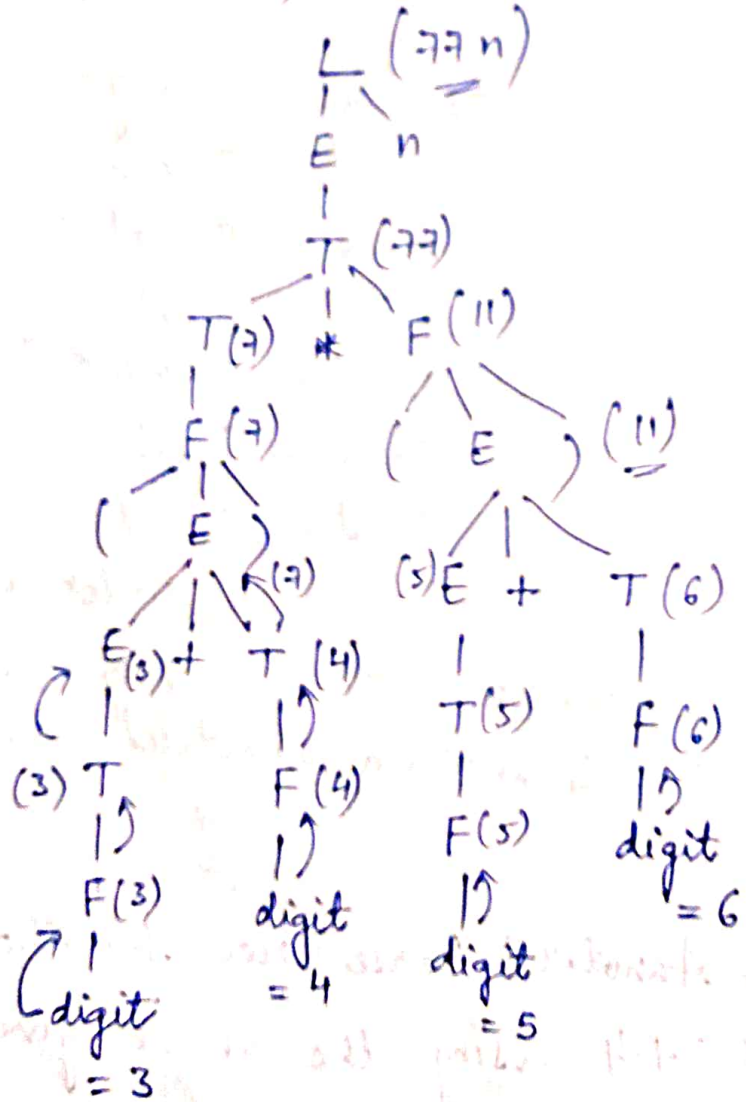
Q.

(1) Construct Annotated Parse Tree for the string 3 * 5 + 4 using the above given grammar.


L
E
n
E1
+
T
T1
*
F
F
digit
F
digit


(19)
L
E (19)
n
(15) E1
+
T1 (4)
T (15)
F1 (4)
T (3)
*
F (5)
digit · lexvalue = 4
F (3)
digit · lexvalue = 5
digit · lexvalue = 3

