

# Module 3 – Semantic Analysis

# Syntax directed definitions

- Syntax directed definition is a generalization of context free grammar in which **each grammar symbol has an associated set of attributes**.
- The attributes can be a number, type, memory location, return type etc....
- Types of attributes are:
  1. Synthesized attribute
  2. Inherited attribute

E. Return Type

# Syntax Directed Translation Scheme

- The Syntax directed translation scheme is a context free grammar
- 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 the action to be executed is shown by enclosed between braces.

# Synthesized attributes

- Value of synthesized attribute at a node can be computed from the value of attributes at the **children of that node** in the parse tree.
- A syntax directed definition that uses synthesized attribute exclusively is said to be **S-attribute definition**.
- Example: Syntax directed definition of simple desk calculator

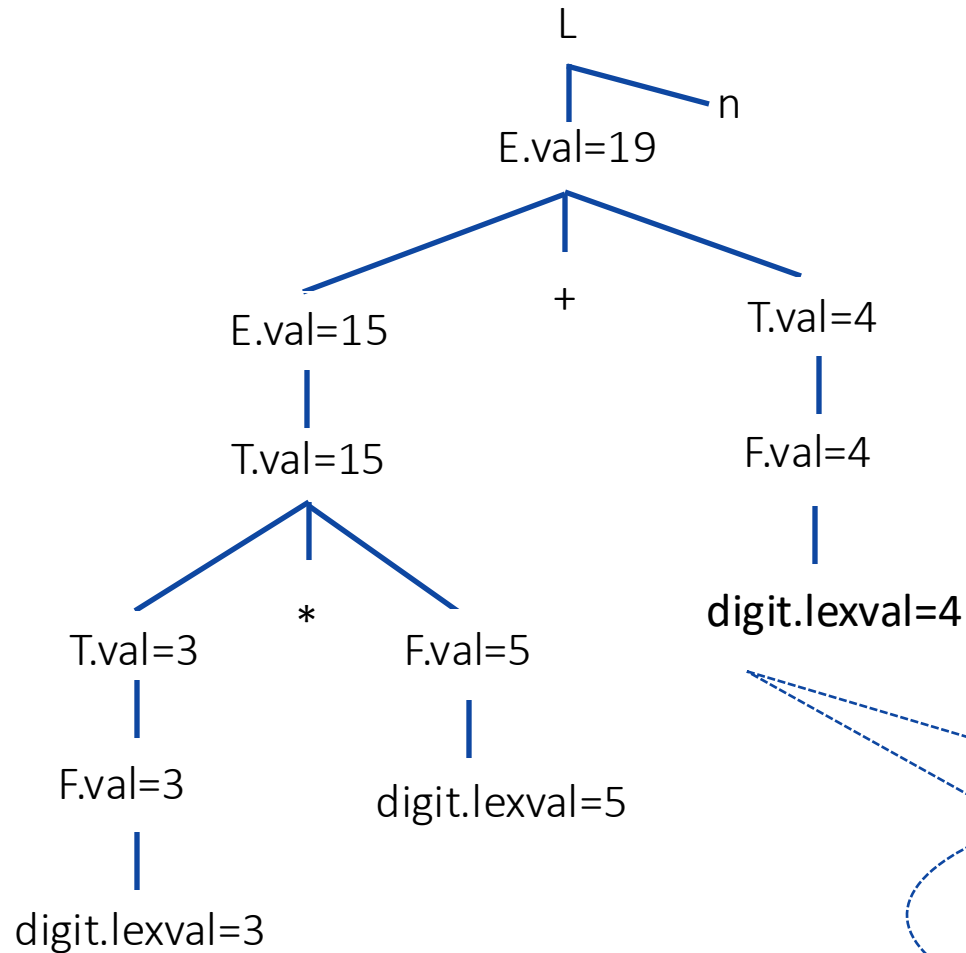
Production	Semantic rules
$L \rightarrow E_n$	
$E \rightarrow E_1 + T$	
$E \rightarrow T$	
$T \rightarrow T_1 * F$	
$T \rightarrow F$	
$F \rightarrow (E)$	
$F \rightarrow \text{digit}$	

# Applications of SDT

- Executing Arithmetic Expression
- Conversion from Infix to Postfix
- Conversion from Infix to Prefix
- Conversion from Binary to Decimal
- Counting No. of Reductions
- Creating Syntax Tree
- Generating Intermediate Code
- Type Checking
- Storing Type information into Symbol Table.

