⊛ Syntax - Directed Definition (SDD) :-

SDD = CFG + Semantic Rules

→ A SDD is a context free grammar together with semantic Rules.

→ Attributes are associated with grammar symbols and semantic rules are associated with productions.

→ if 'x' is a symbol and 'a' is one of its attributes then x.a denotes value at node x.

→ Attributes may be number, strings, references, datatype, etc.

Production	Semantic Rule
$E \to E + T$	$E.Val = E.Val + T.Val$
$E \to T$	$E.Val = T.Val$

(E.Val: E → attribute; grammar s/b.)

⇒ Types of Attributes :-

① Synthesized Attribute :- if a node takes value from its children then it is Synthesized attribute.

eg:- $A \to BCD$, A be a parent node, B, C, D are children nodes

$A.S = B.S$
$A.S = C.S$
$A.S = D.S$

} Parents node A taking value from its children B, C, D.

② Inherited Attribute :- if a node takes value from its parent or siblings.

eg:- $A \to BCD$

$C.i = A.i$ → Parent node
$C.i = B.i$ → Siblings
$C.i = D.i$ → "

⊛ Types of Syntax Directed Definition (SDD) :-

1. S-Attributed SDD (or) S-Attributed Definitions (or) S-Attributed Grammar.
2. L-Attributed SDD (or) L-Attributed Definitions (or) L-attributed grammar.

S-Attributed SDD

1. A SDD that uses only Synthesized attributes is called as S-attributed SDD.

Ex:- $A \rightarrow BCD$
$A.S = B.S$
$A.S = C.S$
$A.S = D.S$

2. Semantic Action are always placed at right End of the production it is also called as "postfix SDD"
3. Attributes are Evaluated with Bottom-up parser.

L-Attributed SDD

1. A SDD that uses both Synthesized & Inherited Attributes is called as L-Attributed SDD but each inherited attribute is restricted to inherits from parent or left Sbling only.

Ex:- $A \rightarrow XYZ$ $\{Y.S = A.S,\ Y.S = X.S,\ Y.S = Z.S\}$

2. Semantic actions are placed anywhere on R.H.S.
3. Attribute are Evaluated by traversing parse Depth first, left to right order.

⊛ Syntax Directed Translation (SDT)

SDT = Grammar + Semantic Rule

⇓

informal notations.

* In SDT, every non-terminals can get '0' zero OR more attributes. (depending on the type of attributes.)

⊛ In semantic rule, attribute is value of attributes can hold → String, No., Memory locations. } it is represented as (Val.)

Example:-

Production	Semantic Rule (Action)
$E \rightarrow E+T$	$E.Val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T*F$	$T.val = T.val * F.val$
$T \rightarrow F$	$T.val = f.val$
$F \rightarrow Num$	$f.val = num.lix\ val.$

⇓

attribute return by LA.

⊛ Syntax Directed Translation Scheme:-

- ⊛ The Syntax Directed Translation scheme is a CFG.
- ⊛ It is used to evaluate the order of semantic rules
- ⊛ In translation scheme the semantic rules are embedded within the right side of the production.
- ⊛ The position at which an action is to be executed is shown by enclosed b/w braces. it is written within the right side of the production.

Example:

Aug Augumented grammar

Production	Semantic Rules → Actions
$S \to E \$$	{ Print E.Val } → Print
$E \to E+E$	{ E.Val = E.Val + E.Val }
$E \to E*E$	{ E.Val = E.Val * E.Val }
$E \to (E)$	{ E.Val = E.Val }
$E \to I$	{ E.Val = I.val }
$I \to I$ digit	{ I.Val = 10*I.Val + LexVal }
$I \to$ digit	{ I.Val = LexVal }