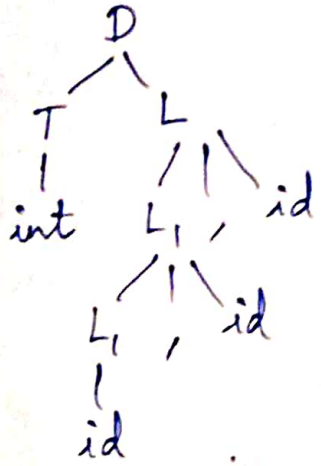
② (3+4) * (5+6) n

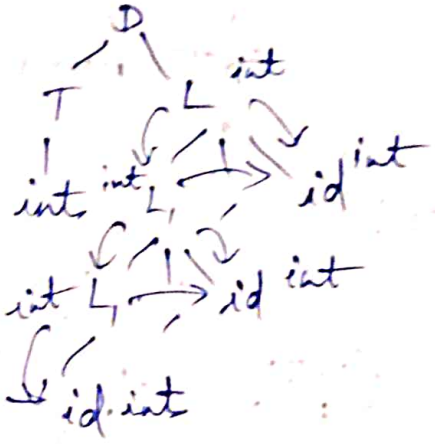

L (77 n)
E
n
T (77)
T (7)
*
F (11)
F (7)
(E)
(E) (11)
(7)
E (3)
+
T (4)
(5) E
+
T (6)
(3) T
F (4)
T (5)
F (6)
F (3)
digit = 4
F (5)
digit = 6
digit = 3
digit = 5

Inherited attributes :-

$D \to TL$	Att. Gr.	$L.in = T.type$
$T \to int$		$T.type = int$
$T \to real$		$T.type = real$
$L \to L_1, id;$		$L_1.in = L.in$ addtype(id.entry, L.in)
$L \to id$		addtype(id.entry, L.in)

Q: ① int a_1, a_2, a_3 ;





Dependency Graph :-

The inter dependencies among the inherited & synthesized attributes can be represented in a parse tree with the help of a directed graph known as "Dependency graph".

Q. $E \to E+T$
$E \to E-T$
$E \to T$
$T \to (E)$
$T \to id$
$T \to num$

Construct SDD for the given CFG and also construct the syntax tree for the expression a - 4 + c.

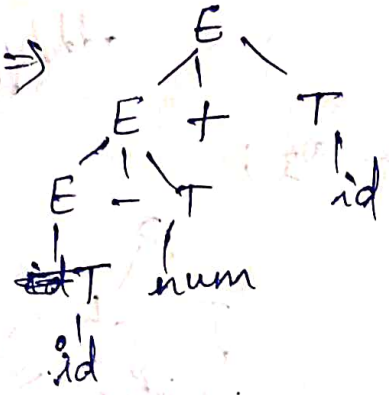
Sol^n:

SDD
~~E.val~~ → ~~E.val~~ + ~~T.val~~

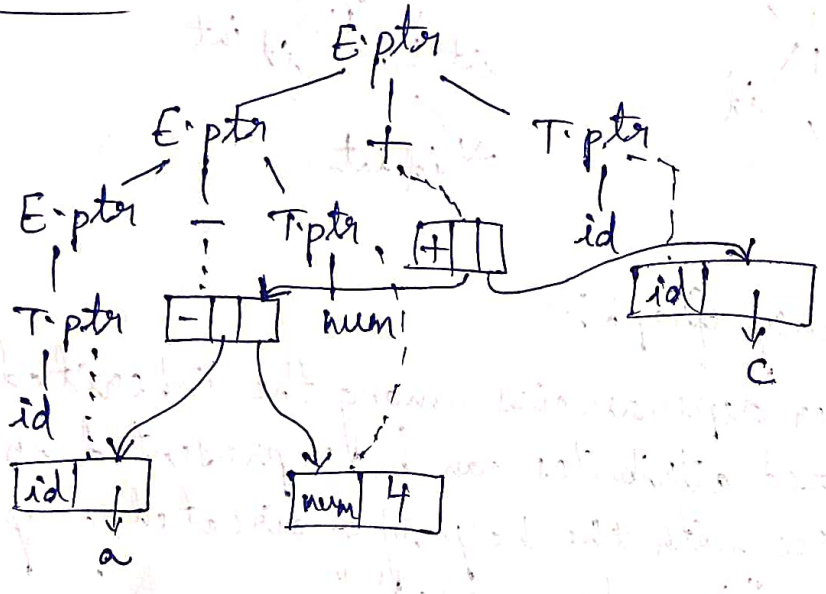
E.ptr = mknode('+', E.ptr, T.ptr)
E.ptr = mknode('-', E.ptr, T.ptr)
E.ptr = T.ptr
T.ptr = E.ptr
T.ptr = mkleaf(id, id.entry)
T.ptr = mkleaf(num, value)

a - 4 + c

Parse tree ⇒



Syntax Tree:



