Chapter: 09

Syntax-Directed Translation

Grammar + semantic rules = SDT.
means translation of languages using context-free grammar or syntax analysis.

In practice whenever we go-for Compiler design, along with CFG, that grammar is associated with informal notations

so in computer design we are not going to use plan grammar, along with it we use some meaningful rules or semantic rules so that syntax analysis as well as many thing like, code generation, storing of some value in symbol table, Intermediate code generation, expression evaluation etc can be done parallel to syntax analysis.

There are two notations for associating semantic rules with productions.

1] Syntax-directed definition :- (SDD) [CFG together with attributes & rules] are high-level specifications for translations. They hides many implementation details & free the user from having to specify explicitly the order in which semantic rules are to be evaluated

2] Syntax-Directed Translation Scheme :- indicate the order in which semantic
(SDT)

rules are to be evaluated, so they allow some implementation details to be shown.

Why SDT is used?

SDT is used in compilers for type checking, generating intermediate code, expression evaluation, converting infix to postfix etc.

What are attributes?

Attribute can be anything like data type of variable, data stored in variable or any information which give more about non-terminal or terminal attributes are always written after . (dot).

ex A.attributes

SDT for evaluation of expression:-

Semantic Rule/Action

1] $E \rightarrow E + T$ { E.value = E.value + T.value }

/T { E.value = T.value }

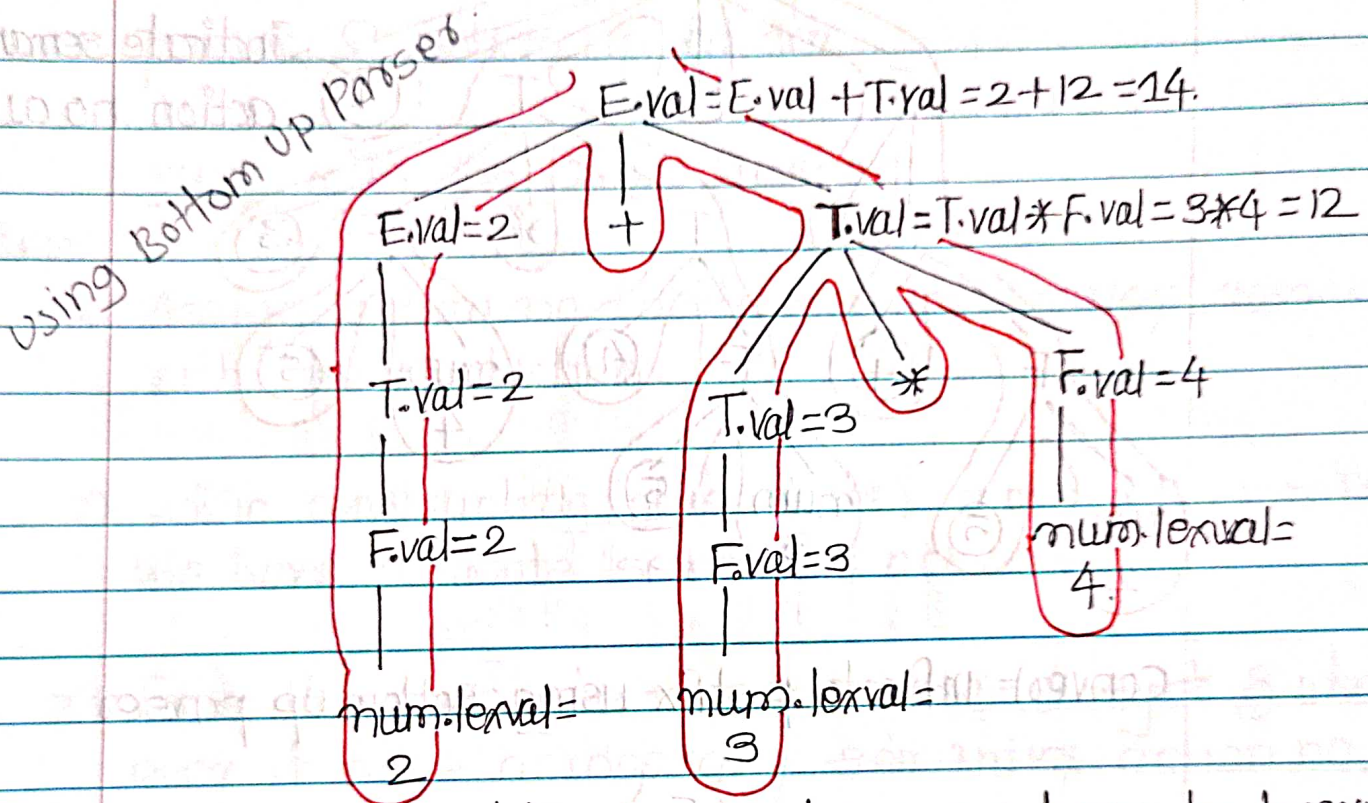
$T \rightarrow T * F$ { T.value = T.value * F.value }

