1. SYNTAX ANALYSIS

RGPV Notes Hub

1. Introduction :

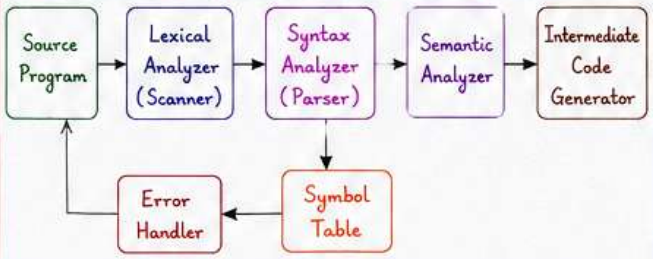
Syntax Analysis (Parsing) is the second phase of a compiler. The lexical analyzer generates a stream of tokens from the source program. The syntax analyzer receives this token stream and checks whether the tokens can be arranged according to the grammar rules of the language or not.

If the arrangement is correct, it creates a parse tree and passes the information to the next phase (Semantic Analyzer).

2. Purpose of Syntax Analysis :

- To check the grammatical structure of the source program.
- To verify whether the sequence of tokens follows the rules of CFG (Context Free Grammar).
- To build the parse tree or syntax tree.
- To detect and recover from syntax errors.
- To provide input to the Semantic Analyzer.

3. Position of Syntax Analysis in Compiler :



4. Input and Output of Syntax Analyzer :

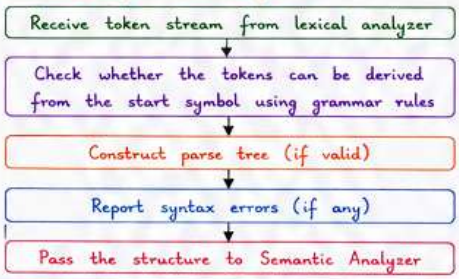
(i) Input :

- Stream of tokens < token-name, attribute-value > from lexical analyzer.
- Grammar or set of production rules.
- Information from symbol table.

(ii) Output :

- If input is syntactically correct → parse tree / abstract syntax tree.
- If input is incorrect → error messages.
- Updated information in symbol table.

5. Process of Syntax Analysis :



6. Grammar Used in Syntax Analysis :

The grammar used is Context Free Grammar (CFG). It is of the form,

$$G = (V, T, P, S)$$

Where,

V = Set of non-terminal symbols
T = Set of terminal symbols
P = Set of production rules
S = Start symbol

$V \cap T = \phi$ and $V \cup T \neq \phi$

Example :

$E \rightarrow E + T \mid T$

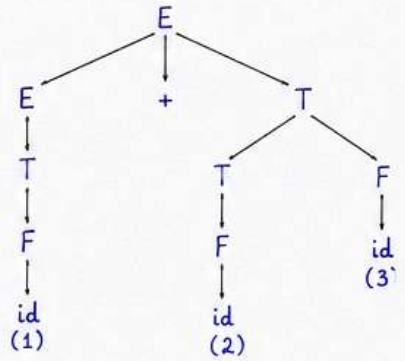
$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

7. Parse Tree (Syntax Tree) :

A parse tree represents the syntactic structure of the source program according to grammar.

Example : For expression id + id * id



Important :

Leaves of parse tree are either operators or operands (id, constants etc.). Each internal node represents a non-terminal.

8. Types of Parsing Methods :

(i) Top - Down Parsing :

- Starts from start symbol and tries to produce the input string.
- Builds parse tree from root to leaves.
- Example : Recursive Descent, Predictive Parsing.

(ii) Bottom - Up Parsing :

- Starts from input string and tries to reduce it to start symbol.
- Builds parse tree from leaves to root.
- Example : Operator Precedence, LR Parsing.

Comparison	
Top - Down	Bottom - Up
Root → Leaves	Leaves → Root
Leftmost derivation	Rightmost derivation
May backtrack	No backtracking
Easy to implement	More powerful

9. Error Handling in Syntax Analysis :

- Detects syntax errors like missing operators, extra tokens, unmatched parentheses etc.
- Reports line number and nature of error.
- Uses error recovery techniques (panic mode, phrase level recovery, error productions etc.) to continue analysis.

10. Importance of Syntax Analysis :

- Ensures that the program follows the grammatical rules.
- Builds hierarchical structure (parse tree) for further processing.
- Detects syntax errors before code generation.
- Forms the basis for semantic analysis and optimization.

Important Points for Exams :

★ Syntax Analysis is also called Parsing.
★ Uses Context Free Grammar.
★ Builds Parse Tree / Syntax Tree.
★ Two main approaches - Top Down and Bottom Up.
★ Essential phase for correct compilation.