# Example: Synthesized attributes

String: 3\*5+4n;



Annotated parse tree for 3\*5+4n

The process of computing the attribute values at the node is called **annotating** or **decorating the parse tree**

parse tree showing the value of the attributes at each node is called Annotated parse tree

Production	Semantic rules
$L \rightarrow E_n$	Print (E.val)
$E \rightarrow E_1 + T$	$E.Val = E_1.val + T.val$
$E \rightarrow T$	$E.Val = T.val$
$T \rightarrow T_1 * F$	$T.Val = T_1.val * F.val$
$T \rightarrow F$	$T.Val = F.val$
$F \rightarrow (E)$	$F.Val = E.val$
$F \rightarrow \text{digit}$	$F.Val = \text{digit} . \text{lexval}$

# Exercise

Draw Annotated Parse tree for following:

1.  $7+3*2n$
2.  $(3+4)*(5+6)n$

# Syntax directed definition to translate arithmetic expressions from infix to prefix notation

Production	Semantic rules
$L \rightarrow E$	
$E \rightarrow E + T$	
$E \rightarrow E - T$	
$E \rightarrow T$	
$T \rightarrow T * F$	
$T \rightarrow T / F$	
$T \rightarrow F$	
$F \rightarrow F \wedge P$	
$F \rightarrow P$	
$P \rightarrow (E)$	
$P \rightarrow \text{digit}$	



# Inherited attribute

- An inherited value at a node in a parse tree is computed from the value of attributes at the **parent and/or siblings** of the node.

Production	Semantic rules
$D \rightarrow T L$	
$T \rightarrow \text{int}$	
$T \rightarrow \text{real}$	
$L \rightarrow L_1 , \text{id}$	)
$L \rightarrow \text{id}$	

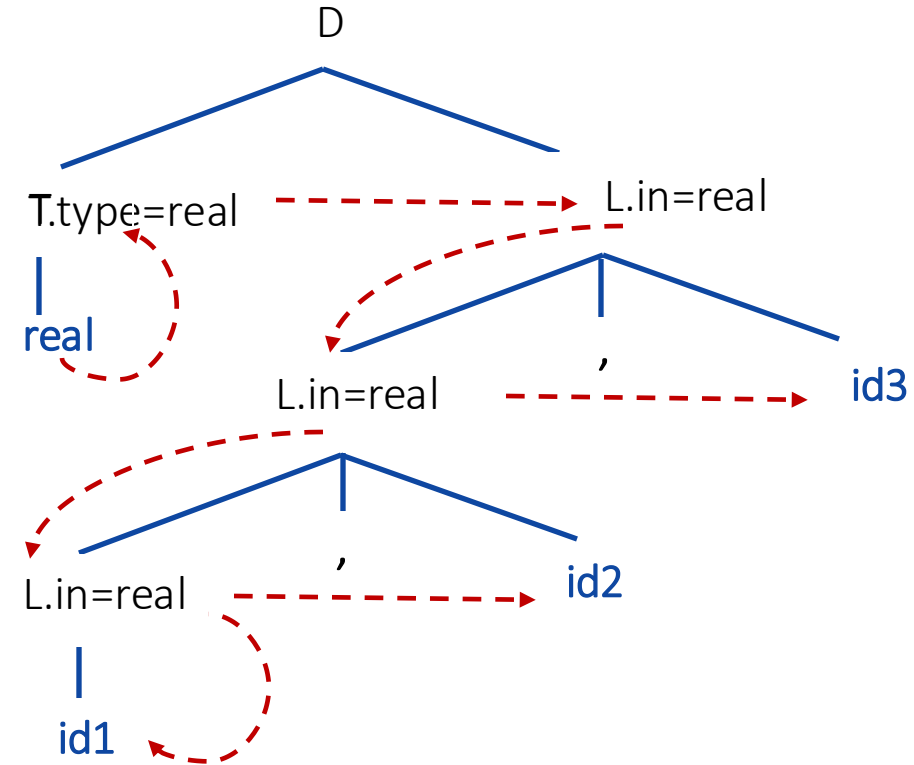
Syntax directed definition with inherited attribute L.in

- Symbol T is associated with a **synthesized attribute type**.
- Symbol L is associated with an **inherited attribute in**.