/F { T.value = F.value }

$F \rightarrow num$ { F.value = num.lexval }

Note :- num.lexval → indicate value of num given by lexical analysis phase. (Actual value)

→ 2+3*4 & suppose i have to evaluate it.
If i carry out given SDT, i will get answer as 14



After constructing parse tree we have to travel it from (top ~~to bottom~~ down & left to right) top down left to right.

Whenever there is reduction we go to the production & carry out action (semantics rules)

2] SDT for converting infix to postfix expression.

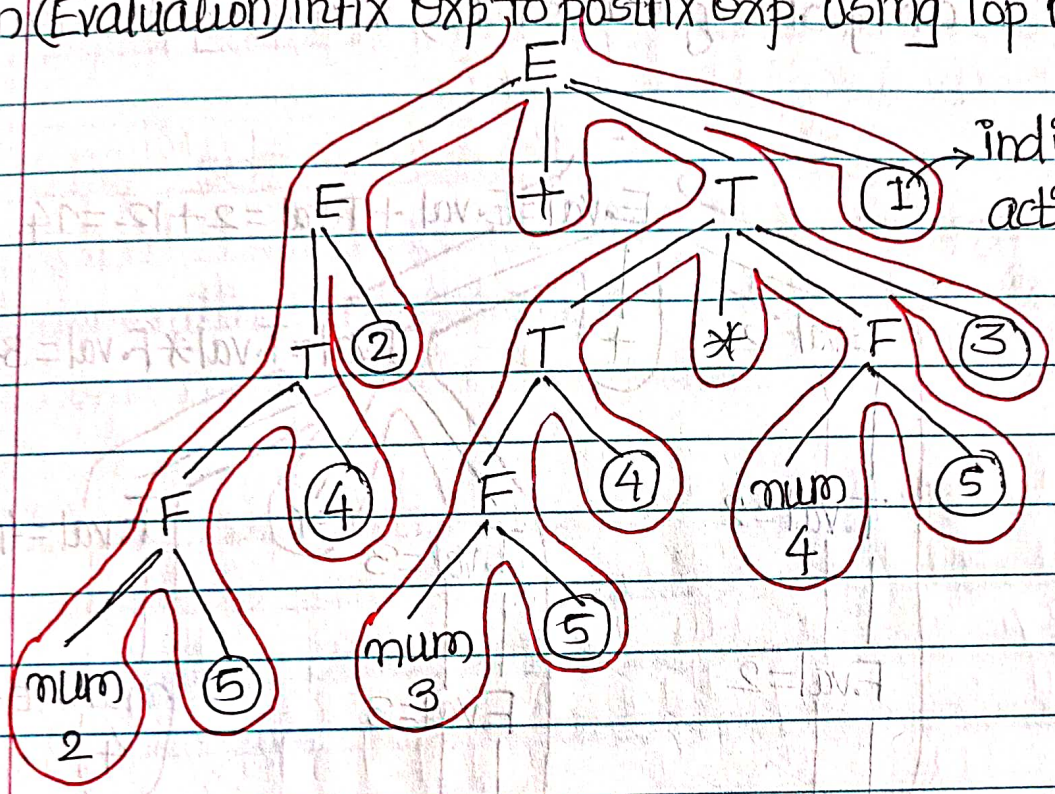
E → E+T {printf("+");}
  | T {}

T → T*F {printf("*");}
  | F {}

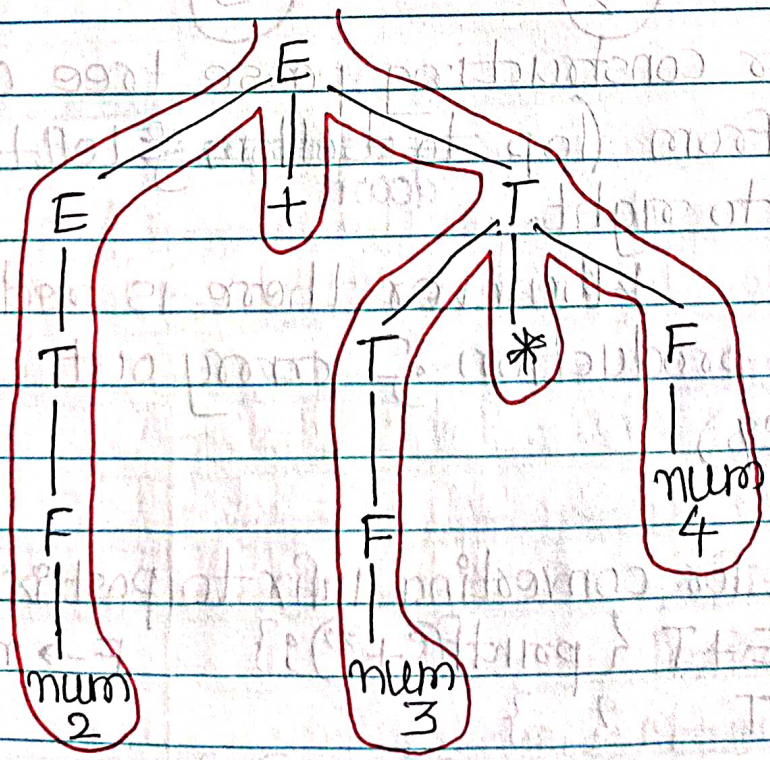
F → num {printf(num.lexval);}

→ 2+3*4. ⇒ 234*+

Conversion (Evaluation) infix exp to postfix exp. using Top Down parser



Convert infix to postfix using Bottom up parser.



There are two ways to carry out SDT.
One is during parsing or syntax analysis phase.
There are two types of parser one is top down parser & Bottom up parser.

SDT with Top Down Parser:-

steps

1] Assign action no to the semantic rules associated with the production.

2] While constructing parse tree, along with production we have to write semantics action no.

3] Follow the parse tree top down left to right & whenever it came across any semantics action no. just perform it.

SDT with Bottom Up parser:-

steps:

