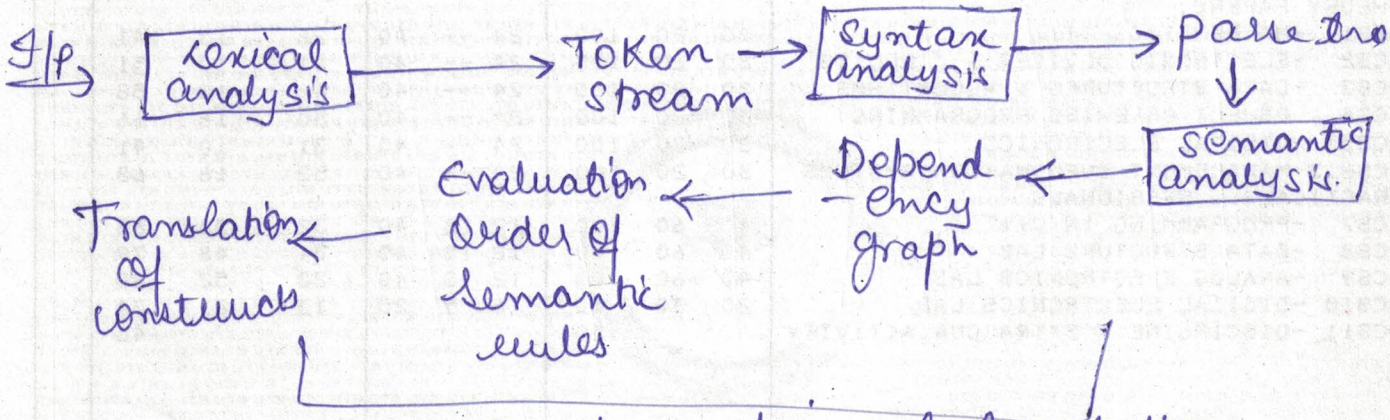
⇒ ~~conflicts~~

Syntax directed translation

1) programming language —requires translation→ programming constructs

2) translation —Semantic rules→ productions



3) Attributes → Information attached to grammar symbols

$F \rightarrow digit \quad \{ F.val := digit.lexval \}$

4) Semantic rules → values of attributes are computed using Semantic rules

$T.val = T_1.val * F.val$

$T.val = 2 * 3.$

$T.val = 6.$

Two type of Attributes

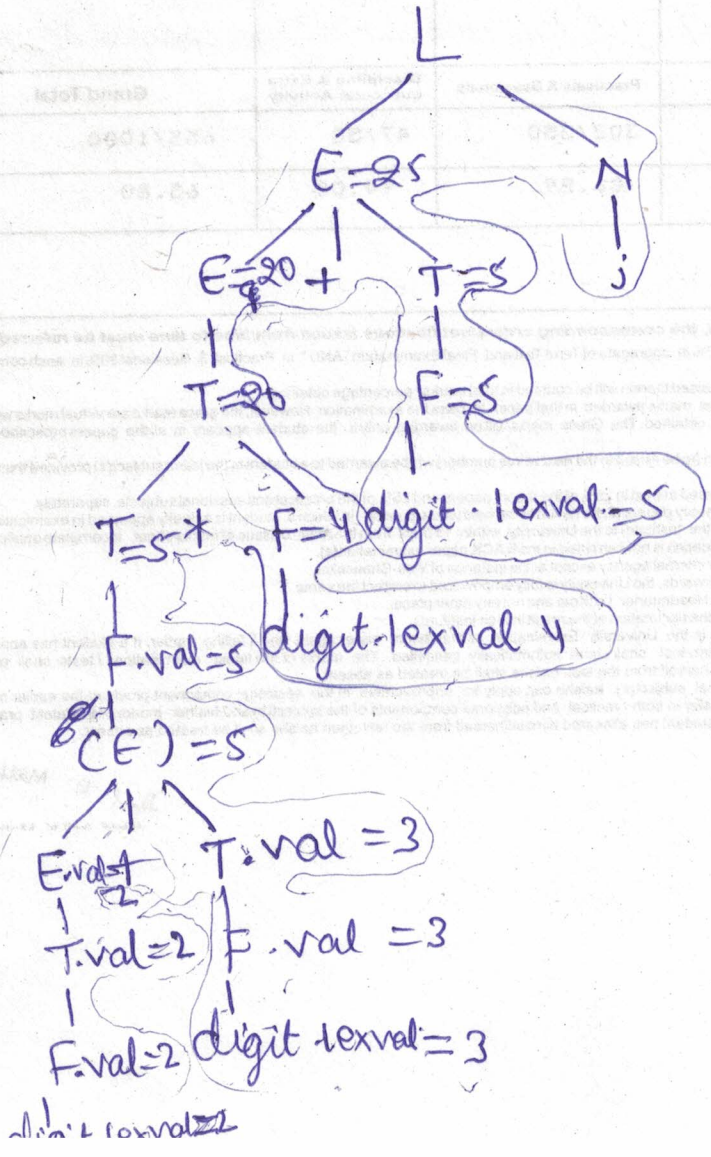
i) Synthesized attributes

ii) value = value of atributes of children of that node in parse tree

eg

$L \to EN$	{ print E.val }
$E \to E+T$	{ E.val := E_1.val + T.val }
$E \to T$	{ E.val = T.val }
$T \to T*F$	{ T.val = T_1.val * F.val }
$T \to F$	{ T.val = F.val }
$F \to (E)$	{ F.val = E.val }
$F \to$ digit	{ F.val := digit.lexval }
$N \to ;$	{ Ignored by lexical analyzer as ; is terminating symbol.

(2+3) * 4 +5
= 25



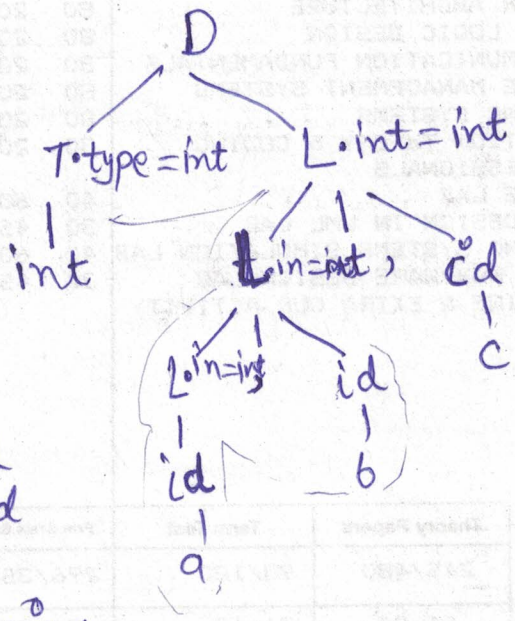
2) parse tree = bottom-up fashion

Each node values of attributes are shown = Annotated parse tree

⇒ Inherited attributes

1) Value is computed from value of attribute of siblings & parent node of that node.

Ex $D \to TL$
$T \to int$
$T \to real$
$L \to L_1, id$
$L \to id$



⇒ Two child have type attached with it would
⇒ be computed from child node
→ L & L_1 will have inherited from its siblings

$D \to TL$ { L.in = T.type }
$T \to int$ { T.type = integer }
$T \to real$ { 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).

