SDD + SDT

Outlines:

1. **SDD**
2. **Attributes of SDD: synthesized and inherited**
3. **Difference between synthesized and inherited**
4. **SDT**
5. **Attributes of SDT: L-attribute and S-attribute**
6. **Difference between L-attribute and S-attribute**
7. **Practice Problems on SDT**

Syntax Directed Definition

A CFG with attributes and rules is called a **syntax-directed definition (SDD)**. The properties are linked to the grammar symbols in an extended CFG (i.e. nodes of the parse tree). The rules are associated with grammar production.

SDD = grammar + semantic rules

Types of Attributes

There are two types of attributes:

1. Synthesized Attributes

2. Inherited Attributes

1. Synthesized Attributes: These are those attributes which derive their values from their children's nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children's nodes in parse tree.

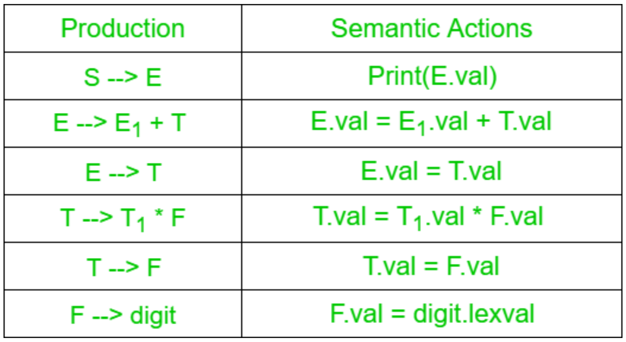
Example:

E --> E1 + T { E.val = E1.val + T.val}

In this, E.val derive its values from E1.val and T.val

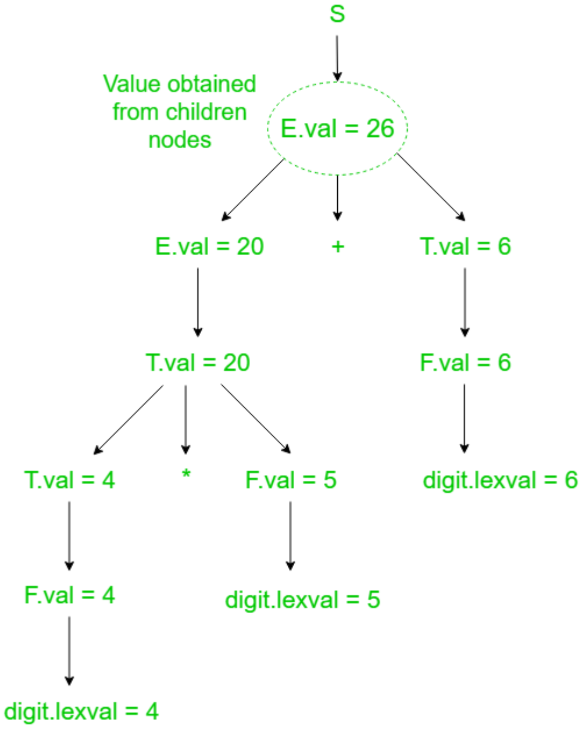
Example: Consider the following grammar

SDD + SDT



Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is :