Bottom-up evaluation of S-attributed definition using a Parser ~~tag~~ Stack :-

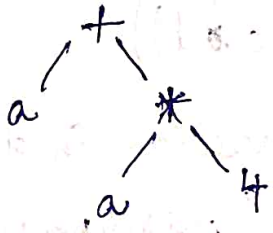
I/P	State	Val	Prod. used
3*5+4	—	—	—
*5+4	3	3	—
*5+4	F	3	F → digit
*5+4	T	3	T → F
5+4	T*	3	—

+4	T*5	3	—
+4	T*F	3□5	F → digit
+4	T	15	T → T*F
+4	E	15	E → T
4	E+	15	—
—	E+4	15	—
	E+F	15□4	F → digit
	E+T	15□4	T → F
	E	(19)	E → E+T

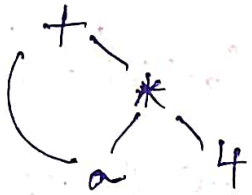
Directed Acyclic Graph (DAG):-

A DAG is a special type of tree used for representing expressions where any node representing a common sub expression has more than 1 parent.

ex:- ① a+a*4

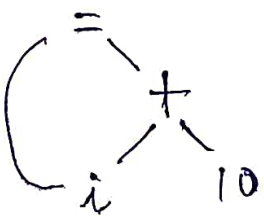


Parse Tree

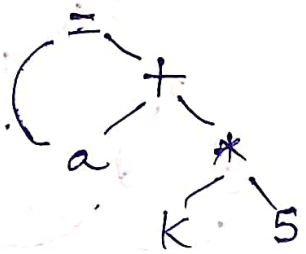


DAG

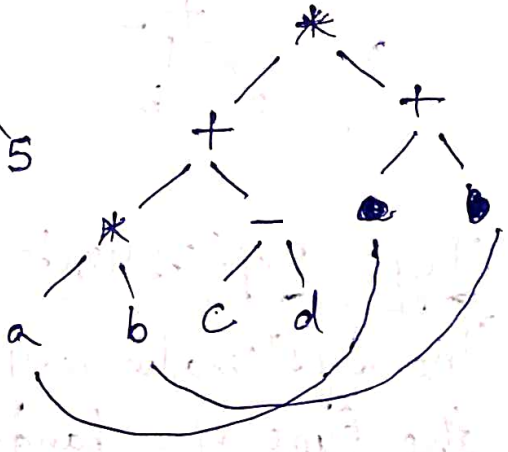
② i = i+10



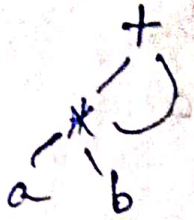
③ a = a+k*5



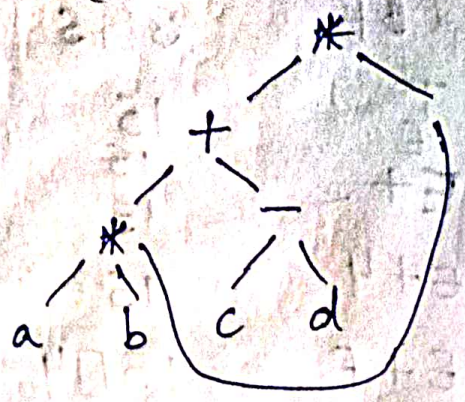
④ (a*b)+(c-d) * (a+b)