Implementation of Syntax directed translation:-

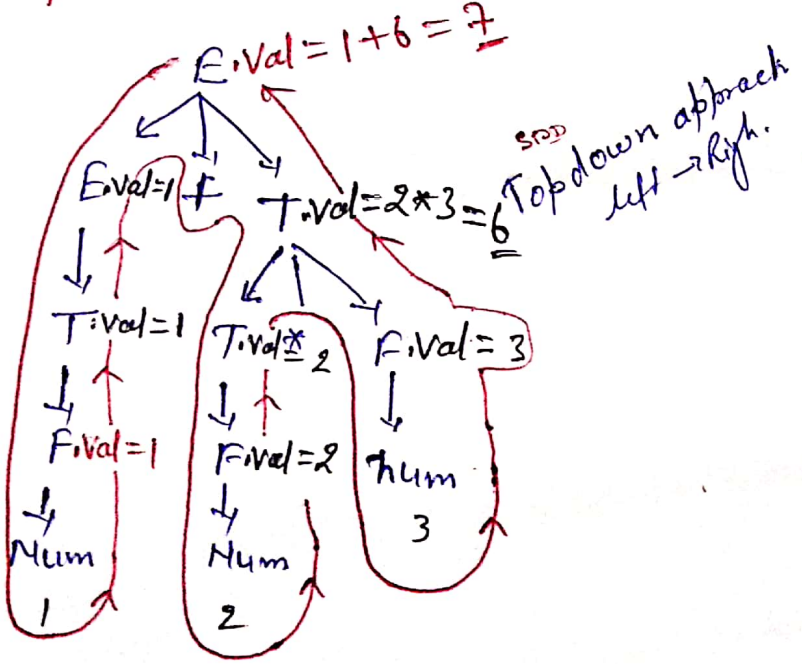
- SDT is implemented by constructing a parse tree and performing the actions in a left to right depth first order.
- SDT is implementing by parse the input and a parse tree as a result.

Example:-

	Action
$E \rightarrow E+T$	$\{E.val = E.val + T.val\}$
$E \rightarrow T$	$\{E.val = T.val\}$
$T \rightarrow T*F$	$\{T.val = T.val * F.val\}$
$T \rightarrow F$	$\{T.val = F.val\}$
$F \rightarrow num$	$\{F.val = num.lexval\}$

for this

Eg: $1+2*3$

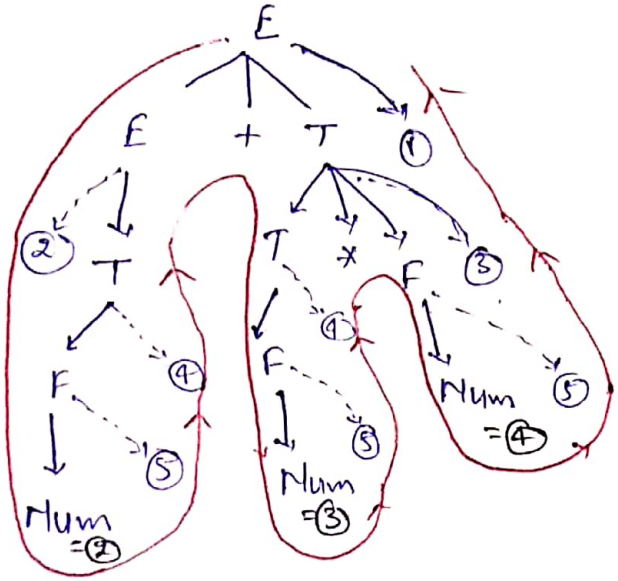


SDD
Top down approach
left → Right.

✱ Top Down Approach / Top Down Translation Approach..

Production	Semantic Rule.
$E \to E+T$	$\{E.val = \{E.val + T.val\}$ — (1)
$E \to T$	$\{E.val = \{T.val\}$ — (2)
$T \to T * f$	$\{T.val = \{T.val * F.val\}$ — (3)
$T \to f$	$\{T.val = \{f.val\}$ — (4)
$F \to Num$	$\{f.val = \{Num.lexval\}$ — (5)