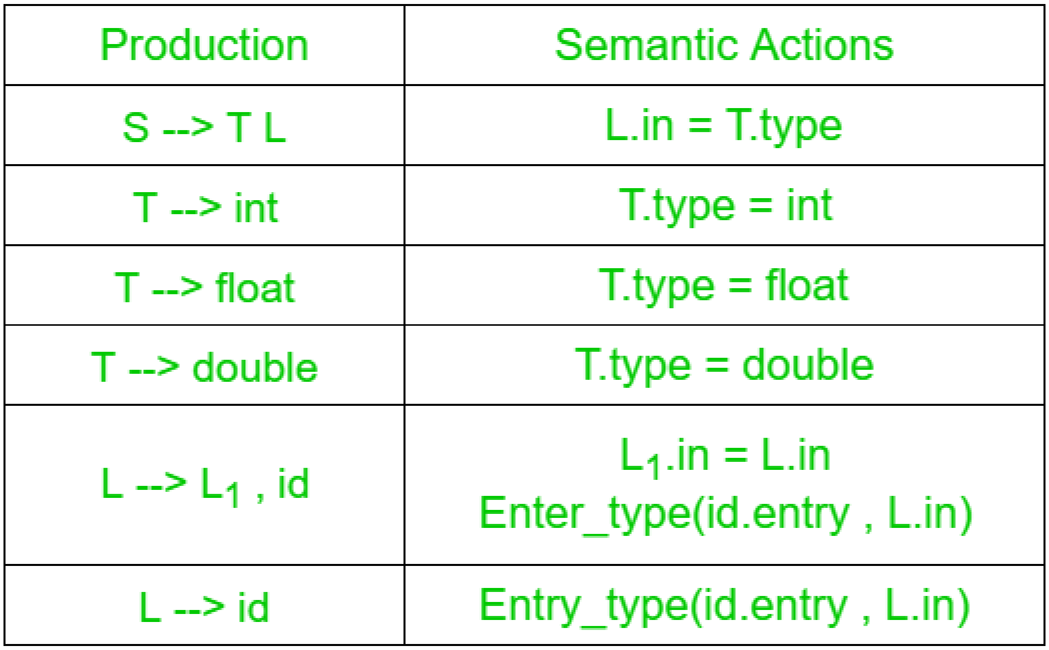
Annotated Parse Tree

2. Inherited Attributes: These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or sibling nodes.
Example:

A --> BCD { C.in = A.in, C.type = B.type }

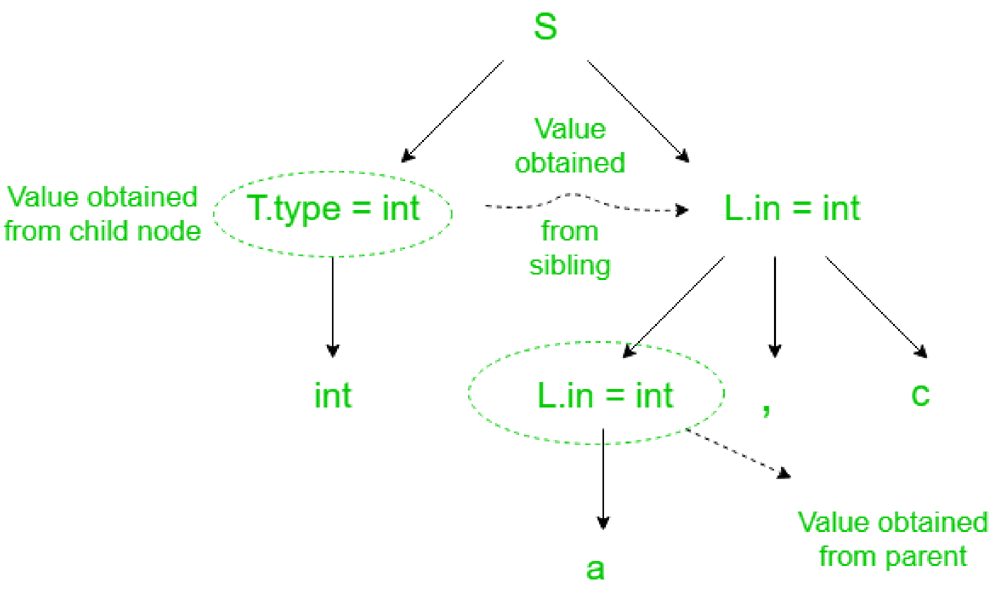
The SDD for the above grammar can be written as follow

SDD + SDT

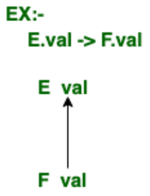
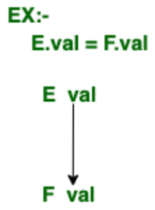


Production	Semantic Actions
S --> T L	L.in = T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is



S.NO	Synthesized Attributes	Inherited Attributes
1.	An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings' node.
2.	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.

