Module 4

Syntax Directed Translation & Intermediate code Generation.

Syntax Directed Translation.

Semantic analysis –

major function of semantic analysis is type checking. [checking the meaning and providing the meaning].

Syntax Directed Translation / Definition (SDT/SDD) is merely a CFG in which a program fragment called output action or Semantic action or semantic rule is associated with each production.

The semantic rule may involve,

→ the computation of values for Variables belonging to the compiler.

→ the generation of intermediate code

→ the printing of an error diagnostic or the placement of some value in a table.

A value associated with a grammar symbol is called a translation of that symbol.

In SDT, along with the grammar, we associate some informal notations and these notations are called semantic rules.

So

Grammar + Semantic rule = SDT

SDT Scheme

→ It is a CFG.

→ It is used to evaluate the order of semantic rules.

→ the semantic rules are embedded with in the right side of the productions

→ This can be written as { rule }

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

If 'x' is a symbol and 'a' is one of its attribute, then x.a denotes value at node 'x'.

Attributes may be numbers, strings, references, data types ..etc.

Eg:-

Productions	Semantic rule
$E \rightarrow E+T$	$\{E.Val = E.Val + T.Val\}$
$E \rightarrow T$	$\{E.Val = T.Val\}$.

SDT is a technique used to seamlessly integrate semantic analysis with syntax analysis during parsing.

keypoints

- SDT performs translation actions (like assigning variable types or generating code) as the parser encounters each grammar rule, eliminating the need for a seperate semantic analysis phase.

- 'Attributes' associated with grammar symbols, which can store information like variable types, values or mly locations.

- Applications:-
  - → Evaluating arithmetic expressions
  - → converting infix notation to postfix
  - → Perform type checking
  - → generating intermediate code.

Annotated Parse tree :

A parse tree showing values of attributes at each node is called annotated parse tree.

The process of computing the attribute value is called annotating or decorating parse tree.

Types of Attributes:

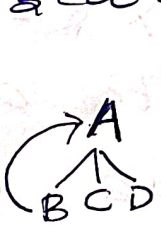
1) Synthesized Attribute

If node takes value from its children then it is synthesized attribute.

Eg:- $A \rightarrow BCD$

A ↙ parent node

BCD → children nodes.



$A.S \longrightarrow B.S$
$A.S \longrightarrow C.S$
$A.S \longrightarrow D.S$

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

2) Inherited Attribute:

If a node takes value from its parent or siblings, then it is inherited attribute.

Eg:- $A \rightarrow BCD$

$B.i = A.i$ ⟶ parent node

$C.i = B.i$
$C.i = D.i$

} ⟶ sibling node

Types of SDD

1) S-Attributed SDD
2) L-Attributed SDD

S-Attributed SDD or S-Attributed grammar.

→ It uses only synthesized attributes.

Eg: $A \rightarrow BCD$

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

→ semantic actions are always placed on right end of the production.

Eg: $S \rightarrow BC$ { ____ }.

→ Attributes are evaluated generally during bottom-up traversal.

L-Attributed SDD

→ It uses both synthesized and inherited attributes.
(But each inherited attribute is restricted to inherit from either parent or left sibling.)

Eg:- $A \rightarrow XYZ \{ Y.S = A.S$ ✓

$Y.S = X.S$ ✓

$Y.S = Z.S$ ✗ $\}$

→ Semantic actions are placed anywhere on the RHS.

Eg:- $S \rightarrow BC \{ \_\_\_ \}$

$S \rightarrow B \{ \_\_\_ \} C$

$S \rightarrow \{ \_\_\_ \} BC$

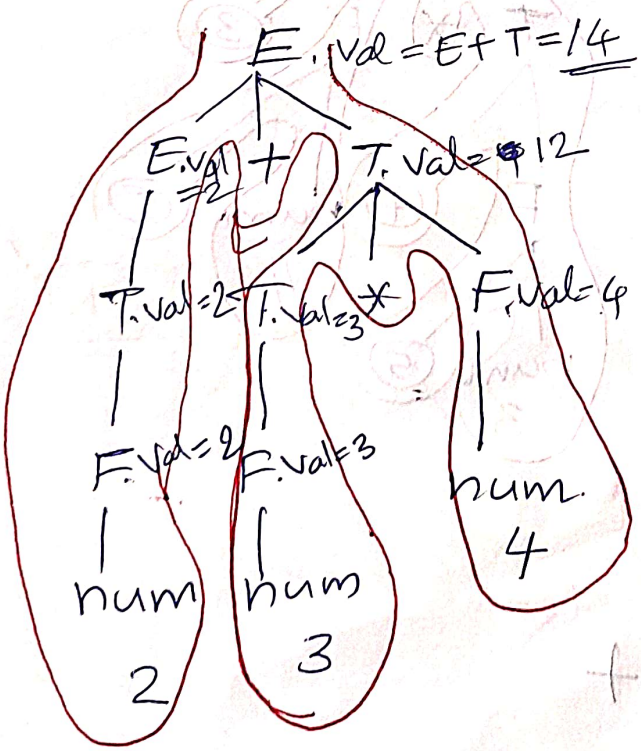
→ Attributes are evaluated by traversing the parse tree by depth first, left-right traversal.