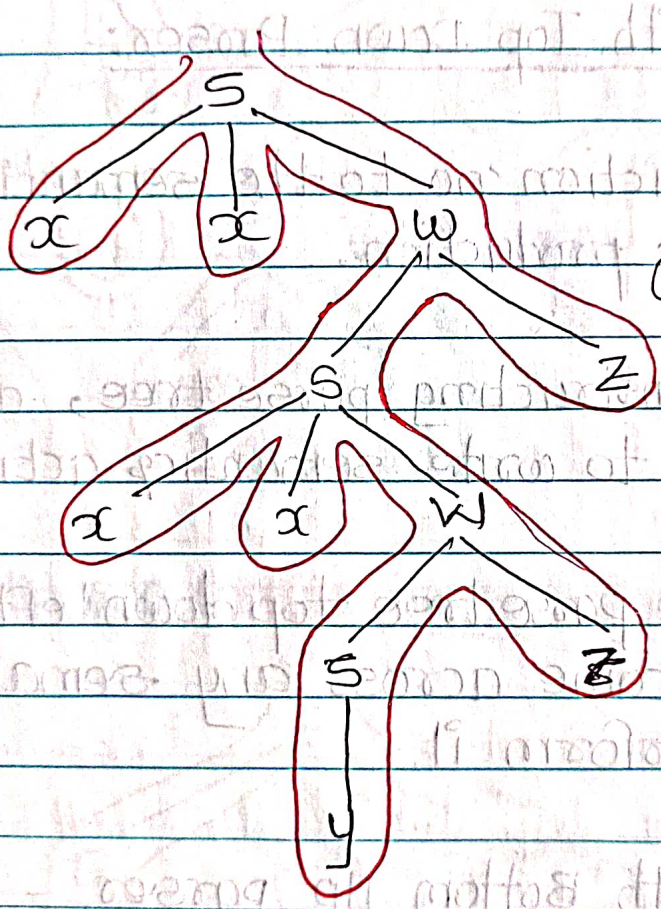
1] Construct parse tree for given input string.

2] follow the parse tree top down left to right & whenever there is reduction we go to the production & carry out the action.

3] S → xxxW {printf(1);}
~~W/////~~ /y {printf(2);}
W → Sz {printf(3);}

String → xxxxxyzz

Using BUP



O/P ⇒ 23131

4] E → E*T {E.val = E.val * T.val;}
/T {E.val = T.val;}
T → F-T {T.val = F.val - T.val;}
/F {T.val = F.val;}
F → 2 {F.val = 2;}
/4 {F.val = 4;}

W = 4-2-4*2

Understand - is having highest preeedence & it is right associative. * is left associative.

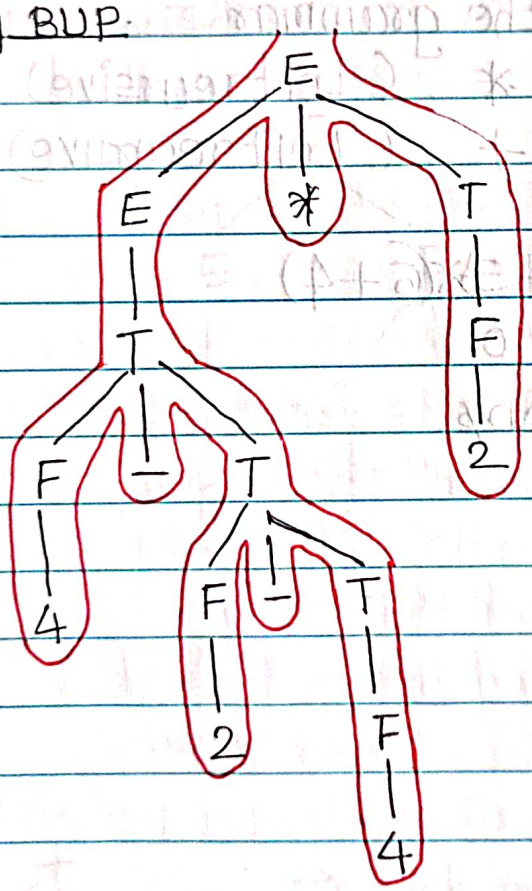
((4 - (2 - 4)) * 2)

((4 - (-2)) * 2)

6 * 2

= 12

Using BUP:



Every non-leaf show the reduction.
In above tree 10 non-leaf nodes are there. So 10 reductions are there.