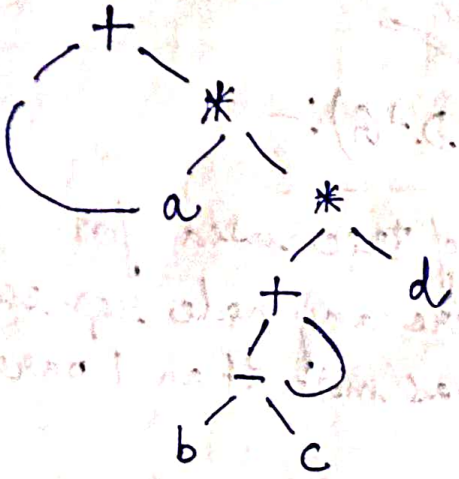
⑤ $(a*b)+(a*b)$



⑥ $((a*b)+(c-d))*(a*b)$



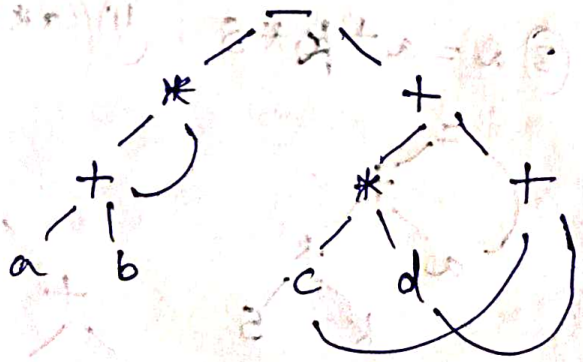
⑦ $a+(a*((b-c)+(b-c))*d)$



⑧ $((a+b)*(a+b))-((c*d)+(c+d))$

Generate 3 AC & represent expression in diff. forms of ICG & also in DAG. (Assignment).

DAG:



Type Expression:-

The systematic way of expressing ~~the~~/representing the type of a lang. construct is called a type expression.

Lang. Construct	Type Expression
→ Array	Array (I, T), where I is the index no. & T is the type.
ex: int a[100]	TE of a is array(100, int).
float sal[20]	TE of sal is array(20, float).
→ Structure	struct((member1 name type) x (m2 name type) x ...)
ex: struct student {char name[20], float avg} student details[100];	TE of student is, struct((name array(20, char)) x (avg float)) TE of student details is, array(100, student)
→ Pointer int *p	TE of pointer is, pointer(int)
→ Functions	TE of functions is, Type of Parameters → Return type
ex: int sum(int a, int b)	TE for "sum" is, int, int → int
float max(float sal[100])	TE of max is, array(100, float) → float

Type Conversion:-

* Implicit TC or coersion is done by the compiler.
* Explicit TC is done by the user.
* If there is an expression of the form, $E \rightarrow E_1$ op E_2, then the type of E is obtained from the types of E_1 & E_2.

- if $E_1.type = int$ and $E_2.type = int$
  then $E.type = int$
- if $E_1.type = float$ and $E_2.type = int$
  then $E.type = float$
- if $E_1.type = int$ and $E_2.type = float$
  then $E.type = float$
- if $E_1.type = float$ and $E_2.type = float$
  then $E.type = float$.

Type Checker:-

Type checker is a translation scheme in which the type of each expression from the types of its sub expressions.

Type Checking of Expressions:-

$E \longrightarrow literal$ { E.type = char. }

$E \longrightarrow digit$ { E.type = int }

$E \longrightarrow id$ { E.type = lookup(id.entry) }

$E \longrightarrow E_1 \text{ mod } E_2$ { E.type = if $E_1.type = int$ & $E_2.type = int$
then int
else type-error }.

$E \longrightarrow E_1 \text{ op } E_2$ { E.type = if $E_1.type = int$ & $E_2.type = int$
then int
else type error }.

In place of op we can use +, −, *, /, %.

Array reference

$E \longrightarrow E_1[E_2]$ { E.type = if $E_2.type = int$
& $E_1.type = array(range, t)$
then t
else type-error }.