SDT for evaluation of expression

$E \rightarrow E+T$ / T	{ E.Val = E.Val + T.Val } { E.Val = T.Val }
$T \rightarrow T*F$ / F	{ T.Val = T.Val * F.Val } { T.Val = F.Val }
$F \rightarrow num$	{ F.Val = num.lvalue }

How to evaluate some expressions, for example 2+3*4 using SDT scheme.

First we've to create the parse tree for the given expression.



lvalue → lexical value.
what these value actually have.

$E \rightarrow E+T$ { printf("+"); } ①

$/T$ { } ②

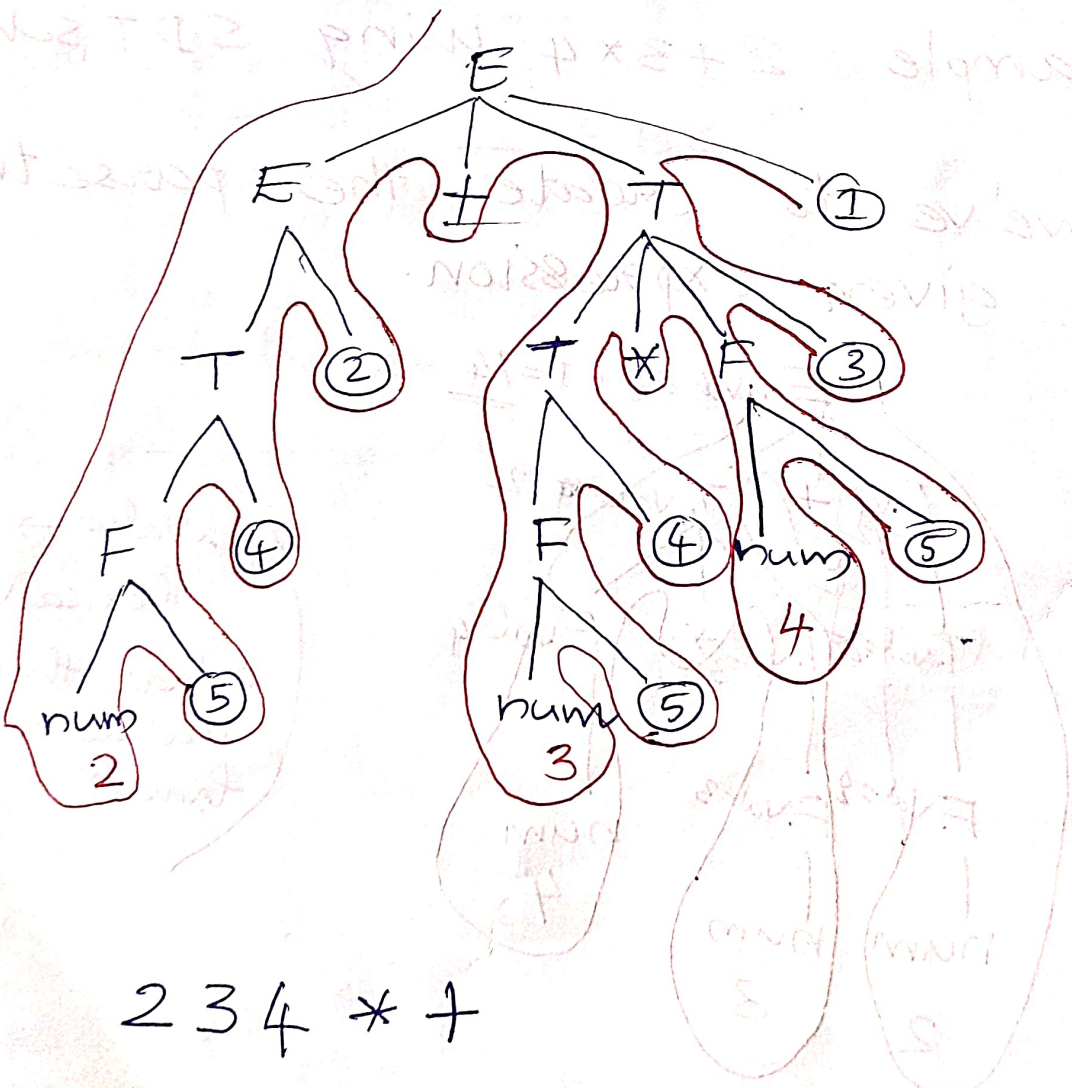
$T \rightarrow T*F$ { printf("*"); } ③

$/F$ { } ④

$F \rightarrow num$ { printf(num.lval); } ⑤

Convert the given infix expression to postfix.