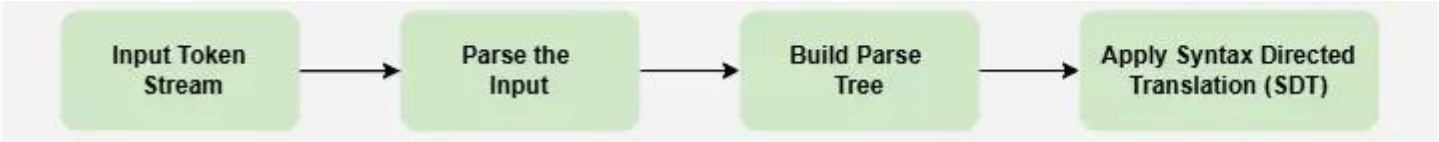
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
5.	Synthesized attributes can be contained by both the terminals or non-terminals.	Inherited attributes can't be contained by both, It is only contained by non-terminals.
6.	Synthesized attribute is used by both S-attributed SDT and L-attributed SDT.	Inherited attribute is used by only L-attributed SDT.
7.		

Syntax Directed Translation in Compiler Design

It refers to the translation of a string into an array of actions. This is done by adding an action to a rule of context-free grammar. It is a type of compiler interpretation.

SDT= grammar + semantic rules

SDT ensures a systematic and structured way of translating programs, allowing information to be processed bottom-up or top-down through the parse tree. This makes translation efficient and accurate, ensuring that every part of the input program is correctly transformed into its executable form.



Input Token Stream → Parse the Input → Build Parse Tree → Apply Syntax Directed Translation (SDT)

SDT relies on three key elements:

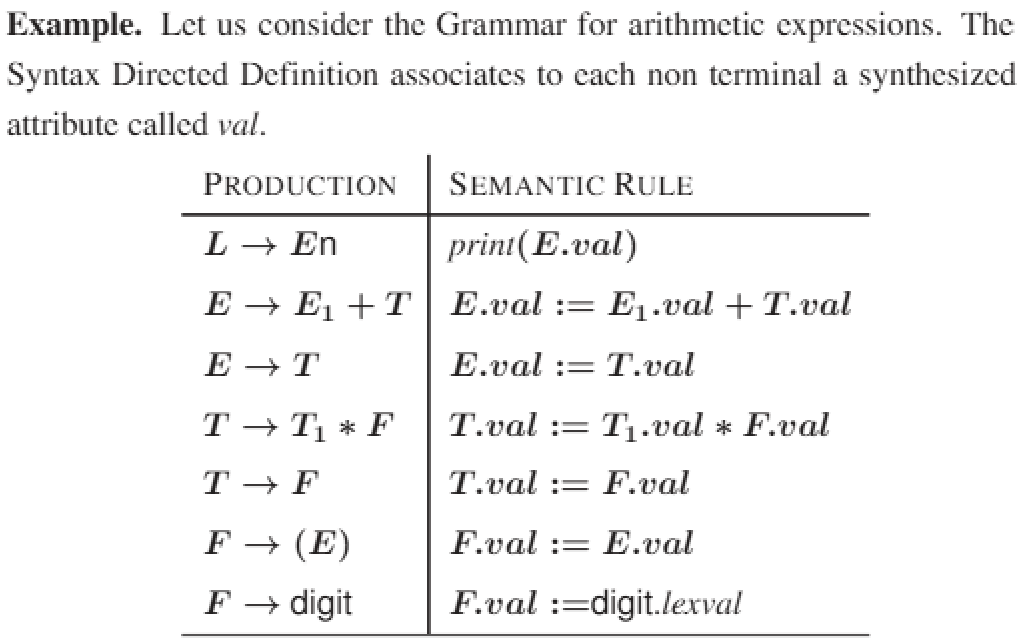
1. **Lexical values of nodes** (such as variable names or numbers).
2. **Constants** used in computations.
3. **Attributes** associated with non-terminals that store intermediate results.

S-attributed Translation

An SDD is S-attributed if the attributes of the node are synthesized attributes. To evaluate S-attributed SDD we can traverse the nodes of the parse tree in any bottom-up order.

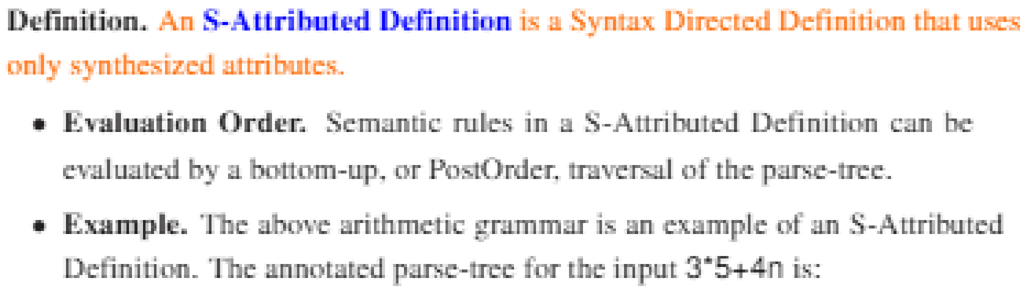
SDD + SDT

In S-attributed SDD the attributes are evaluated by applying post-order traversal. In postorder traversal, node N is evaluated when the traverser traverses node N for the last time.