# Example: Inherited attribute

Example: Pass data types to all identifier  
real id1,id2,id3

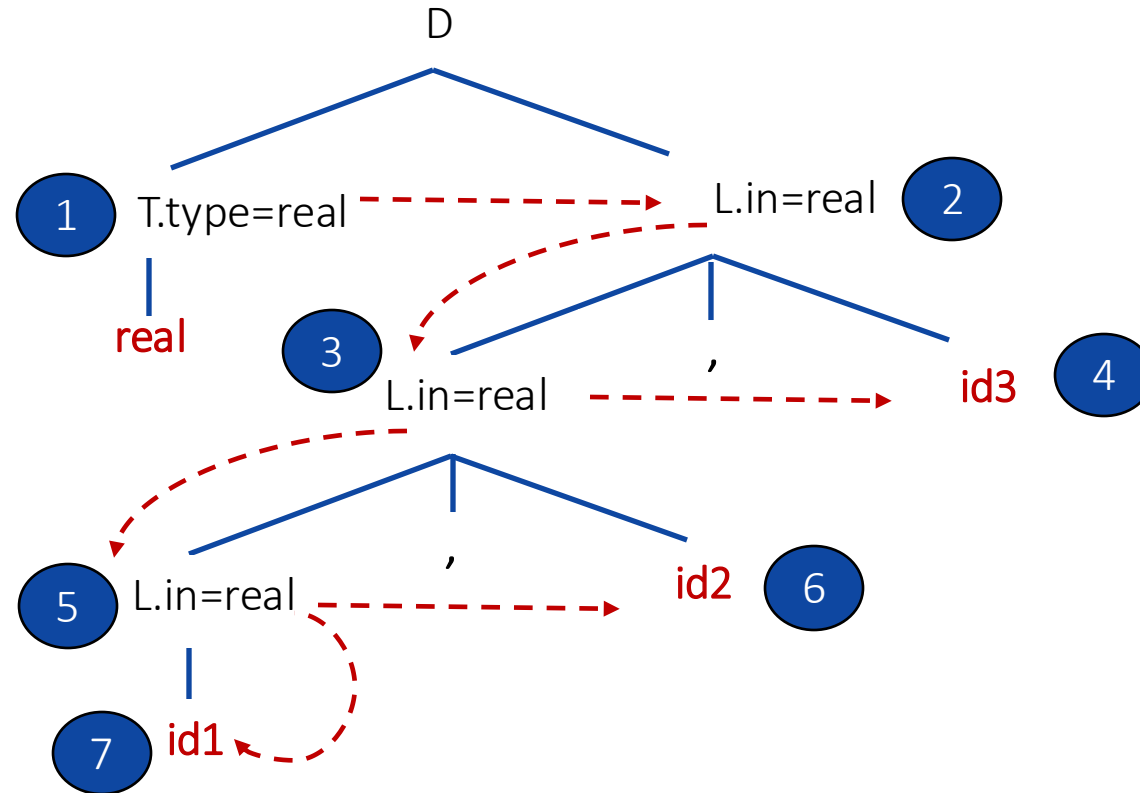
Production	Semantic rules
$D \rightarrow T L$	$L.in = T.type$
$T \rightarrow \text{int}$	$T.type = \text{integer}$
$T \rightarrow \text{real}$	$T.type = \text{real}$
$L \rightarrow L_1 , id$	$L_1.in = L.in,$ $\text{addtype}(id.entry, L.in)$
$L \rightarrow id$	$\text{addtype}(id.entry, L.in)$



$D \rightarrow T L_1 , id$

# Evaluation order

- A topological sort of a directed acyclic graph is any ordering  $m_1, m_2, \dots, m_k$  of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.
- If  $m_i \rightarrow m_j$  is an edge from  $m_i$  to  $m_j$  then  $m_i$  appears before  $m_j$  in the ordering.



# Construction of syntax tree

- Following functions are used to create the nodes of the syntax tree.
  1. **Mknode (op,left,right):** creates an **operator node** with label **op** and two fields containing **pointers to left and right**.
  2. **Mkleaf (id, entry):** creates an **identifier node** with label **id** and a field containing entry, a pointer to the symbol table **entry** for the identifier.
  3. **Mkleaf (num, val):** creates a **number node** with label **num** and a field containing **val**, the value of the number.