$2+3*4$



$234*+$

→ postfix expression.

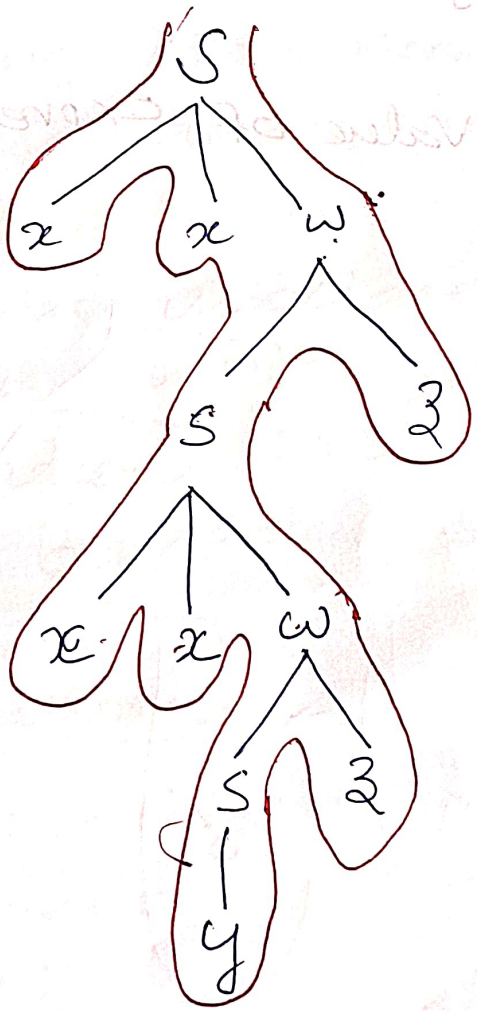
Example ①

$S \longrightarrow xxw$ { printf (1); }

$\quad\quad / y$ { printf (2); }

$W \longrightarrow S3$ { printf (3); }

What will be the output for the string $xxxxy33$. by SDT?



23131 ⟶ output

Example ②

$E \longrightarrow E * T \quad \{E.Val = E.Val * T.Val\}$

$\quad / T \quad \{E.Val = T.Val\}$

$T \longrightarrow F - T \quad \{F.Val - T.Val\}$

$\quad / F \quad \{T.Val = F.Val\}.$

$F \longrightarrow 2 \quad \{F.Val = 2\}$

$F \longrightarrow 4 \quad \{F.Val = 4\}.$

What will be the value of expression

$4 - 2 - 4 * 2$

Examples:

Draw an annotated parse tree for expression $3*5+4\text{n}$

$L \longrightarrow En$ { printf (E.val); }

$E \longrightarrow E_1 + T$ { E.val = E_1.val + T.val }

$E \longrightarrow T$ { E.val = T.val }

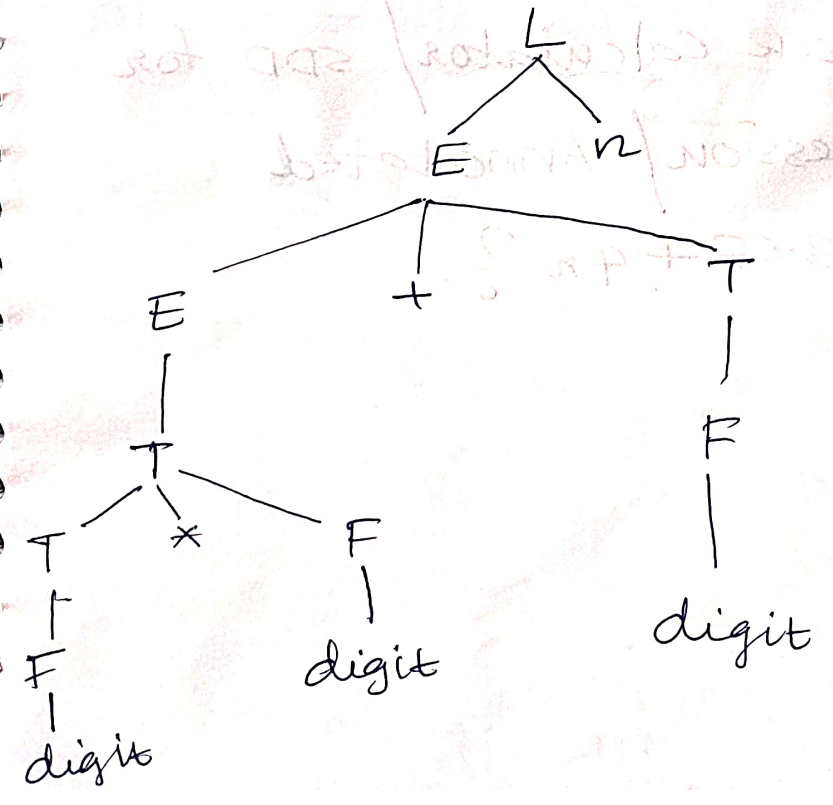
$T \longrightarrow T_1 * F$ { T.val = T_1.val * F.val }

$T \longrightarrow F$ { T.val = F.val }

$F \longrightarrow (E)$ { F.val = E.val }

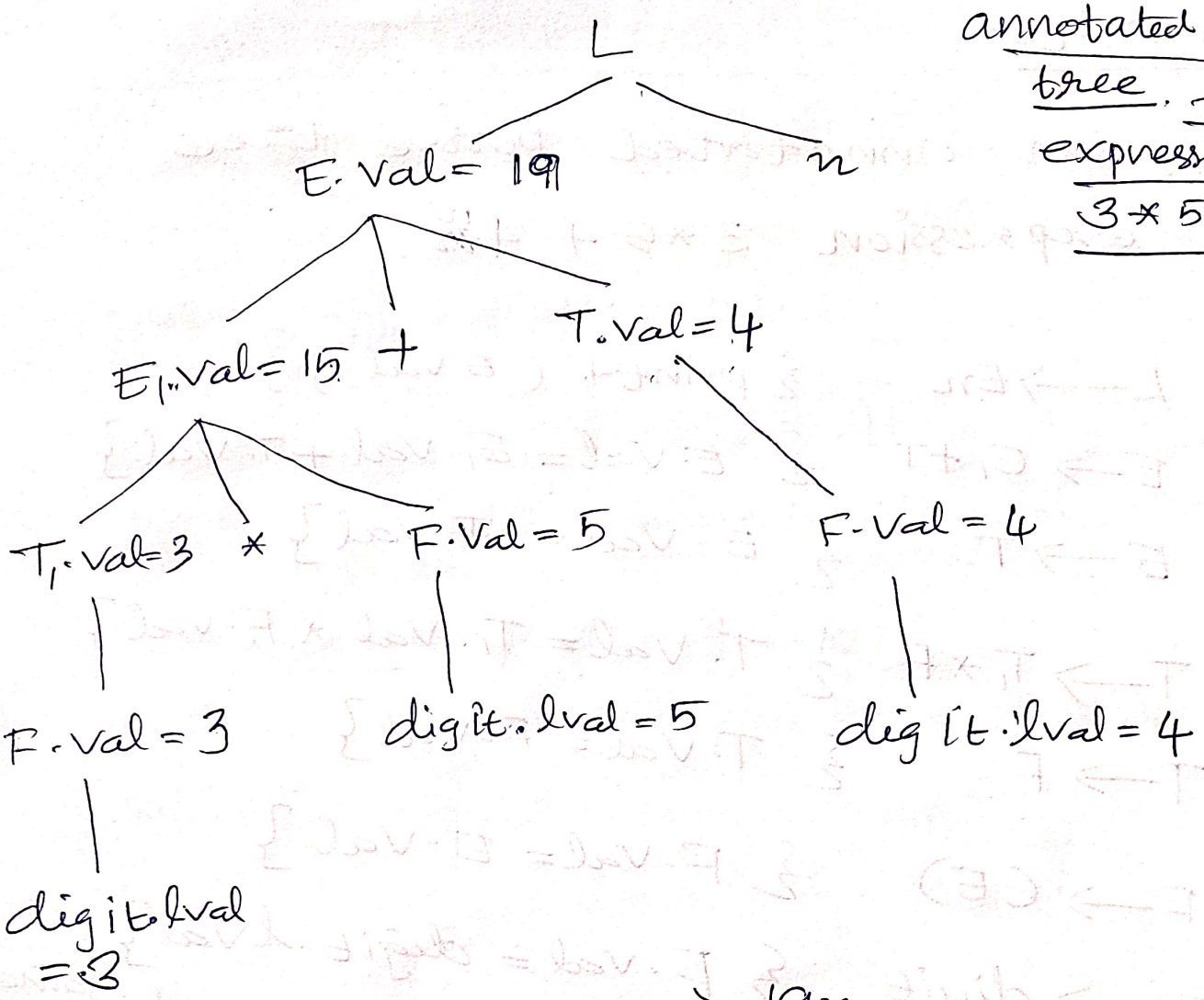
$F \longrightarrow digit$ { F.val = digit.lval }