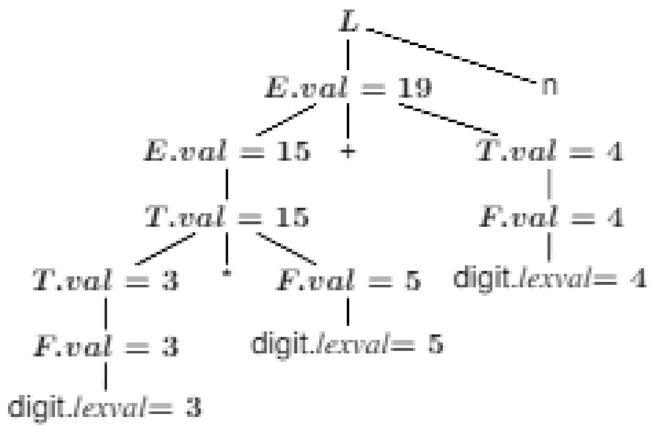
Example. Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

PRODUCTION	SEMANTIC RULE
$L \rightarrow E\text{n}$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \text{digit}$	$F.val :=\text{digit}.lexval$



Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

- **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.
- **Example.** The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:

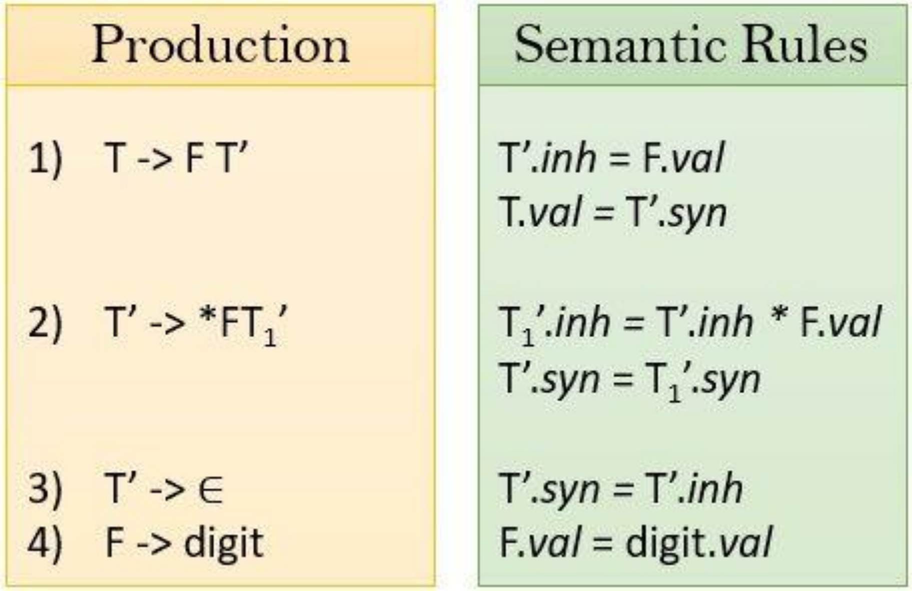


L-attributed Translation

An SDD is L-attributed if the attributes of nodes are either synthesized or inherited. Here we can traverse the parse tree strictly from left to right. This is because, in 'L-attributed translation', L signifies left-to-right traversing.