# Construction of syntax tree for expressions

**Example:** construct syntax tree for  $a-4+c$

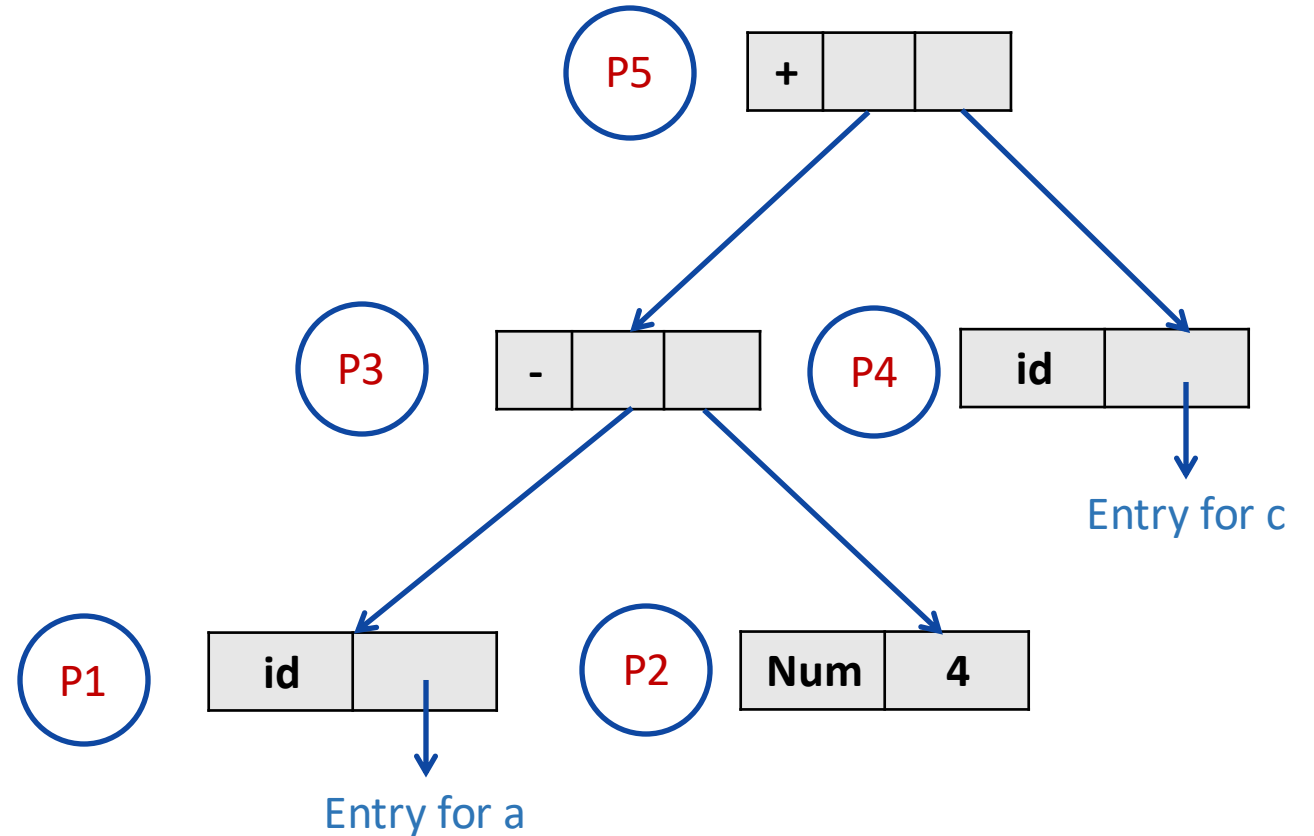
P1: mkleaf(id, entry for a);

P2: mkleaf(num, 4);

P3: mknode('-', p1, p2);

P4: mkleaf(id, entry for c);

P5: mknode('+', p3, p4);



# Bottom up evaluation of S-attributed definitions

- S-attributed definition is one such class of syntax directed definition with synthesized attributes only.
- Synthesized attributes can be evaluated using bottom up parser only.