(En, E_1 and E are the same. T_1 and T are the same)



⟹ parse tree.

annotated parse tree for the expression 3*5+4.



$\Rightarrow$ 19n

(Qn.)

SDD of a simple Desk calculator / SDD for evaluation of expression / Annotated parse tree for 3*5+4n ?

Qn.

SDD for simple type declarations.

$D \rightarrow TL \quad \{L.inh = T.type\}$

$T \rightarrow int \quad \{T.type = int\}$

$T \rightarrow float \quad \{T.type = float\}$

$L \rightarrow L_1, id \quad \{L_1.inh = L.inh$
addType (id.entry, L.inh)}

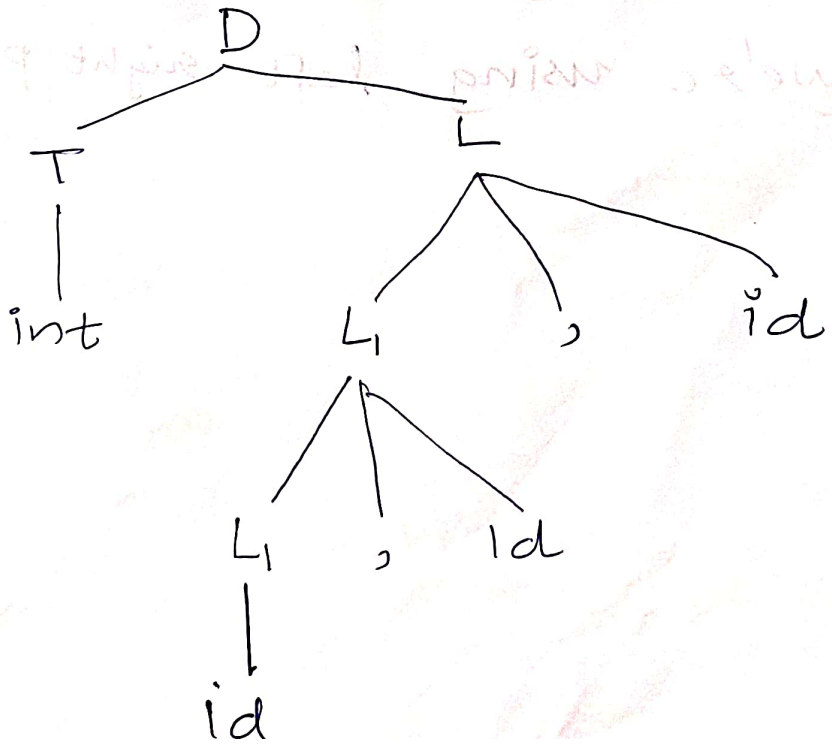
$L \rightarrow id \quad \{addType(id.entry, L.inh)\}$