Dependency graphs

→ order or sequence of sequence of evaluation of subproblems

→ help us determine how those values are computed

→ steps to construct dependency graph

1) put are semantics rules in the form

$$b := f(c_1, c_2, \ldots c_k)$$

2) Dependency graph has a node for each attribute & an edge from attribute c to attribute b if b dep ends on c

3) If there is some production $X \to YZ$, then corresponding semantic rule is $X.x := f(Y.y, Z.z)$

$X.x$ = Synthesized attributes

$Y.y$ & $Z.z \Rightarrow$ Inherited attributes

4) there would be edges from $Y.y$ & $Z.z$ to $X.x$ & solid lines show dependencies b/w the nodes

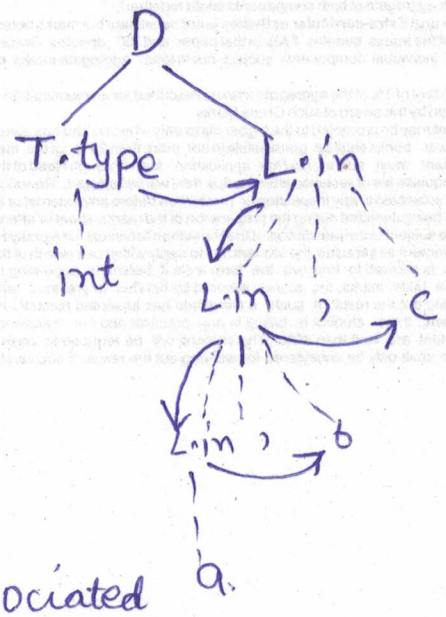
Ex. $D \to TL$

$T \to int$

$T \to real$

$L \to L_1, id$

$L \to id$



1) type int is obtained after analyzing I/p token.

2) L.in is assigned the type in from its sibling T.type.

3) Entry for identifier c gets associated with type int

4) L.in is assigned type int from its parent

5) identifier b also get associated with type int

⇒ Construction of Syntax Trees

1) Operators ⇒ nodes
2) id & num ⇒ leafs

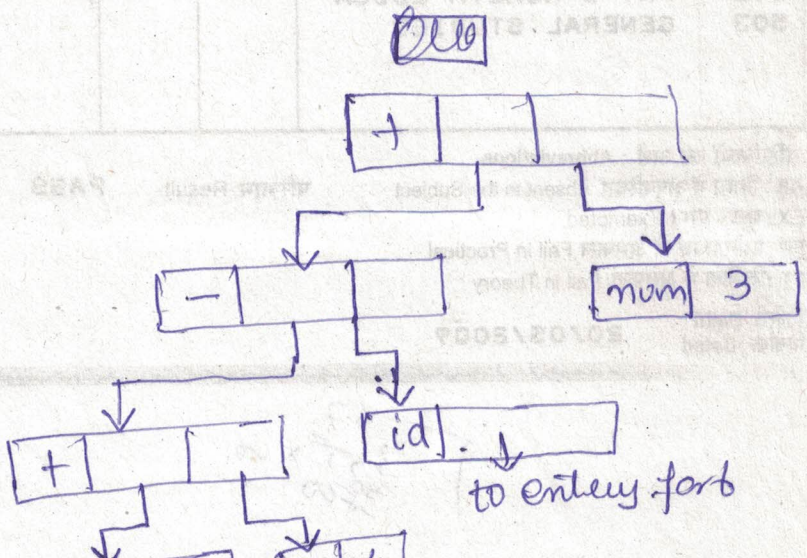
- functions

1 mknode(op, left, right) → operator-label
operands - left & right

2 mkleaf(id, entry) → id - identifier = label.
entry - value

3 mkleaf(num, value) → num - label
val - number

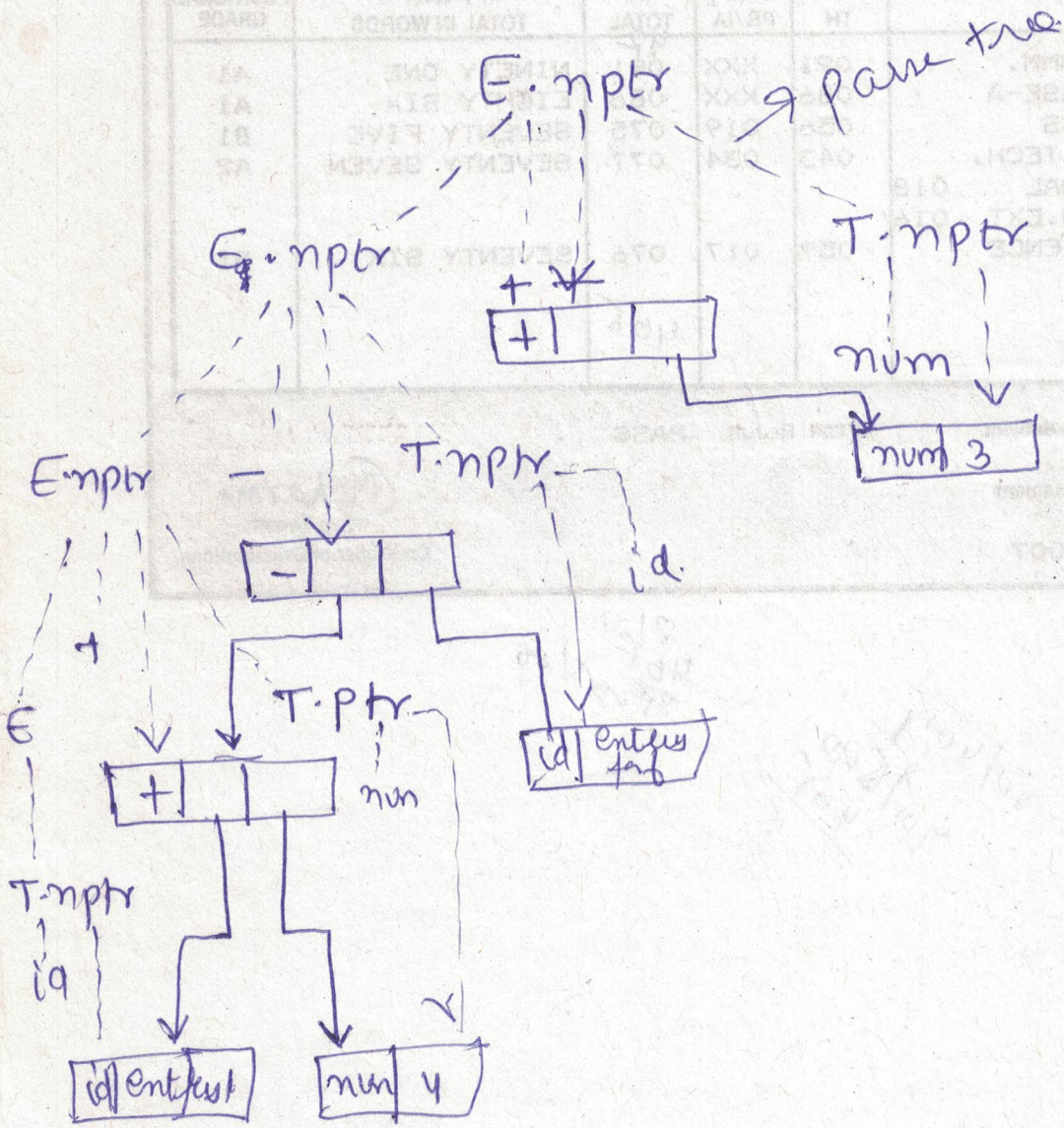
ex $E \to E_1 + T$
$E \to E_1 - T$
$E \to T$
$T \to (E)$
$T \to id$
$T \to num$



sip → a+4-b+3

1) p_1 = mkleaf(id, entry a)
2) p_2 = mkleaf(num, entry 4)
3) p_3 = mknode(+, p_1, p_2)
4) p_4 = mk~~node~~leaf(id, entry b)
5) p_5 = mknode(-, p_3, p_4)
6) p_6 = mkleaf(num, entry 3)
7) p_7 = makenode(+, p_5, p_6)