i/p String - 2+3*4 (Top down) = 14

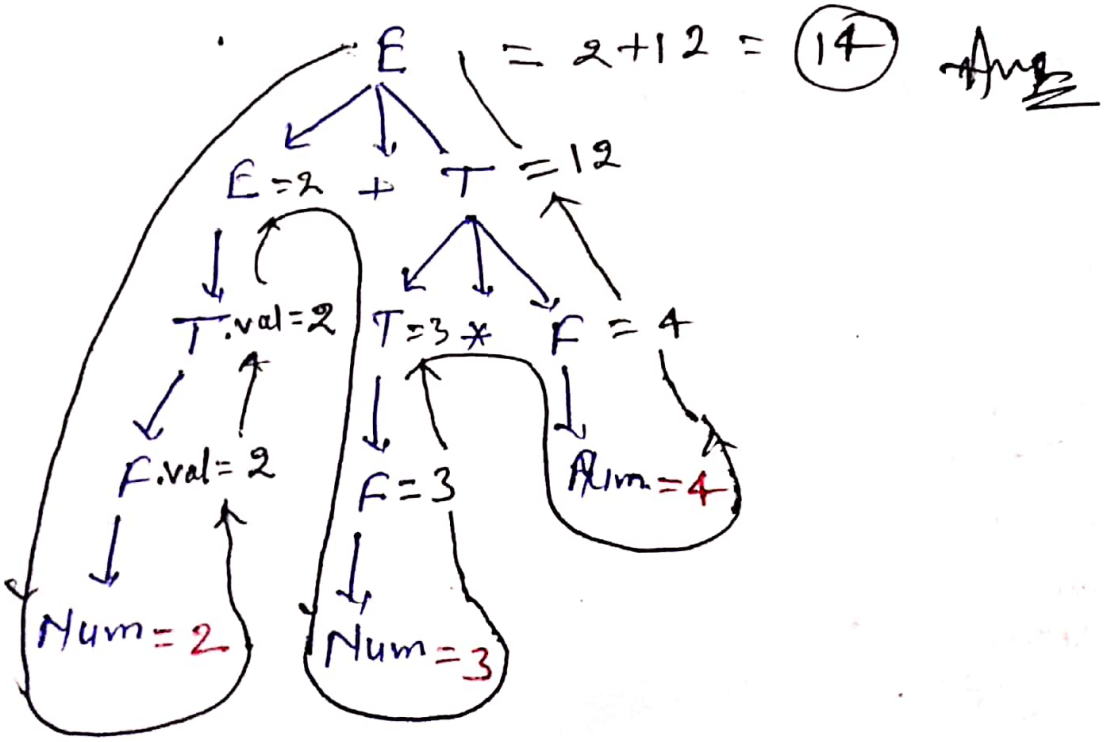


1. $f.Val = 2$
2. $T.val = f.val = 2$
3. $E.val = T.val = 2$
4. $F.val = T.val * f.Val$ Num.lexval = 3
5. $T.val = f.val = 3$
6. $F.val = Num.lexval = 4$
7. $T.val = T.val * f.val = 3*4 = 12$
8. $E.val = E.val + T.val = 2+12 = 14$ Ans

⊛ Bottom Up Approach :-

$E \rightarrow E + T$	(1) $\{ E.val = \{ E.val + T.val \}$
$E \rightarrow T$	(2) $\{ E.val = \{ T.val \}$
$T \rightarrow T * F$	(3) $\{ T.val = \{ T.val * f.val \}$
$T \rightarrow F$	(4) $\{ T.val = \{ f.val \}$
$F \rightarrow Num$	(5) $\{ f.val = \{ Num.val \}$

I/p string 2+3*4 = 14



⊛ Intermediate Code Generation :-

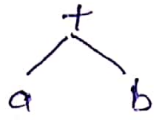
① Post fix Notation
② Syntax Tree
③ Three Address Code.
   → Quadruple
   → Triple

⊛ Postfix Notation :-

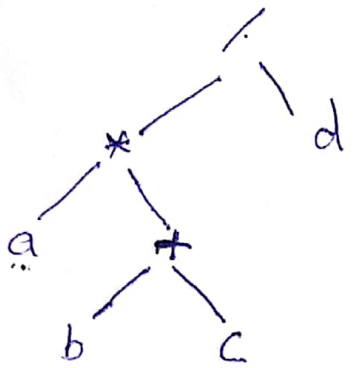
eg	infix	Prefix	Postfix
	a+b	+ab	ab+
	(a+b)*c		ab+c*

⊛ Syntax Tree

a+b ⇒



a*(b+c)/d ⇒

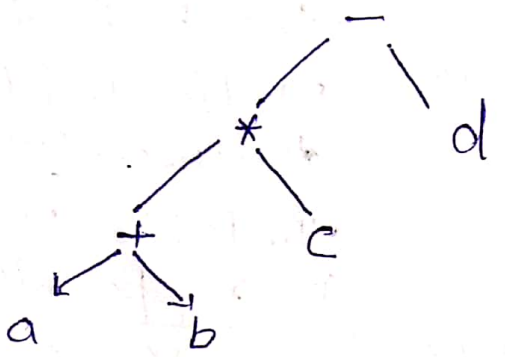


Syntax tree

infix:- LpRoot Right
Prefix: Root Lpt Rgt

infix:- a*b+c/d
Prefix:- /*a+bcd
Postfix:- abc+*d/

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



Prefix :- * + a b c d

Postfix :- a b + c * d -

⊛ Three Address Code:-

Types of Three Address Statement

① logic and Binary operator $x := y \ op \ z$

② Unary operator $x := op \ y$

③ Copy Statement $x := y$

④ Procedure Call parameter y, parameter x

⑤ Indix Assignment $x = y[i]$

$x[i] = y$

⑥ Conditional jump if x relop y goto level

⑦ Address and pointer $x = \&y$

$x = *y$

⊛ Implementation of Three Address Code:-

There are two method

① Quadruple

② Triple.