2. Context Free Grammar (CFG)

RGPV Notes Hub

1. Introduction :

A Context Free Grammar (CFG) is a formal grammar in which every production rule has a single non-terminal on the left hand side. The right hand side contains any combination of terminals and/or non-terminals (including ε). CFG is used to generate context free languages and is the basis of syntax analysis in compilers.

2. Definition :

A grammar G is said to be a Context Free Grammar if it is a 4-tuple,

$$G = (V, T, P, S)$$

where,

- V = Finite set of Non-terminal symbols
- T = Finite set of Terminal symbols
- P = Finite set of Production rules
- S = Start symbol, $S \in V$
- $V \cap T = \phi$ (V and T are disjoint sets)

3. Production Rule in CFG :

$$A \to \alpha$$

- A is a single non-terminal ($A \in V$)
- α is a string of symbols from $(V \cup T)^*$ (may be ε also)
- ε (epsilon) represents empty string.

4. Example of CFG :

Let $G = (V, T, P, S)$ be a grammar where

$V = \{S\}$ $T = \{a, b\}$

$P = \{S \to aSb \mid S \to \varepsilon\}$ $S = S$

This grammar generates strings of the form $a^n b^n$ where $n \geq 0$.

Generated strings : ε, ab, aabb, aaabbb, aaaabbbb, ...

5. Derivation in CFG :

- Derivation is the process of obtaining a string from the start symbol by repeatedly applying production rules.
- If $\alpha, \beta \in (V \cup T)^*$ and $A \to \gamma \in P$, then

  $\alpha A \beta \Rightarrow \alpha \gamma \beta$ (one step derivation)

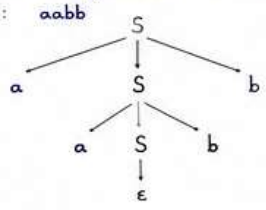
- $\Rightarrow^*$ denotes zero or more steps of derivation.

6. Parse Tree (Syntax Tree) :

A parse tree is a tree representation of the derivation of a string.

Example : For grammar G : $S \to aSb \mid \varepsilon$

String : aabb



Leaves are terminals or ε. Internal nodes are non-terminals.

7. Types of Grammars :

Type	Rule Form	Name	Generates
Type 0	$\alpha \to \beta$ (α contains a non-terminal)	Unrestricted Grammar	Recursively Enumerable languages
Type 1	$\alpha A \beta \to \alpha \gamma \beta$ ($\lvert\gamma\rvert \geq 1$)	Context Sensitive Grammar (CSG)	Context Sensitive languages
Type 2	$A \to \gamma$	Context Free Grammar (CFG)	Context Free languages
Type 3	$A \to aB \mid a \mid \varepsilon$	Regular Grammar (RG)	Regular languages

8. Properties of CFG :

- Each production has exactly one non-terminal on LHS.
- RHS may contain any number of terminals and/or non-terminals.
- ε (empty string) is allowed.
- Parse tree for any string is unique for unambiguous CFG.
- CFGs are more powerful than regular grammars.

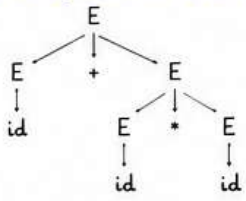
9. Ambiguity in CFG :

A grammar is ambiguous if some string has more than one parse tree (or more than one leftmost/rightmost derivation).

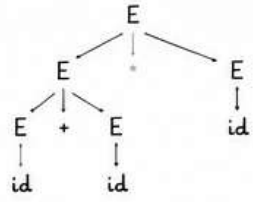
Example : G : $E \to E + E \mid E * E \mid (E) \mid id$

String : id + id * id

This string has two parse trees.



Tree 1 (id + (id * id))



Tree 2 ((id + id) * id)

10. Removal of ε-Productions :

If a non-terminal A has a production $A \to \varepsilon$, it can be removed by adding productions for all places where A appears.

Example : Before removing ε After removing ε

$S \to AB \mid a$
$A \to \varepsilon \mid a$
$B \to b$

$\Rightarrow$

$S \to AB \mid a \mid B \mid a$
$A \to a$
$B \to b$

11. Removal of Unit Productions :

If there is a rule $A \to B$ (where $A, B \in V$), it is called unit production. They can be removed by replacing A with the productions of B.

Example : Before removing unit production

$S \to A \mid a$
$A \to B \mid b$
$B \to c$

$\Rightarrow$

After removing unit production

$S \to b \mid c \mid a$
$A \to b \mid c$
$B \to c$

Important Points for Exams :

- ★ CFG is a 4-tuple (V, T, P, S).
- ★ Production rule in CFG : $A \to \alpha$ (A is single non-terminal).
- ★ Parse tree represents the syntactic structure.
- ★ CFG generates Context Free Languages.