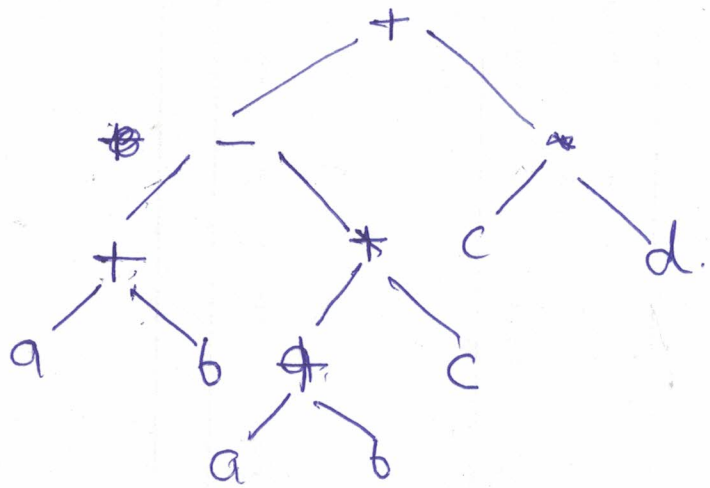
Production	semantic rules
$E \to E_1 + T$	$E.nptr = mknode(+, E_1, T)$
$E \to E_1 - T$	$E.nptr = mknode(-, E_1, T)$
$E \to T$	$E.nptr = T.nptr$
$T \to (E)$	$T.nptr = E.nptr$
$T \to id$	$T.nptr = mkleaf(id, id.entry)$
$T \to num$	$T.nptr = mkleaf(num, num.val)$



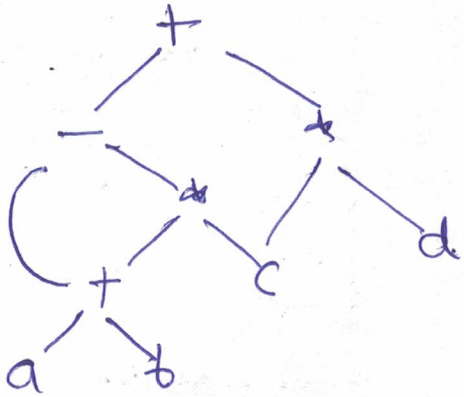
⇒ DAG (Directed acyclic graph)

1) a syntax tree in which there is single node for common sub expression
2) ~~Common~~ nodes have more than one parent node in syntax tree
3) DAG is a directed graph that contains no cycles

Ex (a+b) − (a+b) * c + c * d.



⇒



- P_1 = mkleaf (id, entry a)
- P_2 = " (id, " b)
- P_3 = mknode (+, P_1, P_2)
- P_4 = mkleaf (id, entry a)
- P_5 = " (" , " b)
- P_6 = mknode (+, P_4, P_5)
- P_7 = mkleaf (id, entry c)
- P_8 = mknode (*, P_6, P_7)
- P_9 = mknode (−, P_3, P_8)
- P_{10} = mkleaf (id, entry c)
- P_{11} = " (" , " d)
- P_{12} = mknode (*, P_{10}, P_{11})
- P_{13} = " (+, P_{12}, P_9)

⇒ S-attributed SDT

1) Uses only synthesized attribute

2) $A \to BCD$
$A.S = f(B.s | C.s | D.s)$ } synthesized
$C.i = D.i$
$C.i = A.i$ } inherited
$C.i = B.i$

3) Semantic actions are placed at right end of production

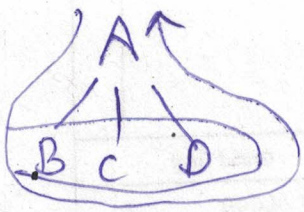
$A \to BCC\{\}$

4) Attributes are evaluated during BUP

5) Less powerful.

6) More restricted

7) Post fix SDT



L-attributed SDT

1) Uses both synthesized as well as inherited attributes,
Ex $A \to XYZ \{Y.S = A.S,\ Y.S = X.S\}$
Each inherited attribute is restricted to inherit from Parent or left sibling

2) Semantic action can be placed anywhere on RHS

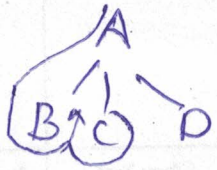
$A \to \{\}\ BC$
$\quad / D\ \{\}\ E$
$\quad / FG\ \{\}$

3) Attributes are evaluated

4) More powerful

5) Flexible

6) Infix or Prefix.



Ex 1) $A \to LM \{L.i = f(A.i);\ M.i = f(L.s);\ A.s = f(M.s);\}$
2) $A \to QR \{R.i = f(A.i);\ Q.i = f(R.i);\ A.s = f(Q.s)\}$
3) $A \to BC \{B.S = A.s\}$
L-attributed

L-ator

⇒ S-attributed translation.

i) Bottom-up parser makes use of stack to hold information about subtrees that have already been parsed.

ii) Array 'state' & 'val' are used to implement stack

iii) 'state' is pointer to LR(1) parsing table.

iv) 'val' is the value of attribute

v) if state[i] represent grammar symbol 'A' then val[i] will hold the value of the attribute associated with node of the tree for 'A'

vi) for indicating top of the stack, pointer 'top' is used.

→ # translation scheme

→ for production $A \to XYZ$

Each synthesized attribute is evaluated before reduction

A → XYZ

state	value
X	X.x
Y	Y.y
top → Z	Z.z

Before reduction

⇒

state	val
A	A.a

→ If some symbol has no attribute entry for its val array is made undefined

→ reduce RHS to LHS, top is decremented by 2 & state[TOP] contains LHS i.e state symbol on LHS of production & its val[TOP] contains the value of corresponding attribute.

	Code fragment
$L \to EN$	Print (val[top])
$E \to E+T$	val[top] = val[top-2] + val[top]
$E \to T$	
$T \to T_1 * F$	val[top] = val[top-2] × val[top]
$T \to F$	
$F \to (E)$	val[top] = val[top-1]
$F \to$ digit	
$N \to ;$	

for $E \to E_1 + T$

~~Avtop+~~ val[top-1] contain '+' & symbol for $T \to T_1 * F$

val[top] contains ~~$F \to E$~~ right parenthesis for $F \to (E)$

I/p string $2*3+4$, content of stack at each step

→ When a parser shift a digit, the token is placed state(top) & its value is placed in val[top]

I/p	state	val	production	position
2*3+4;	—	—		~~Shift~~
*3+4;	2	2		Shift
*3+4;	F	2	F → digit	
*3+4;	T	2	T → F	Shift
3+4;	T*	2_		
+4;	T*3	2_3		Shift
+4;	T*F	2_3	F → digit	
+4;	T	6	T → T*F	Shift
+4;	~~T~~ E	6	E → T	Shift
4;	E+	6_		Shift
;	E+4	6_4		
;	E+F	6_4	F → digit	
;	~~E+T~~	6_4	T → F	
;	E	10	E → E+T	Shift
	E;	10_		
	EN	10_	N → ;	
	L	10	L → EN	

⇒ L- attributed definition.