Pointer

$E \rightarrow *E_1$ { E.type = if E_1.type = pointer(t)
then t
else type error }

Function call

$E \rightarrow E_1(E_2)$ { if E_1.type = $s \rightarrow t$ then t
else type error }

Symbol Table :-

* It stores the information about the identifiers.
* The info. will be about the name, type, scope, value, binding etc.

There are 2 diff. types of ST :

(i) Ordered ST (ii) Unordered ST

Ordered ST :

In this ST, the identifiers are entered in alphabetical order. The main adv. of this ST is that searching is easy. Dis. adv. is that, insertion is difficult.

Unordered ST :

The identifier names are inserted into the ST without following any order.

Adv : insertion is easy.

Disadv. : Searching is difficult.

Name representation in ST :-

1) Fixed representation (length) 2) Variable length rep.

LC	Type	Name
100	2 int	age
101	4 float	salary

FR

a	g	e	$	$	$	$	$
s	a	l	a	r	y	$	$
				-			

VR

a	g	e	$			
s	a	l	a	r	y	$

(or)

a	g	e	$	s	a	l	a	r	y	$
0	1	2	3	4	5	6	7	8	9	10

Start index	len
0	4
4	7

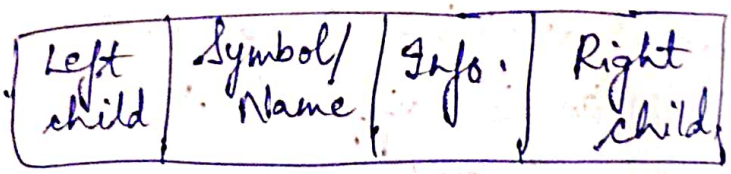
ST management :-

The data structures that are used to represent a ST are linear lists, binary trees, hash tabl

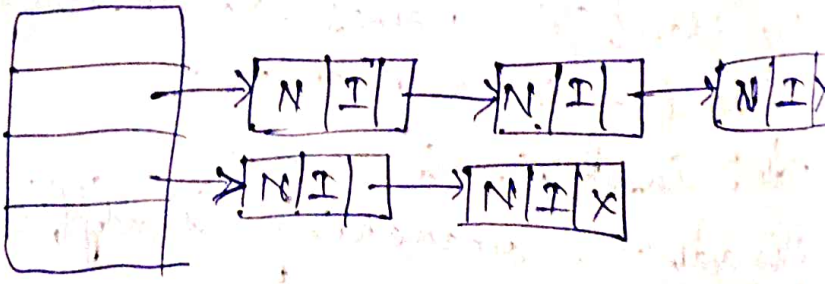
Linear lists :

Name	Info
N2	I2
:	:

Binary Tree :



Hash Tables :



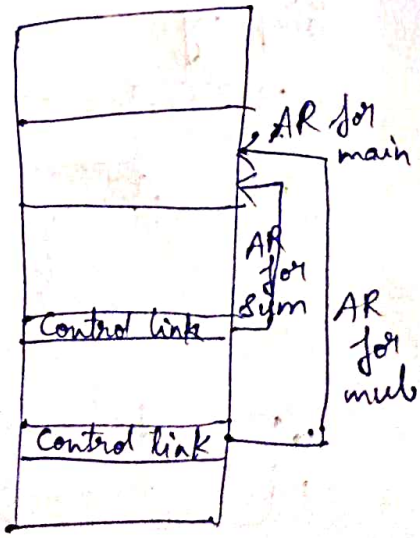
Activation Record :-

It consists of the info. about a single procedure.

Return value
Actual parameters
Control link
Access link
Saved Machine Status
Local Variables
Temporaries

- Return value is the value returned by the procedure.
- Actual parameters consists of the values of actual parameters
- control link points to the activation record of the calling procedure.
- Access link refers to the non-local data in other activation records.
- Saved Machine Status - before the procedure is called some control info. will be stored in the registers.
- Local Variables holds all the local variables of a procedure.
- Temporaries - all temporary variables generated by the compiler in order to execute the procedure are ~~stored~~ in this field.

```
main()                    float sum(x, y)
{ ==                      { ==
  a = sum(b, c);          }
  ==
  ==
  b = mul(c, f);          int mul(p, q)
  ==                      { ==
  ==                      ==
}                         }
```



MM

Access to non-local names:-

1) Static scope / Lexical scope.