## Synthesized attributes on the parser stack

- Consider the production  $A \rightarrow XYZ$  and associated semantic action is  $A.a = f(X.x, Y.y, Z.z)$

	State	Value
top-2 →	X	X.x
top-1 →	Y	Y.y
top →	Z	Z.z

Before reduction



	State	Value
top →	A	A.a

After reduction



# Bottom up evaluation of S-attributed definitions

Production	Semantic rules
$L \rightarrow E_n$	Print (val[top])
$E \rightarrow E_1 + T$	val[top]=val[top-2] + val[top]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	val[top]=val[top-2] * val[top]
$T \rightarrow F$	
$F \rightarrow (E)$	val[top]=val[top-2] - val[top]
$F \rightarrow \text{digit}$	

Implementation of a desk calculator  
with bottom up parser

Input	State	Val	Production Used
3*5n	-	-	

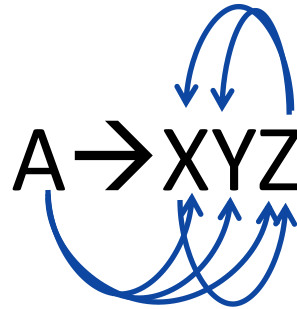
Move made by translator



# L-Attributed definitions

- A syntax directed definition is L-attributed if each inherited attribute of  $X_j$ ,  $1 \leq j \leq n$ , on the right side of  $A \rightarrow X_1 X_2 \dots X_n$  depends only on:
  - The attributes of the symbols  $X_1, X_2, \dots, X_{j-1}$  to the left of  $X_j$  in the production and
  - The inherited attribute of A.

- Example:



~~Not L-Attributed~~

Production	Semantic Rules
$A \rightarrow LM$	$L.i := l(A.i)$ $M.i := m(L.s)$ $A.s := f(M.s)$
$A \rightarrow QR$	$R.i := r(A.i)$ $Q.i := q(R.s)$ $A.s := f(Q.s)$

- Above syntax directed definition is *not L-attributed* because the inherited attribute  $Q.i$  of the grammar symbol  $Q$  depends on the attribute  $R.s$  of the grammar symbol to its right.

# Bottom up evaluation of S-attributed definitions

- Translation scheme is a context free grammar in which attributes are associated with the grammar symbols and semantic actions enclosed between braces { } are inserted within the right sides of productions.
- Attributes are used to evaluate the expression along the process of parsing.
- During the process of parsing the evaluation of attribute takes place by consulting the semantic action enclosed in { }.
- A translation scheme generates the output by executing the semantic actions in an ordered manner.
- This process uses the depth first traversal.

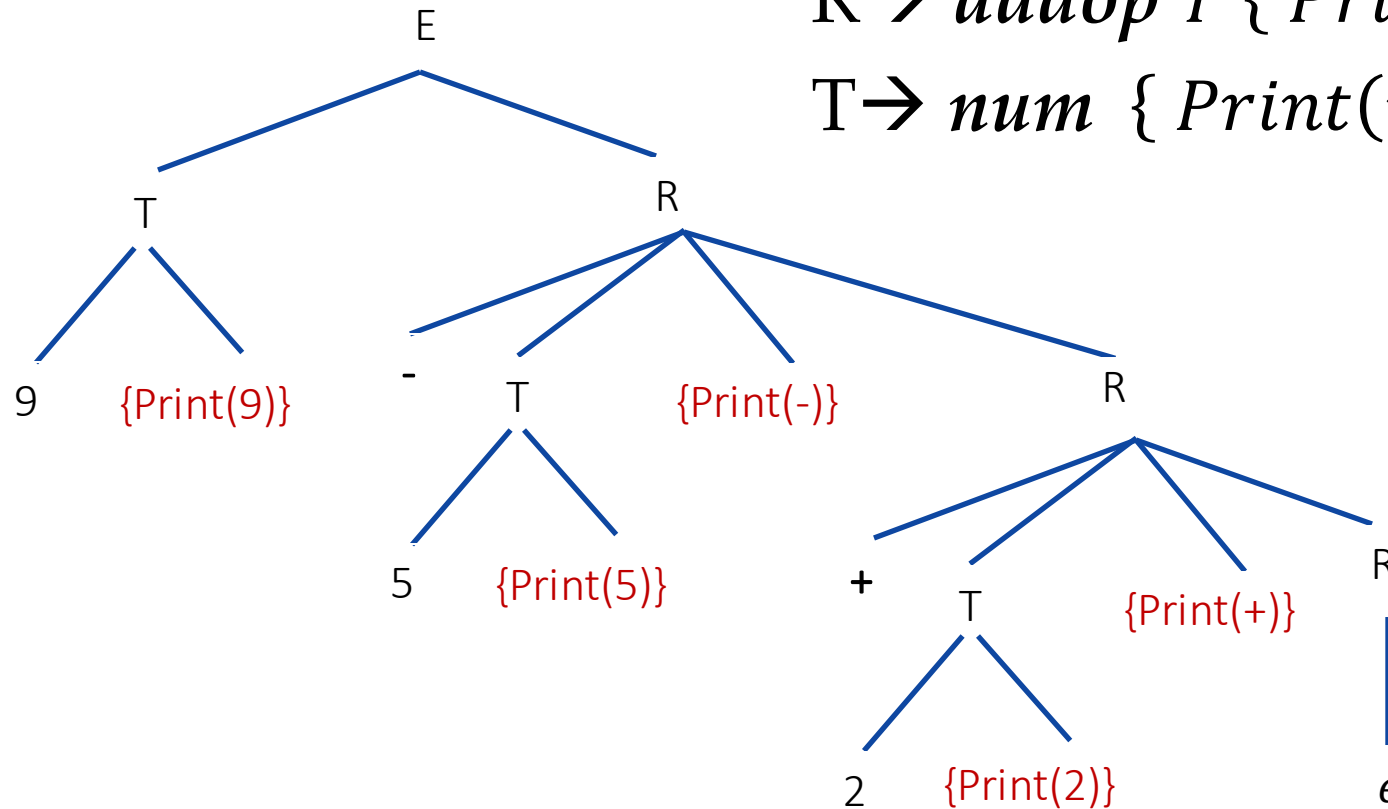
# Example: Translation scheme (Infix to postfix notation)