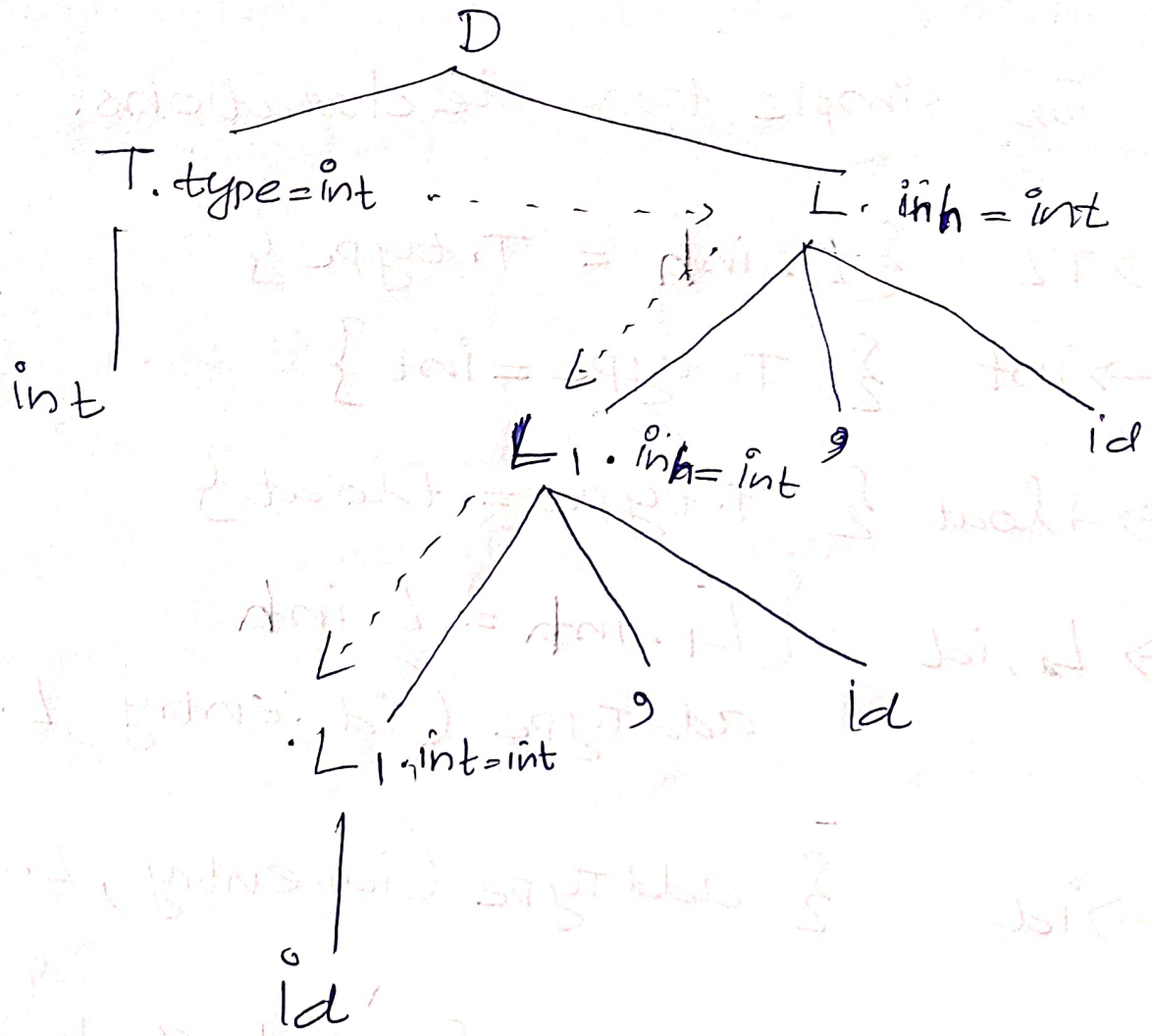
Annotated parse tree for 'int a, b, c'.

id.entry → a lexical value points to a symbol table object.

L.inh → the type being assigned to every identifier on the list.

Soln:
Creating parse tree:




D
T.type=int
L.in=int
int
L
L1.in=int
,
id
L
L1.in=int
,
id
L1.in=int
id

Dependency Graph

→ Dependency graph represents the flow of information among the attributes in a parse tree.