→ For symbol on the right side of production, an inherited attribute must be computed in an action before that symbol is used.

→ A synthesized attribute for a symbol must not referred in the action before that symbol in the production

→ for non-terminal on left side of production, synthesized attribute is computed at last, after all the attributes have been computed. thus the action for such attribute is put at the right most of the production rule

Production	semantic rule
Ex $T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$

Translation scheme

$T \rightarrow T_1 * F \{T.val = T_1.val \times F.val\}$

val ⇒ synthesized attributes

action computes for non-terminal T on LHS of productions

action is on last

	Semantic rule
Ex $A \rightarrow A_1 A_2$	$A_1.in = A.in$
	$A_2.in = A.in$
	$A.s = MAX(A_1.s, A_2.s)$

Translation scheme

$A \rightarrow \{A_1.in := A.in\} A_1 \{A_2.in = A.in\} A_2 \{A.s = MAX(A_1.s, A_2.s)\}$

Or $E \to E+T \mid E-T \mid T$

$T \to (E)$

$T \to num$

$E \to TE'$	$E'.i = T.val$ $E.val = E'.s$
$E' \to +E_1'T$	$E'.i = E'.i + T.val \mid E'.s = E_1'.s$
$E' \to -E_1'T$	$E'.i = E'.i - T.val \mid E'.s = E_1'.s$
$E' \to \varepsilon$	$E'.i = E'.s$
$T \to (E)$	$T.val = E.val$
$T \to num$	$T.val = num.val$

$E \to T \{E'.i = T.val\} E' \{E.val = E'.s\}$

$E' \to +T \{E'.i = E'.i + T.val\} E' \{E'.s = E_1'.s\}$

$E' \to -T \{E'.i = E'.i - T.val\} E' \{E'.s = E_1'.s\}$

$E' \to \varepsilon \{E'.s = E'.i\}$

$T \to (E) = \{T.val = E.val\}$

$T \to num \{T.val = num.val\}$

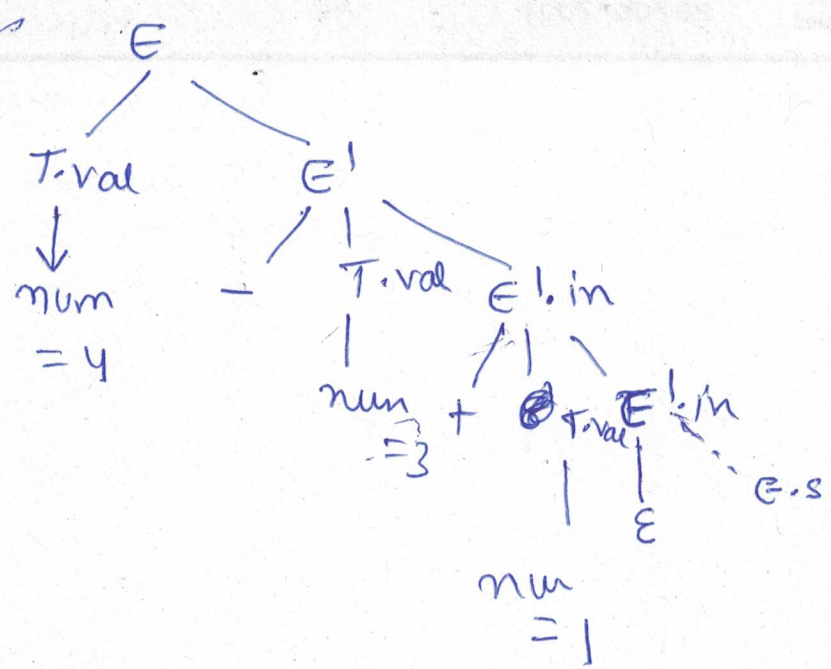
⇒ Construction of syntax tree for translation

Ip 4-3+1

$E' \to +T \{R_1.i =$ mknode('+', $R.in, T.val)\}$ $R_1 \{R.s = R_1.s\}$

$R \to -T \{R_1.i' =$ mknode('-', $R.in, T.val)\}$ $R_1 \{R.s = R_1.s\}$

$T \to num \{T.val =$ mkleaf(num, num.lexval)



Unit-4

Run time storage organization

Procedure → A procedure definition that associates an identifier with a statement. The identifier is the procedure name & statement is a procedure body

for eg following is the definition of procedure name readarray

```
procedure readarray
  var i: integer
  begin
  for i = 1 to 9 do read a[i]
  end.
```

When a procedure name appears within an executable statement, the procedure is said to be called at that point.

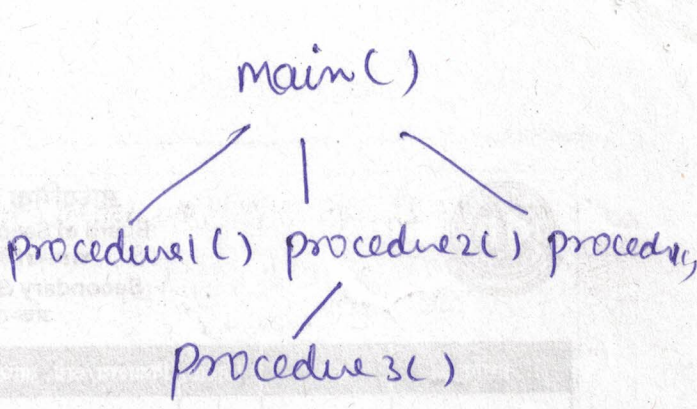
⇒ Activation Trees

An activation tree is used to depict the way control enters & leaves activations in an activation tree

1. Each node represents an activation of a procedure
2. root represents the activation of main program
3. The node for a is the parent of node for b if & only if control flows from a to b.
4. the node for a is to the left of node for b if & only if the lifetime of a occurs before the lifetime of b

for eg

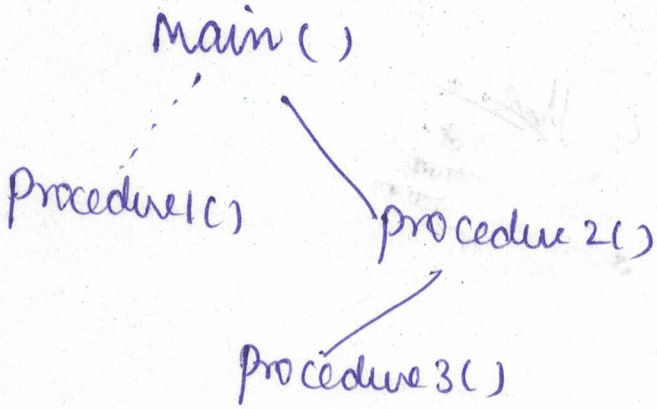
Enter procedure 1 ()
Leave " 1 ()
Enter " 2 ()
" " 3 ()
Leave " 3 ()
" " 2 ()
Enter " 1 ()
Leave " 1 ()
Execution terminates



⇒ Control stack

1) Control stack is used to keep track of live procedure activations. The idea is to push the node for an activation onto control stack as the activation begins & to pop the node when the activation ends

2) The contents of the control stack are related to paths to the root of the activation tree when node n is at the top of control stack, the stack contains the nodes along the path from n to the root



Solid lines shows the live procedures & the dashed lines show the termination of the procedure.