5] $E \rightarrow E \# T$ { E.val = E.val * T.val ; }

/T { E.val = T.val }

$T \rightarrow T \& F$ { T.val = T.val + F.val }

/F { T.val = F.val }

$F \rightarrow num$ { F.val = num.lexval }

w = 2 #3 &5 #6 &4

By analyzing the grammar

is * (Left recursive) (low priority)

& is + (Left recursive) (high priority)

i.e

$2 * (3+5) * (6+4)$

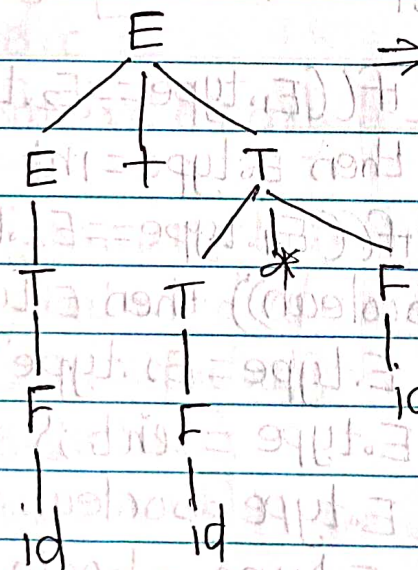
$2 * 8 * 10$

160 Ans

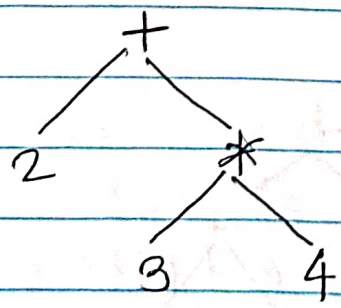
6] SDT to build syntax tree (Abstract parse tree)

E → E+T { E.nptr = mknode(E.nptr, '+', T.nptr); }
/T { E.nptr = T.nptr; }
T → T*F { T.nptr = mknode(T.nptr, '*', F.nptr); }
/F { T.nptr = F.nptr; }
F → id { F.nptr = mknode(null, idname, null); }

w = 2+3*4

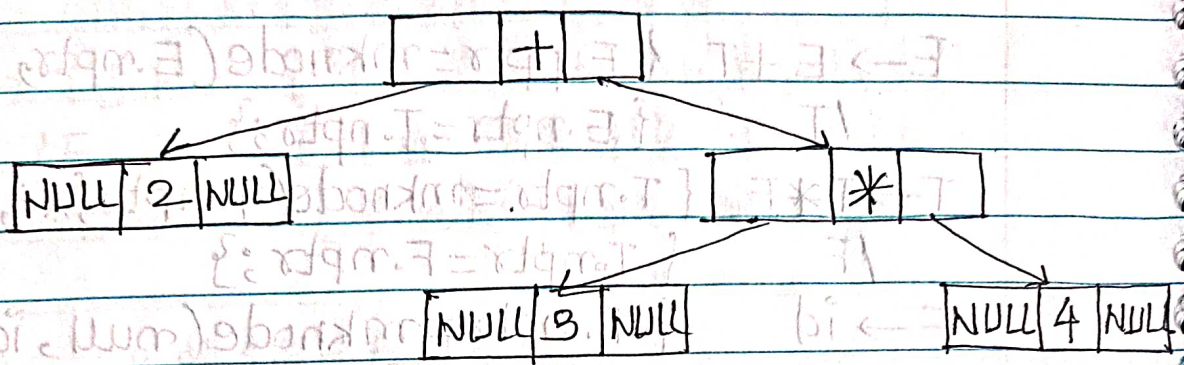


⇒ Such parse tree is called concrete parse tree it shows the derivation in detail manner.



⇒ Such parse tree is called Abstract parse tree. here we are representing the same but in abstract manner.

using BUP approach:-



7] SDT for type checking (considered for only int & boolean)

$E \rightarrow E_1 + E_2$ {if((E1.type==E2.type) && (E1.type=int)) then E.type=int else error;}

$/ E_1 == E_2$ {if((E1.type==E2.type) && E1.type=int/boolean)) then E.type=boolean else error;}

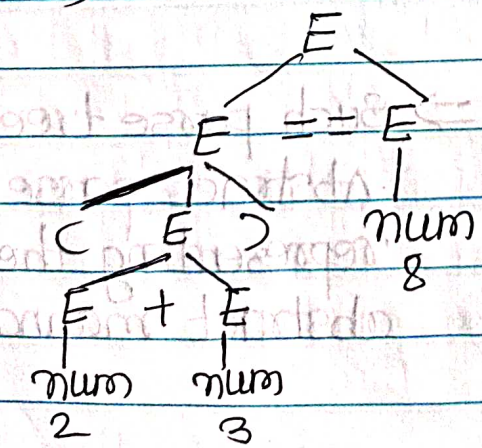
$/ (E_1)$ {E.type=E1.type}

/num {E.type = int;}

/True {E.type=boolean;}

/False {E.type = boolean;}

w = (2+3) == 8 (Ans- boolean)



8] SDT for count no. of 1 in binary number.