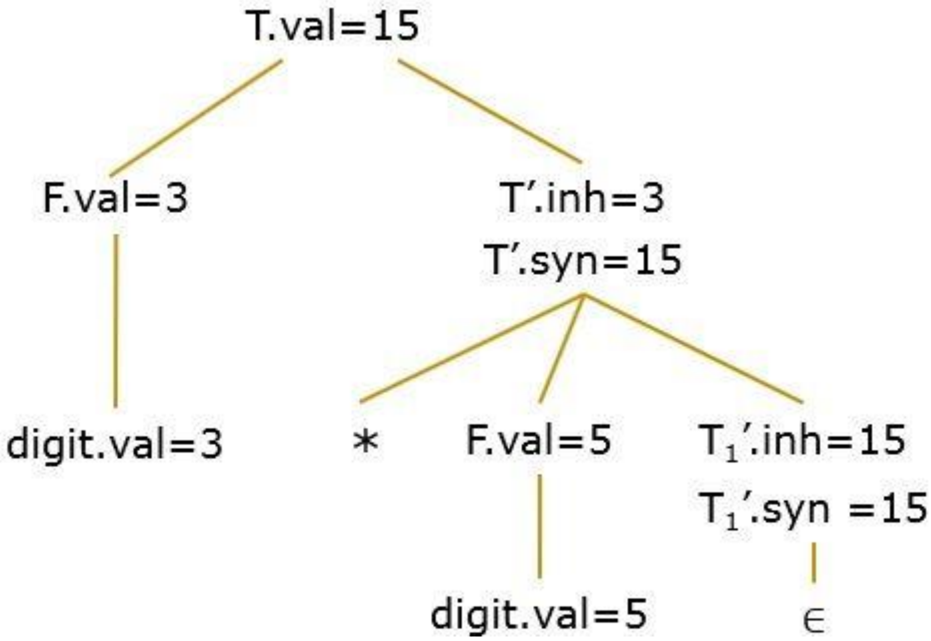
The SDD in the figure below is L-attributed:

SDD + SDT



Production	Semantic Rules
1) T -> F T’	T’.*inh* = F.*val* T.*val* = T’.*syn*
2) T’ -> *FT_1’	T_1’.*inh* = T’.*inh* * F.*val* T’.*syn* = T_1’.*syn*
3) T’ -> ∈	T’.*syn* = T’.*inh*
4) F -> digit	F.*val* = digit.*val*

The Annotated parse tree for the L-attributed SDD above is as follow:



To evaluate the L-attributed SDD, start evaluating the following order:

1. The value of the inherited attribute.
2. Then the value of synthesizing attributes.

SDD + SDT

S- attributed SDD	L- attributed SDD

A SDD that uses only synthesized attribute is called S-attributed SDD.
 	
Ex:
A → BCD
A.i = B.i
A.i = C.i
A.i = D.i
	A SDD that uses both synthesized and inherited attribute is called l-attributed SDD.
	L-attributed SDD but each inherited attribute is restricted to inherit from parent or left siblings only.
	A → BCD
	C.i = A.i
	C.i = B.i
	C.i = D.i

Semantic actions are always placed at right end of the production.	Semantic action are placed anywhere or RHS of the production.
Attributes are evaluated with bottom to parent	Attributes are evaluated by traversing parse tree depth first, left to right.

4. SDD v/s SDT Scheme

Syntax Directed Definition	Syntax Directed Translation
It is a context-free grammar where attributes and rules are combined and associated with grammar symbols and productions, respectively.	It refers to the translation of a string into an array of actions. This is done by adding an action to a rule of context-free grammar. It is a type of compiler interpretation.
Attribute Grammar	Translation Schemes
SDD: Specifies the values of attributes by associating semantic rules with the productions.	SDT: Embeds program fragments (also called semantic actions) within production bodies.
E → E + T { E.val := E1.val + T.val }	**E → E + T { print('+'); }**
Always written at the end of the body of production.	The position of the action defines the order in which the action is executed (in the middle of production or at the end).
More Readable	More Efficient
Used to specify the non-terminals.	Used to implement S-Attributed SDD and L-Attributed SDD.
Specifies what calculation is to be done at each production.	Specifies what calculation is to be done at each production and at what time they must be done.
Left to right evaluation.	Left to right evaluation.
Used to know the value of non-terminals.	Used to generate Intermediate Code.