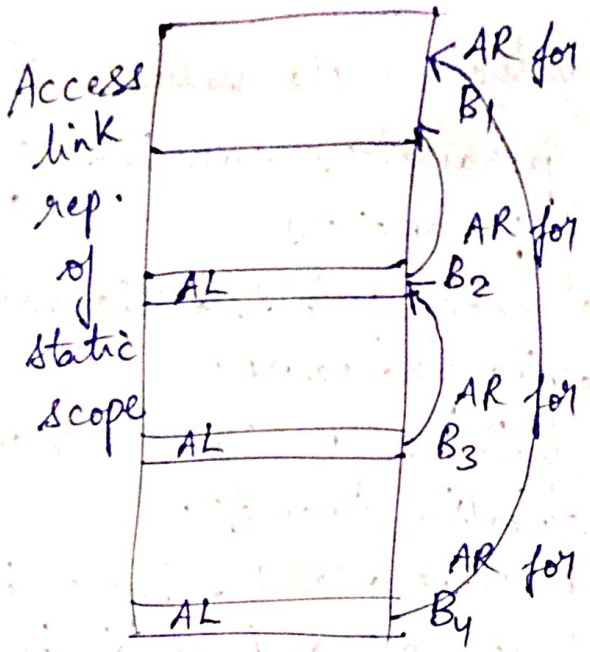
2) Dynamic scope
   - Deep Access
   - Shallow Access

Static / Lexical scope:-

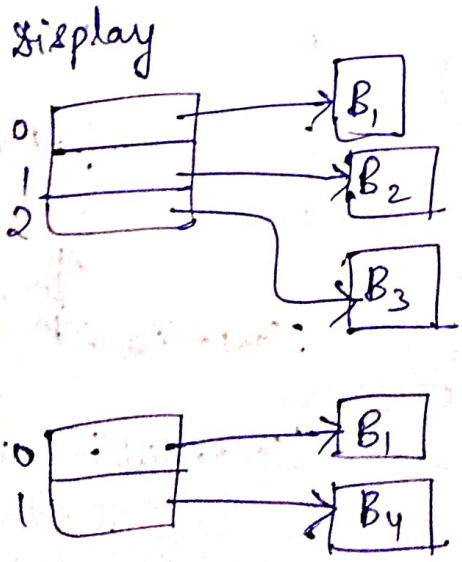
- Access link
- Display

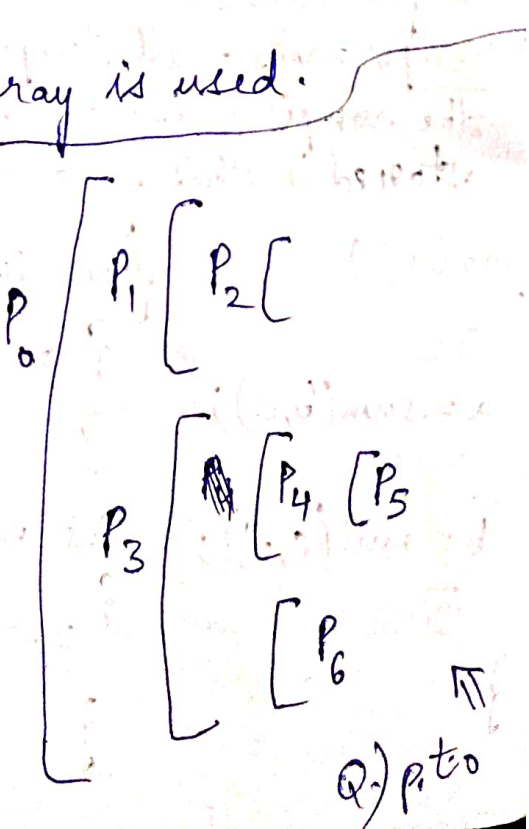
- In access link the activation record is represented using the access link.

ex:
```
B1()
{ ==
   B2()
   { ==
      B3()
      { ==
      }
   }
==
B4()
{ ==
}
}
```



- Display: An auxiliary array is used.