⇒ Quadruple

⇒ Argument1, Argument2, operator, result

(arg1, arg2, op, result)

eg:- $a = b * -c + b * -c$

first convert in 3 address form.

$t_1 = -c$ —— (0)

$t_2 = b * t_1$ —— (1)

$t_3 = -c$ —— (2)

$t_4 = b * t_3$ —— (3)

$t_5 = t_2 + t_4$ ⇒ $a = t_5$ —— (4) —— (5)

Quadruple:-

	operator (op)	argument1 (arg1)	Argument2 (arg2)	Result
(0)	Unary(-)	c		t_1
(1)	*	b	t_1	t_2
(2)	Unary(-)	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_3	t_4	t_5
(5)	Assign	~~t_5~~		a

Triple:- (operator, argument 1, argument 2) => (op, arg1, arg2)

	operator (op)	Argument1 (arg1)	Argument2 (arg2)
(0)	Unary(-)	c	
(1)	*	b	(0)
(2)	Unary(-)	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	Assign	a	(5)

⊛ DAG Representation : (Directed Acyclic Graph)

- DAG stand for Directed Acyclic Graph
- Syntax tree and DAG both are graphical representation Syntax tree does not find the common sub expressions where as DAG can.
- Another usage of DAG is the application of optimization technique in the basic block.
- To apply optimization technique on basic block, DAG is constructed three address code which is the output of an intermediate code generation.

⇒ Algorithm for Construction of DAG :-

Input :- It contains a basic block

Output :- it contains the following information

→ Each node contains a label. for leaves, the label is an identifier.

→ Each node contains a list of attached identifiers to hold the computed values.

Case (i) :- $x := y \text{ op } z$

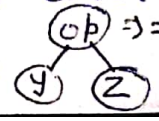
Case (ii) :- $x := \text{op } y$

Case (iii) :- $x := y$

Methods :-

Step ① :- if y operand is undefined then create node (y).
if z operand is undefined then for Case (i) create node (z).

Step ② :- for Case (i) :- Create node (op) whose right child is node (z) and left child is node (y). : $x := y \text{ op } z \Rightarrow$

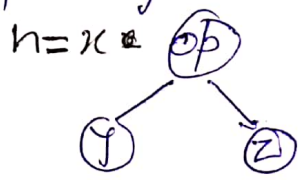


for Case (ii) :- check whether there is node (op) with one child node (y). $x = op\,y$

for Case (iii) :- node n will be (y). , $x = y$.

~~Step 3~~ :-

Output :- for node (x) delete x from list of identifiers. Append x to attached identifiers list for the node n found in step 2. finally set node (x) to n.



DAG - Example

eg :-

$a = b * c$
$d = b$
$e = d * c$
$b = e$
$f = b + c$
$g = d + f$

Sol'n :-

⊛ Translation of Assignment Statements:-

In SDT, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array & records

eg:- $S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow (E_1)$

$E \rightarrow id$

Soln:

The translation Scheme of this grammar is:

Production Rule + Semantic actions

① $S \rightarrow id := E$

```
{ P = look_up (id.name);
  suppose
   if P ≠ nil then
   Emit (P = E.place) /
   else
    Error ;
}
```

② $E \rightarrow E_1 + E_2$

```
{ E.place = newtemp ( );
  Emit (E.place = E1.place + E2.place) //TAC
}
```

E.place = t1
t1 = t1 + t2

③ $E \rightarrow E_1 * E_2$

```
{ E.place = newtemp( );
  Emit (E.place = E1.place * E2.place) //TAC
}
```

④ $E \rightarrow (E_1)$

```
{ E.place = E1.place }
```

⑤ $E \rightarrow id$

```
{ P = look_up (id.name);
  if P ≠ Nil then
  Emit (P = E.place)
  else
                        / error ;
}
```