Frequently Asked in RGPV Exams :

- ★ Define CFG and explain with example.
- ★ Draw parse tree for a given string.
- ★ Remove ε-productions or unit productions.
- ★ Differentiate between CFG and Regular Grammar.

All the Best!

3. TOP DOWN PARSING

RGPV Notes Hub

1. Introduction :

Top Down Parsing (also called as Leftmost derivation based parsing) is a method of syntax analysis in which parsing starts from the start symbol of grammar and tries to derive the input string.
It constructs the parse tree from root to leaves.
In this method, the leftmost non-terminal is always expanded first.

2. Basic Idea :

- Start with the start symbol (S).
- If the current string has a non-terminal, replace the leftmost non-terminal using one of its productions.
- Continue the process until the whole input string is matched.
- If no applicable production is frond or the input cannot be matched, report error.

3. General Algorithm (Top Down Parsing) :

1. Initialize the input string w$ (append end marker $).
2. Put start symbol S on the stack.
3. Repeat

(ii) If stack top is a terminal and matches the current input symbol, pop it and move input pointer.

(ii) If stack top is a non-terminal, replace it by the RHS of a production whose LHS is that non-terminal.

(iii) If stack top is $, and input is also $, then Accept.

(iv) If no matching production is found, then Error.

4. End Repeat

4. Example Grammar :

Consider the grammar G :

S → E
E → E + T | T
T → T * F | F
F → (E) | id

Input String :
id + id * id

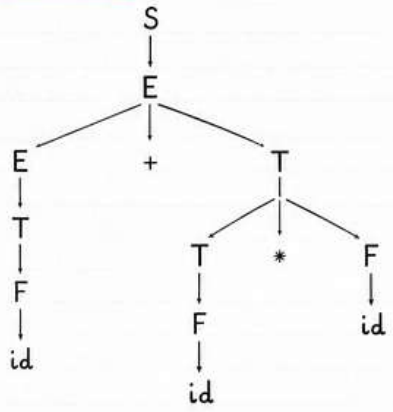
This grammar generates arithmetic expressions.
We will construct the parse tree using Top Down Parsing.

5. Leftmost Derivation (Top Down) for id + id * id :

S ⇒ E	(Using S → E)
⇒ E + T	(Using E → E + T)
⇒ T + T	(Using E → T)
⇒ F + T	(Using T → F)
⇒ id + T	(Using F → id)
⇒ id + T * F	(Using T → T * F)
⇒ id + F * F	(Using T → F)
⇒ id + id * F	(Using F → id)
⇒ id + id * id	(Using F → id) (Input string)

Hence, the string is derived successfully.

6. Parse Tree :



The leaves (from left to right) give the input :
id + id * id

Note :

- Top Down parsing constructs the tree from root to leaves.
- It always expands the leftmost non-terminal first.

7. Stack Implementation (Illustration) :

Step	Stack (Top →)	Input (Remaining)	Action / Production Used
1	S $	id + id * id $	Start
2	E $	id + id * id $	S → E
3	E + T $	id + id * id $	E → E + T
4	T + T $	id + id * id $	E → T
5	F + T $	id + id * id $	T → F
6	id + T $	id + id * id $	F → id
7	+ T $	+ id * id $	Match id, pop
8	T $	id * id $	Match +, pop
9	T * F $	id * id $	T → T * F
10	F * F $	id * id $	T → F
11	id * F $	id * id $	F → id
12	* F $	* id $	Match id, pop
13	F $	id $	Match *, pop
14	id $	id $	F → id
15	$	$	Match id, pop
16	$	$	Accept

8. Advantages :

- Easy to understand.
- Constructs parse tree from top to bottom.
- Useful for recursive descent and predictive parsing.
- Provides early detection of errors.

9. Limitations / Disadvantages :

- May require backtracking if grammar is not suitable.
- Inefficient for certain grammars (left recursion).
- May not always find the correct parse.

10. Applications :

- Used in compilers for syntax analysis.
- Basis for recursive descent and predictive parsers.
- Used in interpreters and scripting language processors.
- Used where input can be processed from left to right.

11. Important Points for Exams :

★ Starts from start symbol and goes down to leaves.
★ Produces leftmost derivation.
★ Parse tree is built from root to leaves.
★ If no production matches, report error.
★ Used in recursive descent and predictive parsing.