⇒ Scope of declaration

A declaration is a syntactic construct that associates information with a name

Declarations may be explicit, such as:

```
var i : integer
```

or they may be implicit. Example any variable name starting with I is assumed to denote an integer

The portion of the program to which declaration applies is called scope of declarations

```
main()
{
int x,y,z
- - - -
- - - -
if (y<=100)
{
int x, sum
}
}
```

← scope level 1

→ scope level 2.

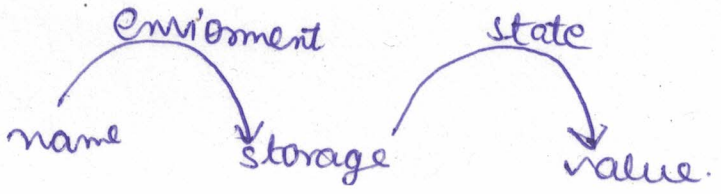
⇒ Binding of Names

→ Even if each name is declared once in a program the same name may denote different data objects at run time

→ Data object corresponds to storage location that holds values

→ The term environment refers to a function that maps a name to a storage location

→ The term state refers to a function that maps a storage location to the value held there



when an environment associates storage locations s with a name x, we say that x is bound to s. this asso - ciation is referred to as a binding of x.

for eg

let the storage addres 2000 associates with the variab x which having a value 1. Now if there is a program which adds 1 to x i.e $X=X+1$ then value of x becomes 2 but after assigning it a value of 2, x associates with the same storage location i.e 2000 from above we can say x bounds to s where x is a name & s is a storage location.

⇒ Activation records

- procedures calls & returns are usually managed by a run time 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, the later activation has its record at the top of the stack.
- Contents of activation record vary with the language being implemented. The diagram below shows the content of activation record.

Actual param	→ used by calling procedures, placed in registers
Returned values	→ hold result of a function call.
control link	→ activation record of caller.
Access links	→ to locate data needed by the called procedure but found elsewhere
saved m/c state	→ info about state of m/c just before call to procedure
local data	→ variables local to the procedures ~~are stored~~
Temporaries	→ temporary values those arising from eval of expression

for eg f1(int a)

```
{
int b = 10
return (a+b);
}
f2 (int b)
{
return (b + f1(b))
}
main()
{
f2(4)
}
```

Step1 ~~f(2)~~ f2(4) called

b = 4

Step 2. Activation record for f_1 is pushed onto stq

f2	b = 4
f1	a = 4
	b = 10

step3 f1(4) returns 14

Activation record f_1 popped from stack

b = 4	(return value = rv)
rv = 14	

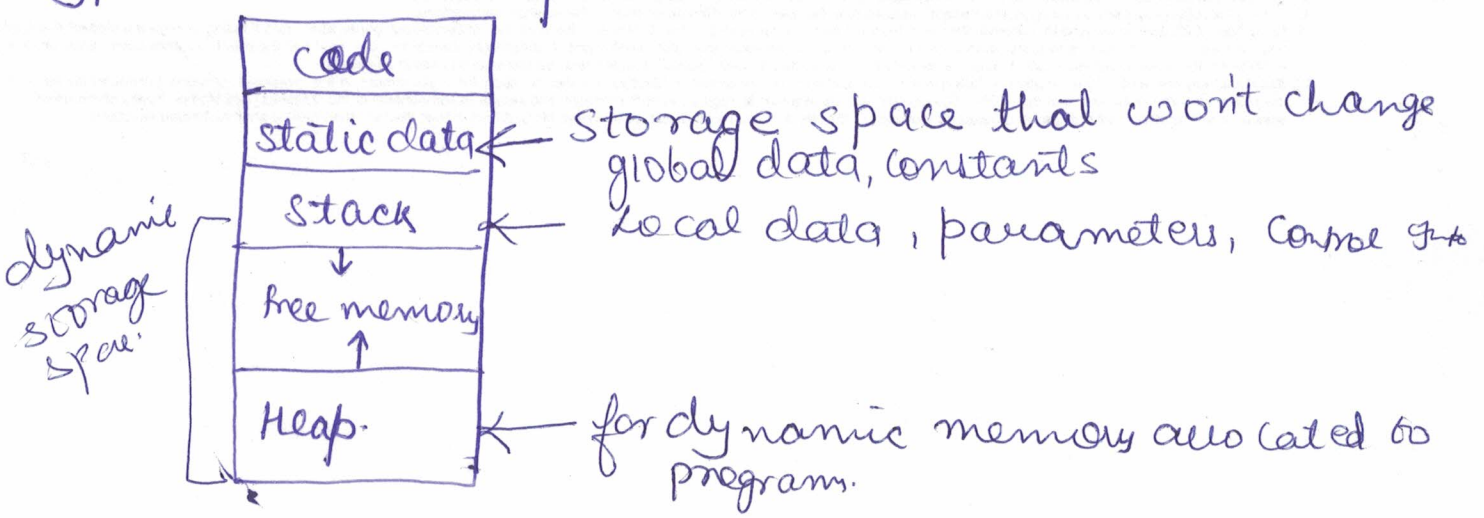
step4 . f2(4) returns value 18.

⇒ storage organisation

→ The executing target program runs in its own logical address space in which each program value has a location

→ The management & organization of this logical space is shared b/w compiler, OS & target machine - The OS maps logical address into physical address, which is usually spread throughout memory.

→ Typical subdivision of run-time memory



→ Run-time storage comes in blocks, where a byte is smallest unit of addressable memory. four bytes form a machine word. Multibyte objects are stored in consecutive bytes & given the address of first byte.

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

→ A character array of length 10 needs only enough bytes bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.

→ This unused space due to alignment consideration is referred to as padding.

→ The size of some program objects may be known at run time & may be placed in an area called static.

→ The dynamic areas used to maximize the utilization of space at run time are stack & heap.

⇒ Storage allocation strategies

Different storage allocation strategies are

1) static allocation — lays out storage for all data objects at compile time

2) stack allocation — manages the run time storage as a stack.

3) Heap allocation — allocates & deallocates storage as needed at run time from a data area known as heap.

⇒ static allocation

i) names are bound to storage as the program is compiled, so there is no need for a run time support package.

2) since bindings do not change at run-time, every time a procedure is activated, its names are bound to same storage locations

3) Therefore values of local names are retained across activations of a procedure. i.e when control returns to a procedure the values of locals are same as they were when control left the last time.

4) from the type of a name, the compiler decides the amount of storage for the name & decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on

⇒ Stack allocation of space

1) All compilers for languages that use procedures, functions or methods as unit of user-defined actions manage at least part of their run-time memory as a stack.