String: 9-5+2

$E \rightarrow TR$

$R \rightarrow \text{addop } T \{ \text{Print}(\text{addop.lexeme}) \} R1 \mid \epsilon$

$T \rightarrow \text{num} \{ \text{Print}(\text{num.val}) \}$



Now, Perform Depth first traversal

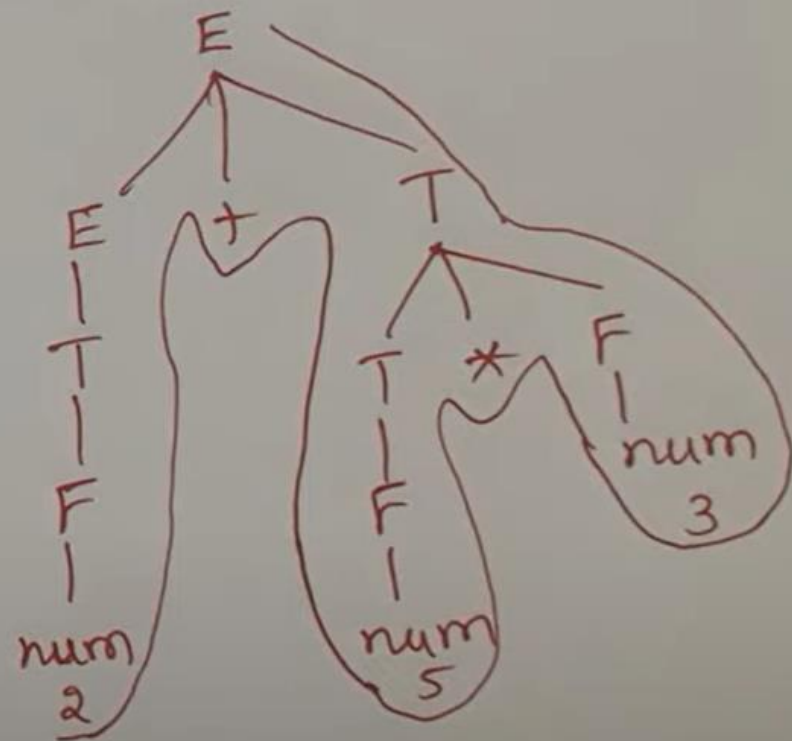
Postfix=95-2+

# SDT for converting infix to postfix expression

$a+b \Rightarrow ab+$

$E \rightarrow E + T$	$\{ \text{printf}("+"); \}$
$E \rightarrow T$	$\{ \}$
$T \rightarrow T * F$	$\{ \text{printf}("*"); \}$
$T \rightarrow F$	$\{ \}$
$F \rightarrow \text{num}$	$\{ \text{printf}(\text{num.lval}); \}$