SDD + SDT

Problems on SDT:

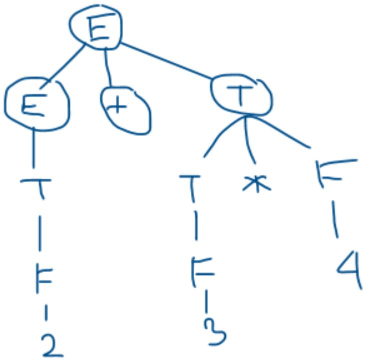
Problem -1 : The following **context-free grammar (CFG)** defines an arithmetic expression with **addition (+) and multiplication (*)**:

```
E -> E + T     { E.val = E.val + T.val }
E -> T         { E.val = T.val }
T -> T * F     { T.val = T.val * F.val }
T -> F         { T.val = F.val }
F -> INTLIT    { F.val = INTLIT.lexval }
```

Let's evaluate the expression: **S = 2 + 3 * 4**

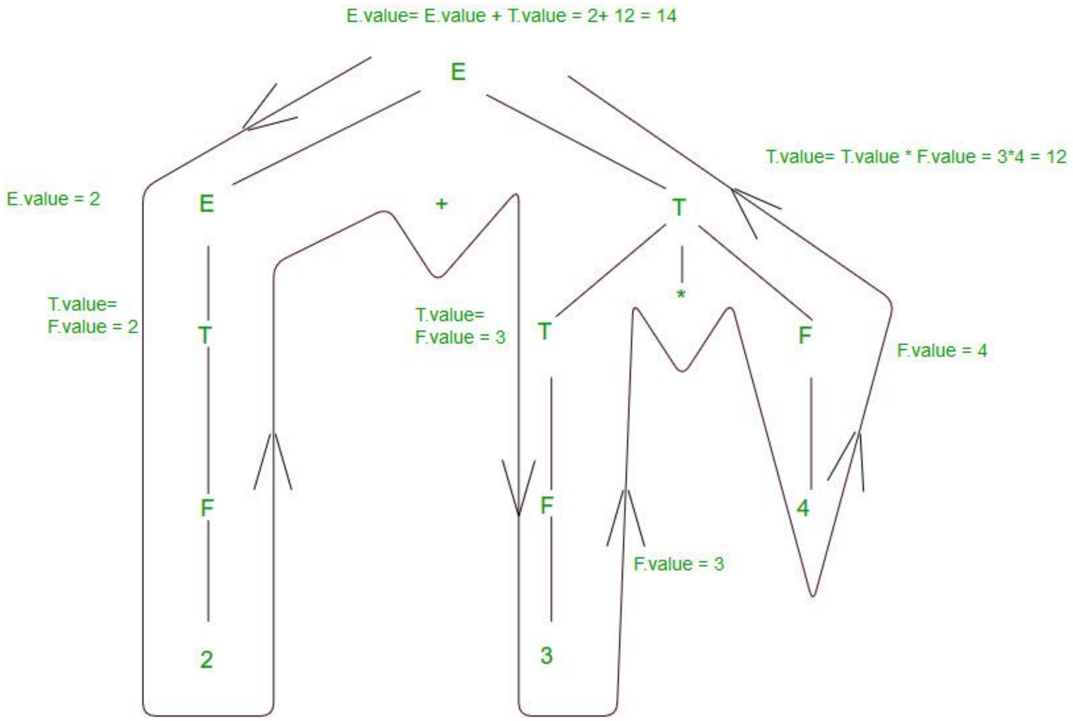
Step 1: Build the Parse Tree

The **parse tree** for 2 + 3 * 4 is structured like this:



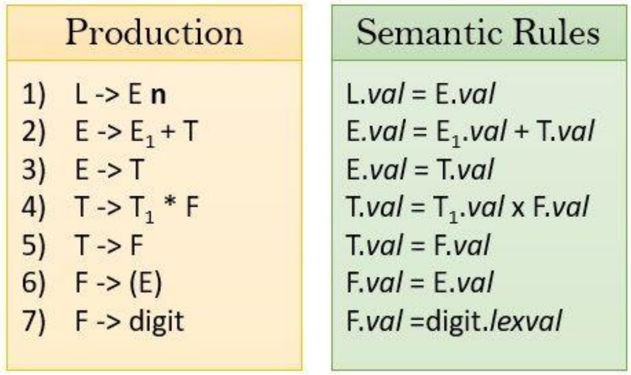
Step 2: Apply Translation Rules (Bottom-Up Evaluation)

We evaluate the expression step by step in a bottom-up manner (from leaves to root)



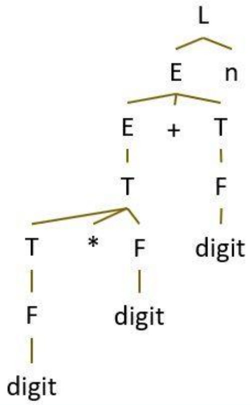
SDD + SDT

Problem -2 :. Consider the SDD provided in the figure below. Here we can see the production rules of grammar along with the semantic actions. And the input string provided by the lexical analyzer is 3 * 5 + 4 n.