2) Each time a procedure is called, space for its local variables is pushed on to a stack, & when the procedure terminates, that space is popped off the stack.

Calling sequences

→ Procedures called are implemented in what is called as calling sequences, which consist of code that allocates an activation record on the stack & enters info into its fields

→ A return sequence is similar to code to restore state

of machine so the calling procedure can continue its procedure execution after the call.

→ the code in calling sequence is often divided b/w the calling procedure (caller) & procedure it calls (callee)

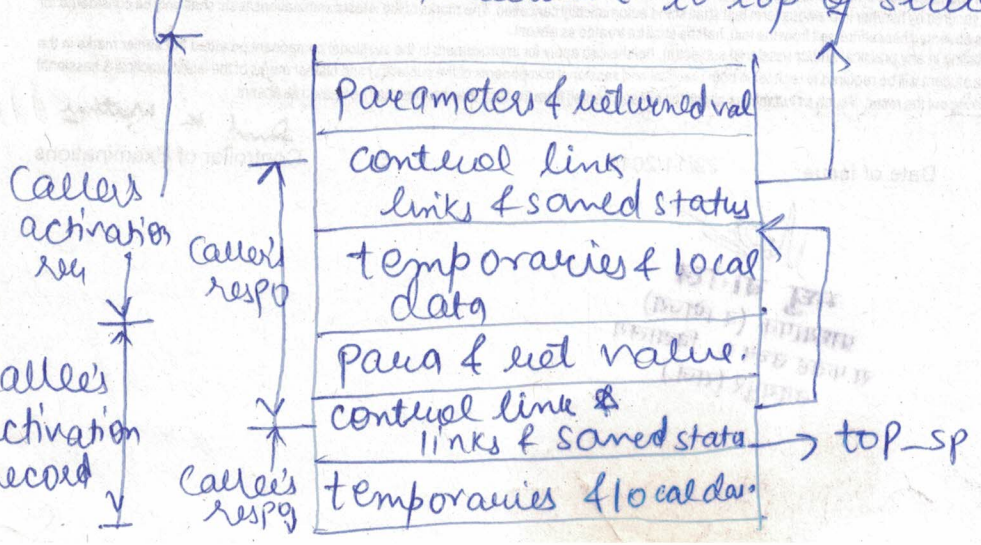
→ When designing calling sequences & the layout of activation records, the following principles are helpful.

1) values communicated b/w caller & callee are generally placed at the begining of callee's activation record, so they are close as possible to the caller's activation record.

2) fixed length items are generally placed in middle. Such item typically include the control link, access link & machine status field.

3) item whose size may not be known early enough are placed at the end of activation record. for eg dynamic size array

4) We must locate top of stack pointer judiciously. A common approach is to have it point to end of fixed length fields in the activation record. Fixed length data can then be accessed by fixed offsets, known to the intermediate code generator, relative to top of stack pointer.



$E \rightarrow T \{ R.i = T.val \} R \{ E.val = R.s \}$

$R \rightarrow +T \{ R_1.i = mknode('+', R.in, T.val) \}$
$R_1 \{ R.s = R_1.s \}$

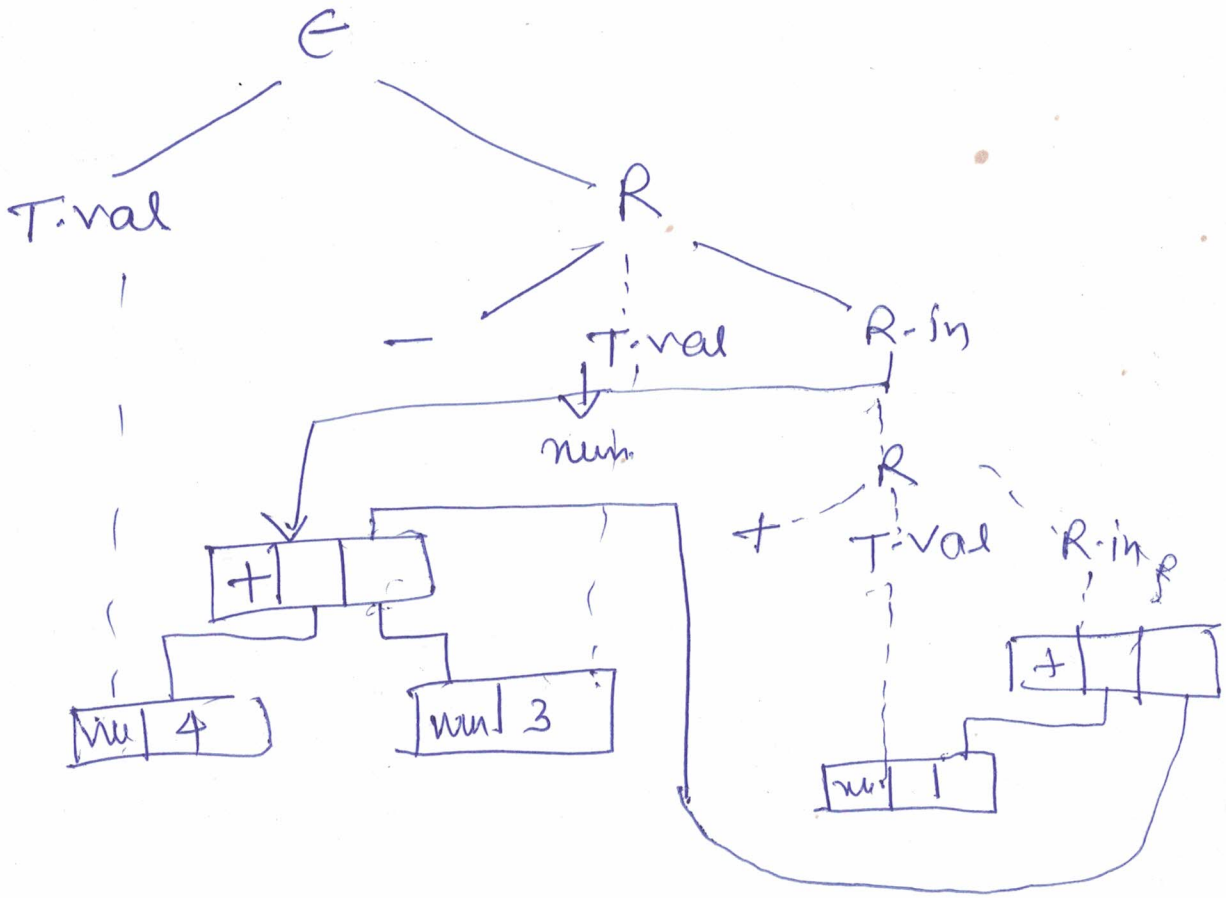
$R \rightarrow -T \{ R_1.i = mknode('-', R.in, T.val) \}$
$R_1 \{ R.s := R_1.s \}$