Ex:  $2 + 5 * 3$

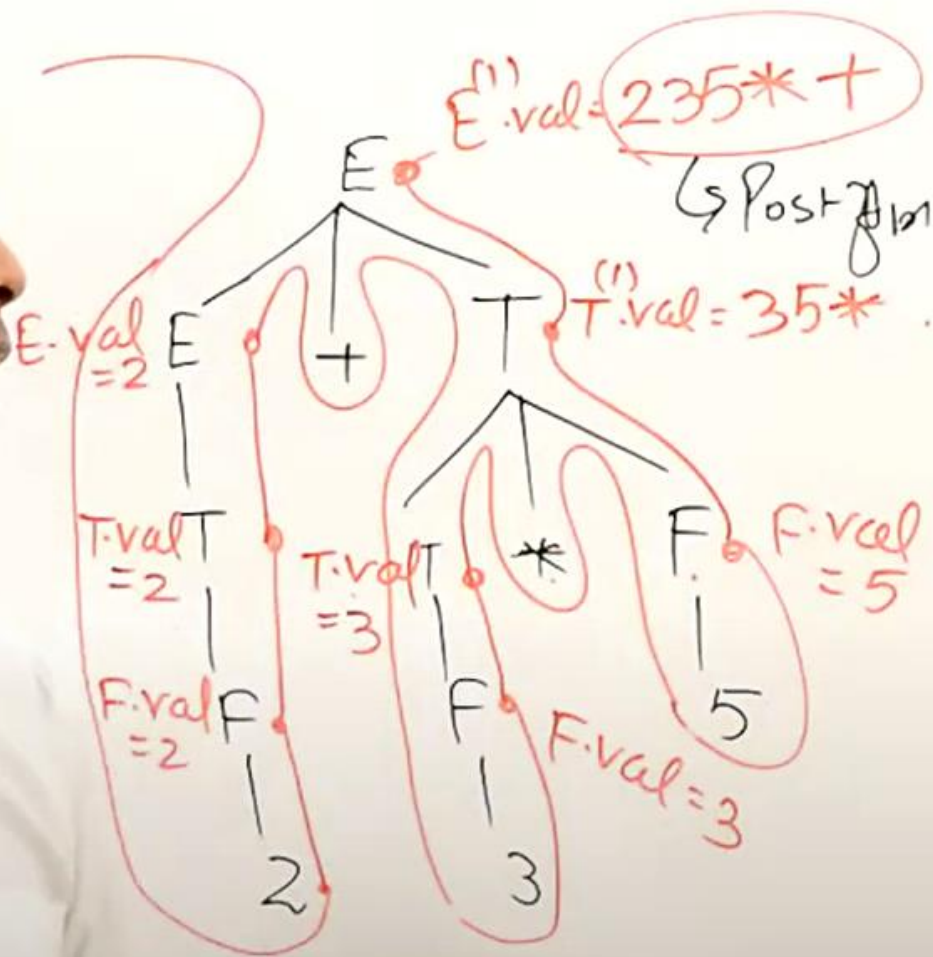


postfix notation

2 5 3 \* +

# SDD for conversion of an Infix Expression to Postfix Expression

For string  $w = 2 + 3 * 5$



SDD

Postfix

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

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

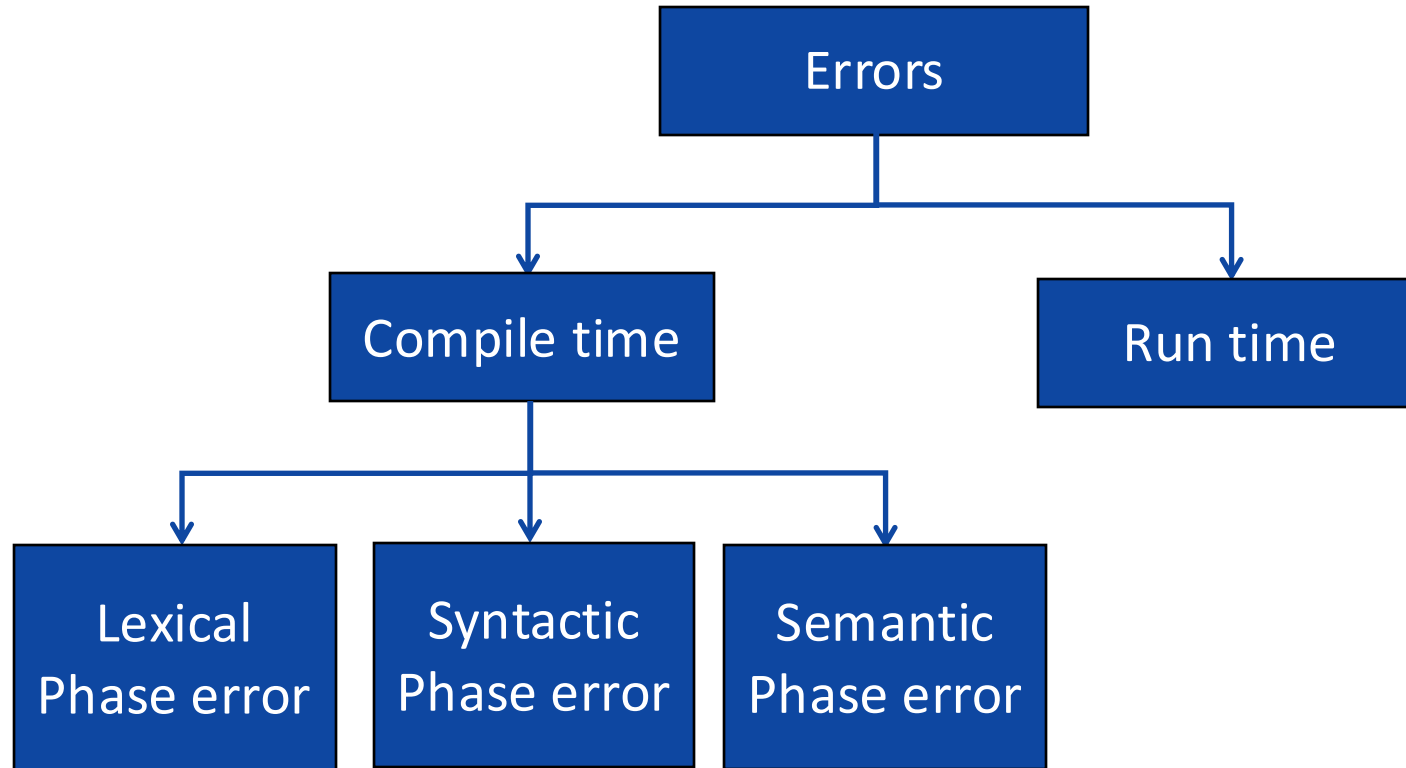
$T \rightarrow T * F \{ T.val = T.val || F.val || * \}$

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

$F \rightarrow 0 | 1 | 2 | 3 | \dots | 9 \{ F.val = 0 | 1 | 2 | \dots | 9 \}$

# Types of errors

# Types of Errors



# Lexical error

- Lexical errors can be detected during lexical analysis phase.
- Typical lexical phase errors are:

1. Spelling errors
2. Exceeding length of identifier or numeric constants
3. Appearance of illegal characters

- Example:

```
fi ( )  
{  
}
```

- In above code 'fi' cannot be recognized as a misspelling of keyword *if* rather lexical analyzer will understand that it is an identifier and will return it as valid identifier.
- Thus misspelling causes errors in token formation.



# Syntax error

- Syntax error appear during syntax analysis phase of compiler.
- Typical syntax phase errors are:
  1. Errors in structure
  2. Missing operators
  3. Unbalanced parenthesis
- The parser demands for tokens from lexical analyzer and if the tokens do not satisfy the grammatical rules of programming language then the syntactical errors get raised.
- Example:

`printf("Hello World !!!")`

← Error: Semicolon missing

# Semantic error

- Semantic error detected during semantic analysis phase.
- Typical semantic phase errors are:
  1. Incompatible types of operands
  2. Undeclared variable
  3. Not matching of actual argument with formal argument
- Example:  
$$\text{id1} = \text{id2} + \underline{\text{id3}} * 60$$

(Note: id1, id2, id3 are real)

(Directly we can not perform multiplication due to incompatible types of variables)