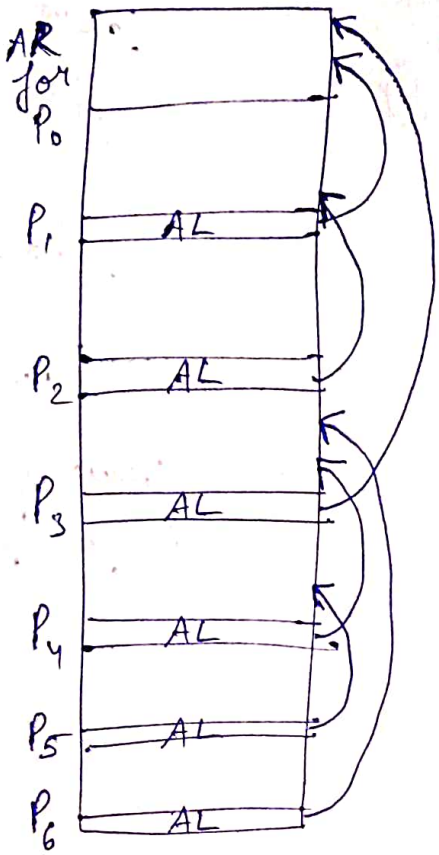
Q.) p.t.o

Q: Represent the access link & display representations for the given procedures.

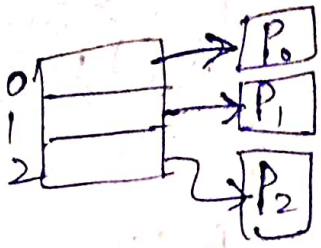
Soln:

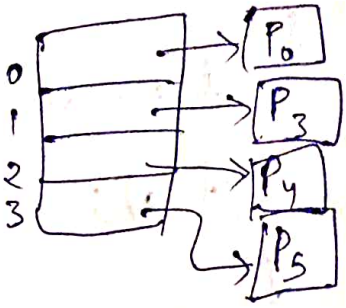
```
P0()
{ ==
  P1()
  { ==
    { P2()
      ==
    }
  }
  ==
  P3()
  { ==
    P4()
    { ==
      P5()
      { ==
      }
    }
    P6()
    { ==
    }
  }
}
```

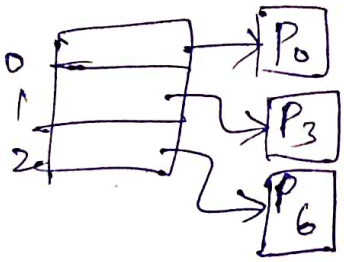
Access link:



Display:







Deep Access :-

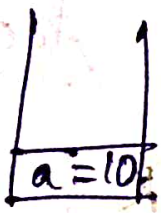
A stack is maintained in order to retrieve the most recent value of for that symbol.

Shallow Access :-

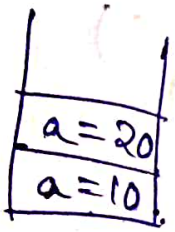
A central storage is kept with one slot for every variable.

```
B1()
{
  int a = 10;
  ==
  pf(a);  ------> (A)
  B2();   ------> (B)
  ==
}
B2()
{
  int a = 20;
  ==
  pf(a);
}
```

Deep Access

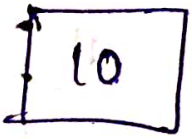


(A)



(B)

Shallow Access



Value of a in ~~for~~ B_1

Value of a in B_2