$R \rightarrow \in \{ R.s := R.i \}$

$T \rightarrow (E) \{ T.val := E.val \}$

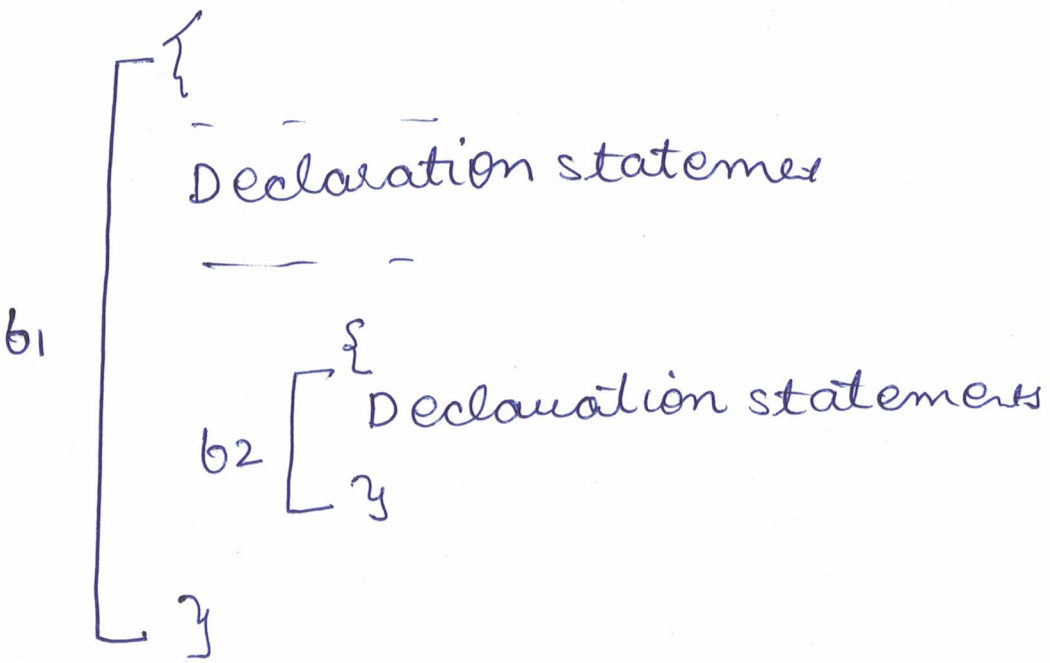
$T \rightarrow num \{ T.val = mkleaf(num, num.lexval) \}$

eg 4-3+1



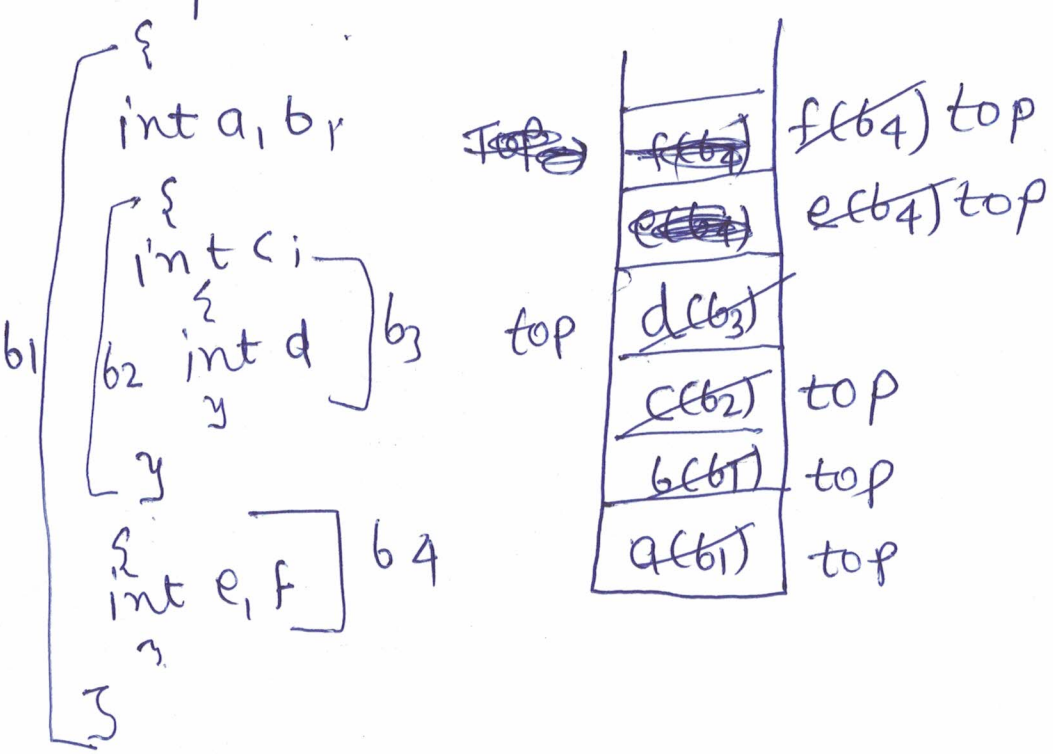
⇒ Accessing Local & Non-Local names in a Block structured language.

⇒ Block



⇒ static scope

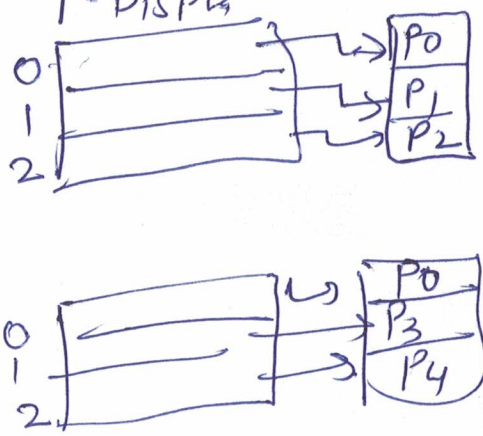
or Scope_obtain()

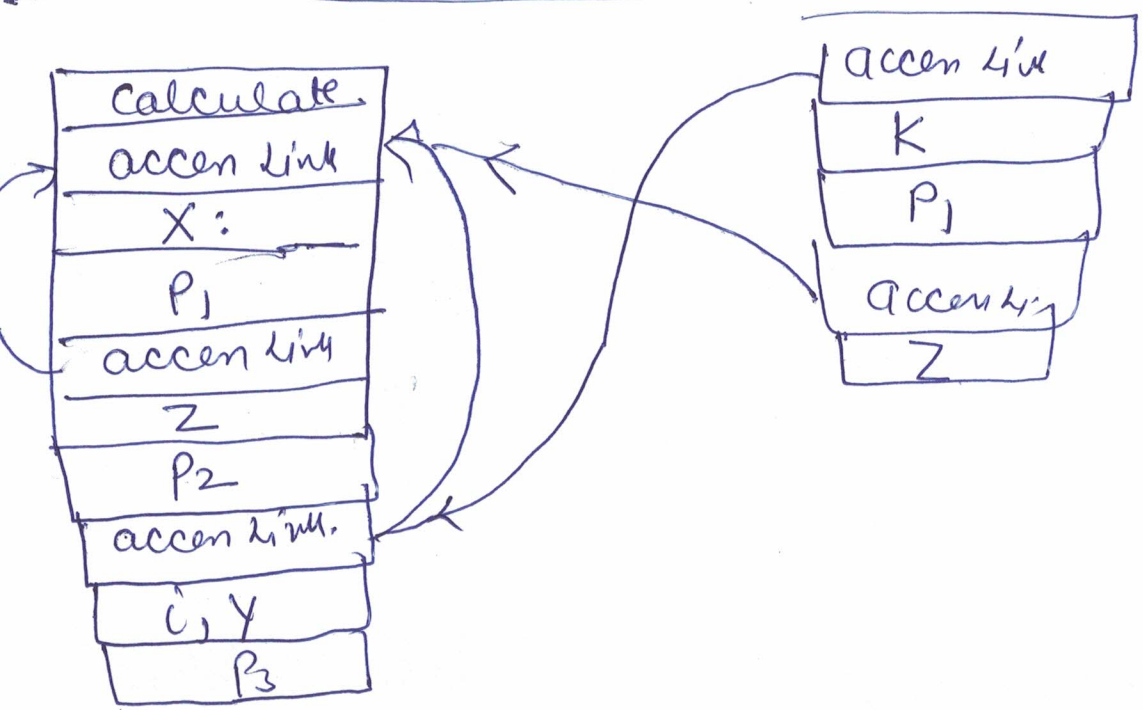


```
program Calculate;
   var x : int;
procedure P1;
  var z : int;
  begin
  x := 1
  end;
procedure P2 (i: int)
  var y : int
  procedure P3
 var k : int
  begin P1; end;
  begin
if (i <> 0) then P2 (i-1)
  else P3
  end
begin P2 (1); end;
```