4. Recursive Descent Parsing

RGPV Notes Hub

1. Introduction :

Recursive Descent Parsing is a Top Down Parsing technique in which each non-terminal of the grammar is represented by a recursive procedure (function). The parser starts from the start symbol and tries to match the input from left to right. It is simple, intuitive and easy to implement.

2. Basic Idea :

- One procedure for each non-terminal.
- Procedure calls other procedures according to productions.
- Matching of terminals is done with current input symbol.
- If all input symbols are matched and start symbol has been derived, then the string is accepted.

3. Requirements (Conditions) :

For a grammar to be suitable for Recursive Descent Parser :

(i) The grammar must be free from left recursion.

(ii) The grammar must be free from common left factoring.

(iii) It should be predictive (LL(1) grammar preferred).

4. Example Grammar :

Consider the grammar G :

$E \rightarrow TE'$

$E' \rightarrow + T E' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow * F T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Input String :

id + id * id

5. Recursive Descent Procedures :

(i) Procedure for E

```
void E()
{
   T();        // E → T E'
   Ep();
}
```

(ii) Procedure for E'

```
void Ep()     // E' → + T E' | ε
{
  if (lookahead == '+')
   {  match('+');
      T();
      Ep();
   }
  else
    /* ε-production */
}
```

(iii) Procedure for T

```
void T()
{
   F();        // T → F T'
   Tp();
}
```

(iv) Procedure for T'

```
void Tp()
{
  if (lookahead == '*')
   {  match('*');
      F();
      Tp();
   }
  else
    /* ε-production */
}        // T' → * F T' | ε
```

(v) Procedure for F

```
void F()
{
  if (lookahead == '(')
   {  match('(');
      E();
      match(')');
   }
  else if (lookahead == id)
      match(id);
  else
     error();
}          // F → (E) | id
```

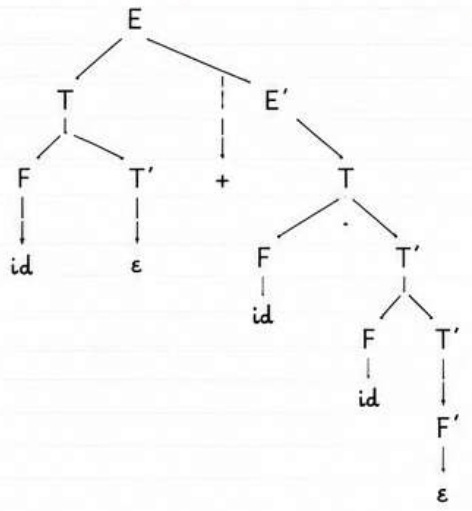
Notes :

- lookahead is the current input symbol.
- match(x) checks if lookahead == x, if yes, it consumes the input and moves to next symbol, else error.

Important Points for Exams :

★ Recursive Descent Parsing is a Top Down Parsing method.
★ One procedure is written for each non-terminal.
★ Grammar must be free from left recursion and left factoring.
★ It works well for LL(1) grammars.
★ Easy to implement but may require backtracking.

6. Parse Tree for id + id * id :



Derivation :

E
⇒ TE'
⇒ FT'E'
⇒ id T'E'
⇒ id ε E'
⇒ id + TE'
⇒ id + FT'E'
⇒ id + id T'E'
⇒ id + id * FT'E'
⇒ id + id * id T'E'
⇒ id + id * id ε E'
⇒ id + id * id ε
⇒ id + id * id

7. Working of Recursive Descent Parser :

Step	Procedure Call	Input (Remaining)	Action
1	E()	id + id * id	Start
2	T()	id + id * id	Call T from E
3	F()	id + id * id	Call F from T
4	match(id)	+ id * id	id matched
5	Tp()	+ id * id	T' (ε as next is +)
6	Ep()	+ id * id	E' called
7	match('+')	id * id	'+' matched
8	T()	id * id	Call T
9	F()	id * id	Call F
10	match(id)	* id	id matched
	Tp()	* id	'*' matched, call F
	F()	id	Call F
	match(id)	$	id matched
	All ε productions	$	Accept

8. Advantages :

- Easy to understand and implement.
- Direct translation of grammar into code.
- Works efficiently for LL(1) grammars.

9. Limitations / Disadvantages :

- Cannot handle left recursive grammars.
- May require backtracking (inefficient).
- Not suitable for all grammars.

10. Applications :

- Used in compilers for syntax analysis of programming languages.
- Used in interpreters.
- Useful for simple language processors.

Summary :

Recursive Descent Parsing is a Top Down Parsing method where each non-terminal is implemented as a procedure. It builds the parse tree from root to leaves and is simple but requires grammar to be suitable (no left recursion, no left factoring).