	No of 1's	No of 0 count
N → L	{ N.count = L.count	
L → LB	{L.count = L.count + B.count}	
| B	{L.count = B.count}	
B → 0	{B.count = 0}	B.c = 1
| 1	{B.count = 1}	B.c = 0

count No of bits

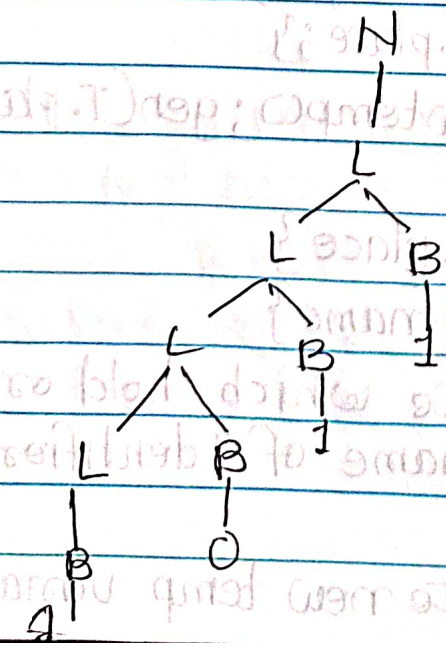
B.c = 1

B.c = 1

SDT to convert binary to decimal.

N → L	{ N.dval = L.dval }
L → LB	{ L.dval = L.dval * 2 + B.dval ; }
| B	{L.dval = B.dval}
B → 0	{B.dval = 0}
| 1	{B.dval = 1}

w = 1011



(included)

SDT to convert binary to decimal

ex 11.01

↳ Assume there is no decimal point

↓

01 → convert to decimal No i.e = 1

$$\frac{1}{2^{(\text{no of bits})}} = \frac{1}{2^2} = \frac{1}{4} = 0.25$$

N → $L_1 . L_2$ { N.dval = L_1.dval + L_2.dval/$2^{L_2.c}$ }

L → LB { L.c = L.c + B.c ; L.dval = L.dval + B.dval }

/B { L.c = B.c ; L.dval = B.dval }

B → 0 { B.c = 1 ; B.dval = 0

/1 { B.c = 1 , B.dval = 1 }

↓ ↓

count decimal value

9] SDT to generate three address code.