displays DISPL

```
P0 ( )
{
P1 ( )
{
P2 ( )
{ }
}
P3 ( )
{
  P4 ( )
}
}
}
```





⇒ parameter passing

1) pass by value
2) pass by reference
3) pass by copy – restore (hybrid b/w call by value, call by ref.)
4) pass by name.

⇒ pass by copy – restore.

```
begin
  int a, b
  procedure by value result (int w)
  begin
    write ("w is", w)
    w = w * 3
    write ("a is now", a)
  end
  a = 10
  write ("a is", a)
by value result (a)
    write ("a's final value is", a)
    end.
```

O/P
a is 10
w is 10
a is now, 10
a's final value is 30

⇒ for Nested procedures

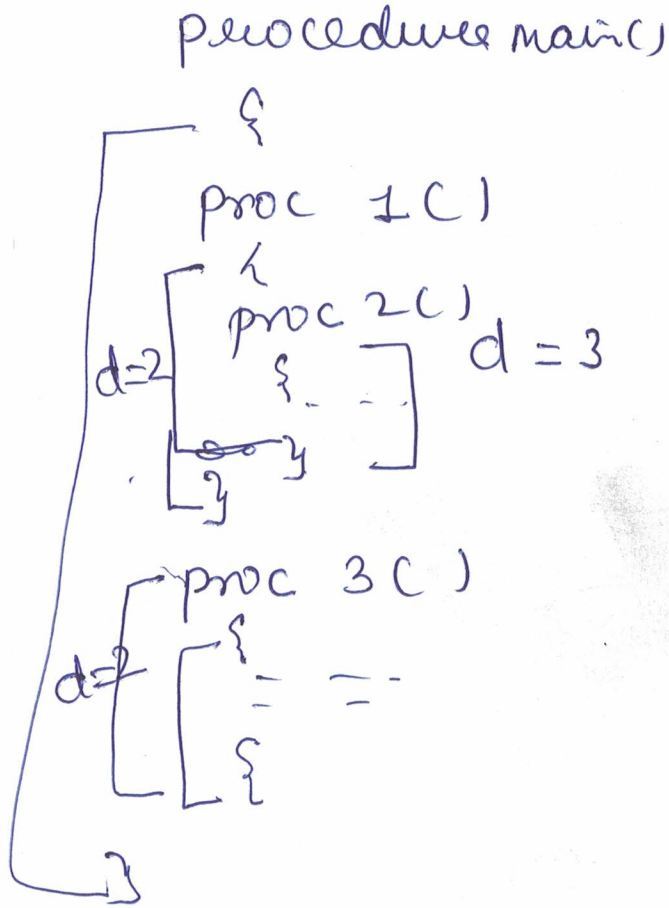
```
procedure 1()
{
- - - - -
procedure 2()
{
- - - - -
}
- - - - -
}
```

⇒ Nesting depth

main program = 1

new procedure = depth + 1

exit = depth - 1



⇒ By access link & displays

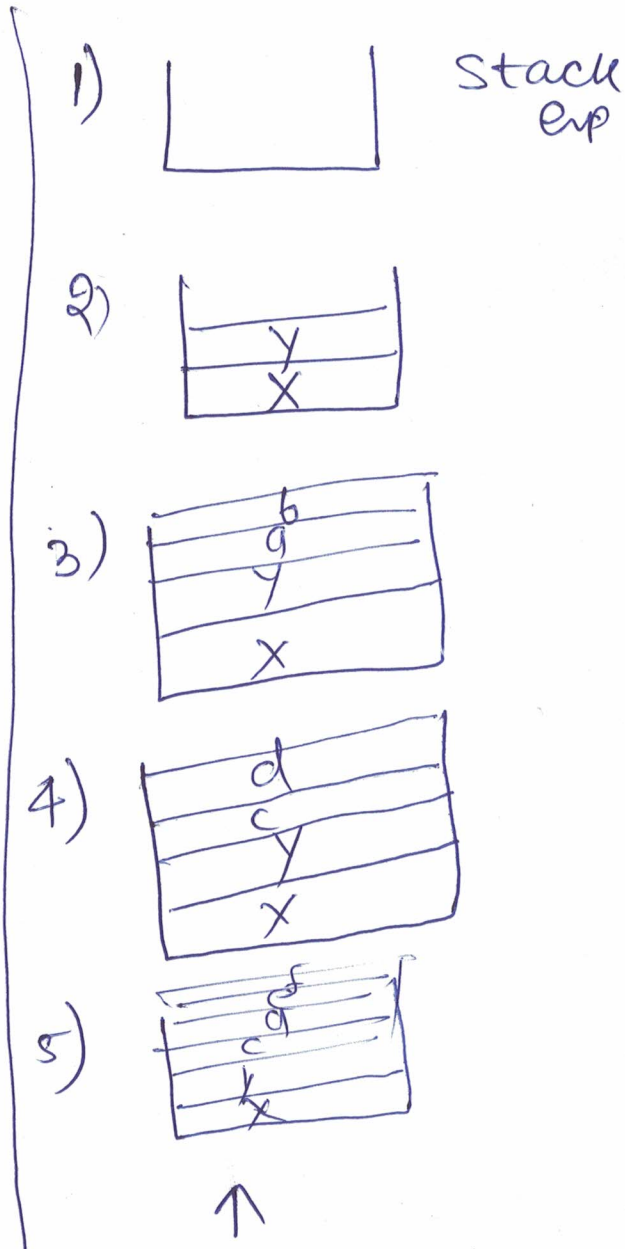
⇒ static allocation

f.eg

```
consumer()
  {
  char *Buf[50]
  int next;
  char c
  c = produce()
    -
    :
  }
char producer
  {
  char * Buffer[80]
  int next
  - -
  - -
  }
```

⇒ stack allocation

```
main()
  {
  int x, y
  - - -
procedure 1()
procedure 2()
  - - -
}
procedure 1()
  {
  int a, b;
  }
```




```
procedure 2()
  {
  in c, d
  - - -

procedure 3()
  - - -
  }
procedure 3()
  {
int e, f
  - -
  }
```