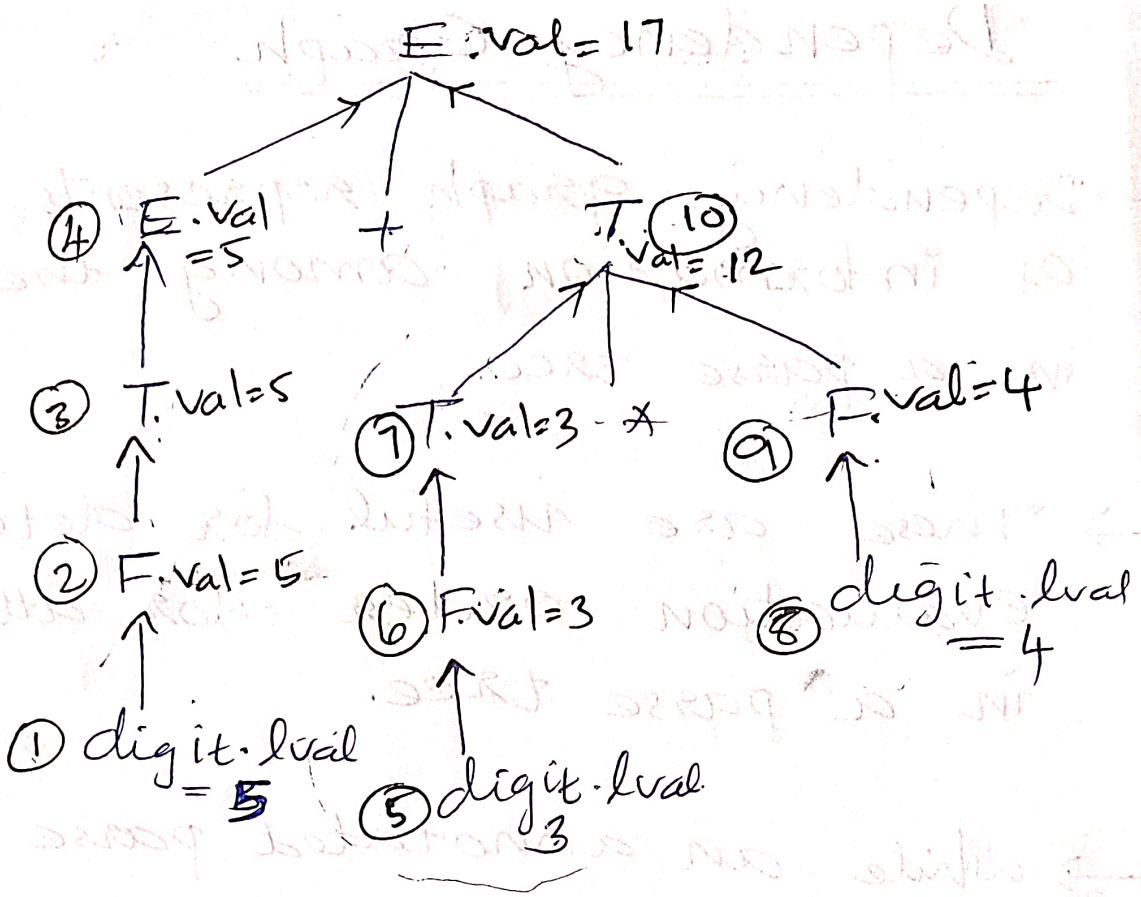
→ These are useful for determining evaluation order for attributes in a parse tree.

→ while an annotated parse tree shows the values of attributes, a dependancy graph determines how those values can be computed.

Example :

$E \rightarrow E+T$	$\{E.Val = E.Val + T.Val\}$
$E \rightarrow T$	$\{E.Val = T.Val\}$
$T \rightarrow T*F$	$\{T.Val = T.Val * F.Val\}$
$T \rightarrow F$	$\{T.Val = F.Val\}$
$F \rightarrow digit$	$\{F.Val = digit.lval\}$

String for evaluation → 5+3*4



The above is the dependency graph. arrow shows the flow of values and numbers shows the order of evaluation of expression.

Bottom-up evaluation of S-attributed Definitions

'23*5+4'

Production	Program fragment
$S \rightarrow E\$$	print VAL[TOP]
$E \rightarrow E+E$	VAL[TOP] = VAL[TOP] + VAL[TOP-2]
$E \rightarrow E*E$	Val[TOP] = Val[TOP] * VAL[TOP-2]
$E \rightarrow (E)$	Val[TOP] = Val[TOP-1]
$E \rightarrow I$	—
$I \rightarrow I.digit$	Val[TOP] = 10 * Val[TOP] + ~~Val~~ len.Val.
$I \rightarrow digit$	Val[TOP] = len.val

$E \rightarrow (E)$

Val[TOP] = Val[TOP-1]

Stack

C	TOP
E	TOP-1
)	TOP-2

$E \rightarrow E+E$

Val[TOP] = Val[TOP] + Val[TOP-2].