S → id = E { gen(id.name = E.place); }

E → E1 + T { E.place = newTemp(); gen(E.place = E1.place + T.place) }

/T { E.place = T.place ; }

T → T1 * F { T.place = newtemp(); gen(T.place = T1.place * F.place) ; }

/F { T.place = F.place }

F → id { F.place = id.name }

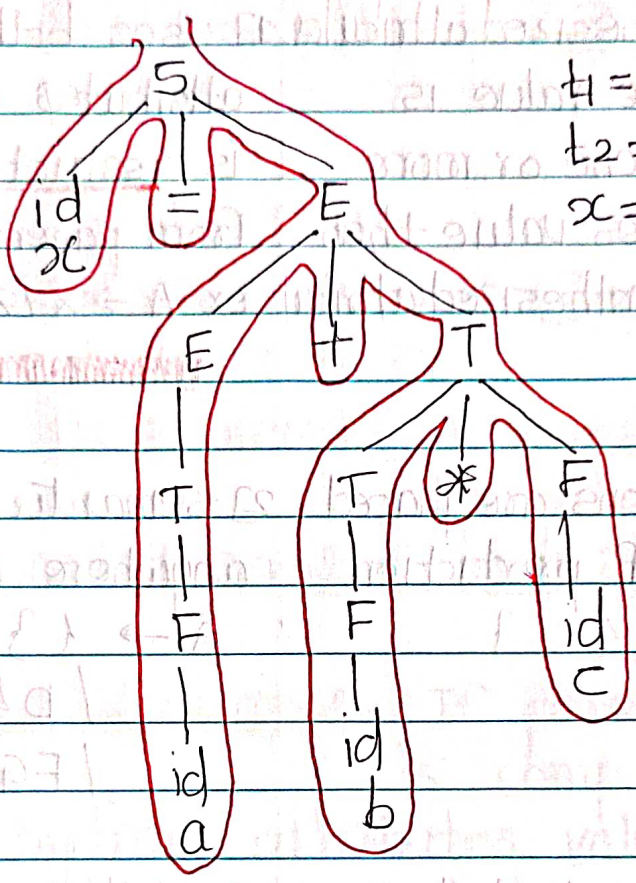
↳ attribute which hold or remember the name of Identifier

w = x = a + b * c ;

newtemp() - generate new temp variable.

gen( ) → which generate three address code point OR

BUP



$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

Attributes of SDD can be of two types

S-attributed SDT	L-attributed SDT
1] Uses only synthesized attributes. (if parent attribute value is computed from one or more children attributes value then it is called synthesized attributes)	1] Uses both inherited & synthesized attributes each inherited attribute is restricted to inherit either from parent or left siblings only. ex $A \to XYZ$ { Y.S = A.S ; Y.S = X.S ; Z.S = A.S + X.S + Y.S ; }
2] Semantics actions are placed at right end of production ex $A \to BC$ { }	2] Semantic actions are placed anywhere on RHS. $A \to$ { } BC / D { } E / FG { }
3] Attributes are evaluated during BUP, as the value of the parent nodes depend upon the values of the child nodes.	3] Attributes are evaluated by traversing parse tree depth first, left to right.
4] if SDT is S-attributed then it is also L-attributed	4] IF SDT is L-attributed then it may or may not be S-attributed SDT.

ex - P1 : $S \to MN$ { S.val = M.val + N.val }
P2 : $M \to PQ$ { M.val = P.val * Q.val ;
P.val = Q.val }

ex ① $A \to LM$ { L.i = f(A.i); M.i = f(L.S); A.S = f(M.S); }
$A \to QR$ { R.i = f(A.i); Q.i = f(R.i); A.S = f(Q.S); }
ⓐ S-attri ⓑ L attri ⓒ both ⓓ none

② $A \to BC$ { B.S = A.S; }
ⓐ S-attri ⓑ L attri ⓒ both ⓓ none

Annotated Parse Tree:- A parse tree showing attribute values at each node is called an Annotated Parse Tree.

Synthesized attributes:- If parent attribute value is computed from one or more childrens attributes values then it is called synthesized attributes.

ex -

$S \to ABC$ { S.val = A.val + B.val + C.val }

S attribute value is computed from childrens attribute value

synthesized attributes never take values from their parent nodes or any sibling nodes

Type of Synthesized attribute:-

① S-attributed SDT.

Inherited attributes :- Value of an attributes can be computated using attribute value of parent and/or siblings.

ex. $S \rightarrow ABC$ { A.val = S.val + B.val; S.val = A.val + B.val; B.val = S.val + A.val; }

Types of Inherited attributes :-

① **L - attribute**

Attributes may be of two types - Synthesized or Inherited.

1] Synthesized attributes - is an attribute of the non-terminal on the left hand side of a production.

In synthesized attribute, attribute can take value only from its children (variable in the RHS of the production)

for ex. let's say A → BC is a production of a grammar & A's attribute is dependent on B's attributes or C's attributes than it will be synthesized attribute

2] Inherited attributes - is an attribute of a nonterminal on the right-hand side of a production is called an Inherited attributes

The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production)

ex - A → BC is a production of a grammar & B's attribute is dependent on A's attributes or C's attributes than it will be inherited attributes