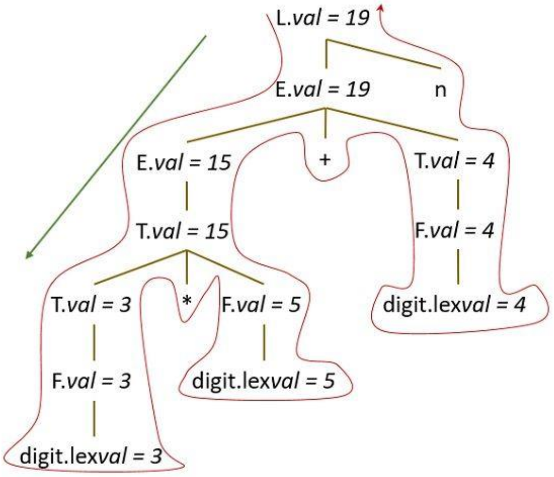
Production	Semantic Rules
1) L -> E **n**	L.*val* = E.*val*
2) E -> E_1 + T	E.*val* = E_1.*val* + T.*val*
3) E -> T	E.*val* = T.*val*
4) T -> T_1 * F	T.*val* = T_1.*val* x F.*val*
5) T -> F	T.*val* = F.*val*
6) F -> (E)	F.*val* = E.*val*
7) F -> digit	F.*val* =digit.*lexval*

To perform SDT the first thing we must do is to create a parse tree with the provided production and lexical input.



Parse Tree for Input String 3 * 5 + 4 n

The next step is to create an annotated parse tree. So, we will traverse the parse tree from top to bottom. Then add the semantic rule to evaluate the attribute of each node. Thus, you can evaluate the output for the given input string as you can see in the image below.



Annotated Parse Tree for Input String 3 * 5 + 4 n

SDD + SDT

Problem -3 : Consider the following grammar and the semantic actions to support the inherited type declaration attributes. Let *X*1, X2, X3, *X*4, *X*5, and *X*6 be the placeholders for the non-terminals D, T, L or L1 in the following table:

Production Rule	**Semantic Action**
D → T L	**_X_1.type = _X_2.type**
T → int	**T.type = int**
T → float	**T.type = float**
L → Ll , id	**_X_3.type = _X_4.type addType(id.entry, _X_5.type)**
L → id	**addType(id.entry, _X_6.type)**

Which one of the following are the appropriate choices for *X*1, *X*2, *X*3 and *X*4?

(A) X1=L,X2=T,X3=L1,X4=L

(B) X1=T,X2=L,X3=L1,X4=T

(C) X1=L,X2=L,X3=L1,X4=T

(D) X1=T,X2=L,X3=T,X4=L1

Problem - 4 :. Consider the following Syntax Directed Translation Scheme (SDTS), with non-terminals {S, A} and terminals {a,b}.

S→aA{print1}

S→a{print2}

S→Sb{print3}

S→aA{print1}S→a{print2}S→Sb{print3}

Using the above SDTS, the output printed by a bottom-up parser, for the input **aab** is:

- (A) 1 3 2
- (B) 2 2 3
- (C) 2 3 1
- (D) syntax error

SDD + SDT

Problem - 5 :

production	semantic rules
E->E *T	E.val=E.val*T.val
E->T	E.val=T.val
T->T&F	T.val=________
T->F	T.val=F.val
F->digit	F.val=digit.val

if the expression 8#12&4#16&12#4&2 is evaluated to 512 then which of the following is correct replacement for blank.

1. T.val = T.val * F.val

2. T.val = T.val + F.val

3. T.val = T.val - F.val

4. none

Problem - 6 :

grammar	rules
S->S#A	S.val = S.val * A.val
A->A&B	A.val = A.val + B.val
B->id	B.val = id.lval

given string w = 5 # 3 & 4. Find output.