E	TOP
+	TOP-1
E	TOP-2

I	TOP
digit	TOP-1

I * 10 + digit

$S \rightarrow E\ \$$ { print E.val }

$E \rightarrow E+E$ { E.val = E.val + E.val }

$E \rightarrow E*E$ { E.val = E.val * E.val }

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

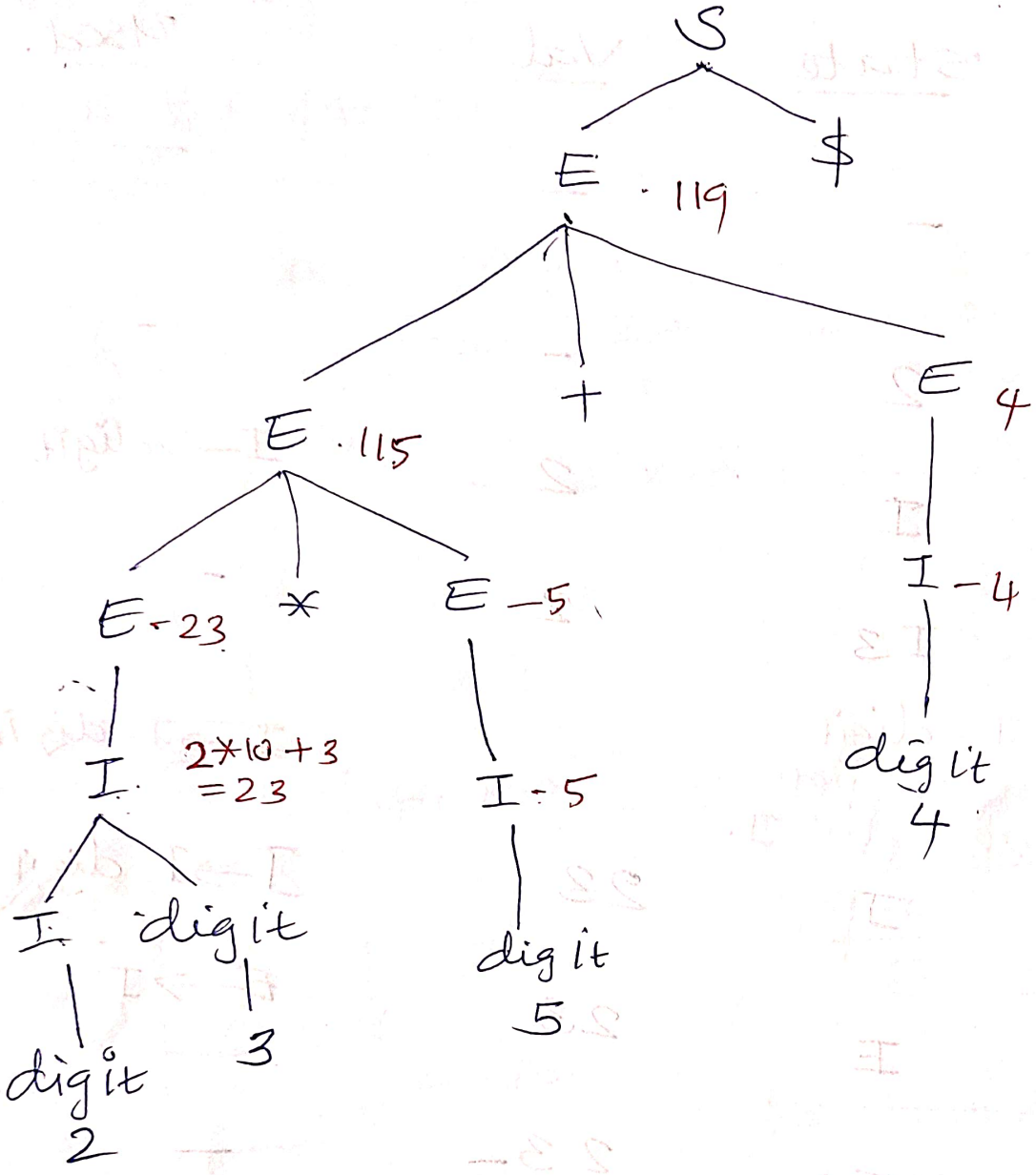
$E \rightarrow I$ { E.val = I.val }

$I \rightarrow I.digit$ { I.val = I.val * 10 + digit.lval }

$I \rightarrow digit$ { I.val = digit.lval. }.

Evaluate the expression 23 * 5 + 4

· parse tree // annotated parse tree.



Implementation Using Stack.

Input	stack state	Val	Production Used.
		—	—
23*5+4$	—	—	—
3*5+4$	2	2	I → digit
3*5+4$	I	2	—
*5+4$	I3	2	~~I → I digit~~
	I. digit then pop and push I.		
*5+4$	I	23	I → I digit
*5+4$	E	23	E → I
5+4$	E*	23–	—
+4$	E*5	23––	—
+4$	E*I	23–5	I → digit
+4$	E*E	23–5	E → I
+4$	E	115	E → E*E
4$	E+	115–	—
$	E+4	115––	—