5. Predictive Parsing
(Top Down Parsing)

RGPV Notes Hub

1. Introduction :

Predictive Parsing is an advanced form of Top Down Parsing. It constructs the parse tree using a parsing table (M) and does not use backtracking.

It is non-recursive, deterministic and uses a single token of lookahead to select the production rule.

2. Requirements (Conditions) :

- ✓ Grammar must be in LL(1) form.
- ✓ No Left Recursion.
- ✓ Left Factoring should be done.
- ✓ For every non-terminal A, A has at most one production for a given lookahead symbol.

3. Components Used :

- **FIRST set** : set of terminals that can appear first in the strings derived from a symbol.
- **FOLLOW set** : set of terminals that can appear immediately to the right of a symbol.
- **Parsing Table** : Table M[A, a] used to select the production A → α.

4. Algorithm (Predictive Parsing) :

1. Construct FIRST and FOLLOW sets.
2. Build the predictive parsing table M.
3. Initialize the stack with $ and start symbol S.
4. Repeat :
   - If stack top is a terminal and matches input → pop both.
   - If stack top is a non-terminal A → use M[A,a] to select production and replace A by RHS in reverse order.
   - If stack top is $ and input is $ → Accept.
   - Else → Error.

5. Example Grammar :

Consider the grammar :

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input String :
id + id * id

6. FIRST and FOLLOW Sets :

FIRST		FOLLOW	
FIRST(E)	= { id, (}	FOLLOW(E)	= {), $ }
FIRST(E')	= { +, ε }	FOLLOW(E')	= {), $ }
FIRST(T)	= { id, (}	FOLLOW(T)	= { +,), $ }
FIRST(T')	= { *, ε }	FOLLOW(T')	= { +,), $ }
FIRST(F)	= { id, (}	FOLLOW(F)	= { *, +,), $ }

Important Points for Exams :

- ★ Predictive Parser is a Non-recursive Top Down Parser.
- ★ Uses FIRST and FOLLOW sets to build the parsing table.
- ★ Works only for LL(1) grammars.
- ★ Predictive Parsing avoids backtracking.

7. Predictive Parsing Table :

Non-Terminal	id	(	)	+	*	$
E	E→TE'	E→TE'	-	-	-	-
E'	-	-	E'→ε	E'→+TE'	-	E'→ε
T	T→FT'	T→FT'	-	-	-	-
T'	-	-	T'→ε	T'→ε	T'→*FT'	T'→ε
F	F→id	F→(E)	-	-	-	-

8. Predictive Parsing Stack Simulation :

Step	Stack	Input	Action
1	$ E	id + id * id $	Start
2	$ E' T	id + id * id $	E → T E'
3	$ E' T' F	id + id * id $	T → F T'
4	$ E' T' id	id + id * id $	F → id
5	$ E' T'	+ id * id $	Match id
6	$ E'	+ id * id $	T' → ε
7	$ E' T +	+ id * id $	E' → + T E'
8	$ E' T	id * id $	Match +
9	$ E' T' F	id * id $	T → F T'
10	$ E' T' id	id * id $	F → id
11	$ E' T'	* id $	Match id
12	$ E' F T' *	* id $	T' → * F T'
13	$ E' T' F	id $	Match *
14	$ E' T' id	id $	F → id
15	$ E' T'	$	Match id
16	$ E' F	$	T' → ε
17	$	$	E' → ε
			Accept ✓

9. Advantages :

- ✓ No backtracking → faster than recursive descent.
- ✓ Deterministic parsing using table lookups.
- ✓ Simple and efficient for LL(1) grammars.
- ✓ Easy to implement in parser generators (e.g., ANTLR).

10. Limitations :

- ⚠ Works only for LL(1) grammars.
- ⚠ Requires elimination of left recursion.
- ⚠ Grammar must be left factored.
- ⚠ Not suitable for all programming languages.

11. Applications :

- Used in compiler design for syntax analysis.
- Used in language processors and interpreters.
- Helpful in generating parse trees and syntax trees.

Summary :

Predictive Parsing is a table-driven, non-recursive Top Down Parsing technique which uses FIRST and FOLLOW sets to select the correct production. It is efficient, deterministic and works for LL(1) grammars, making it an important topic in Compiler Design.

6. Bottom Up Parsing
(Shift-Reduce Parsing)

RGPV Notes Hub

1. Introduction :

Bottom Up Parsing (also called Shift-Reduce Parsing) is a method of syntax analysis in which parsing starts from the input string and tries to reduce it to the start symbol (root).

It constructs the parse tree from leaves to root.

2. Basic Idea :

- Handle (small substring) is reduced to its non-terminal.
- Repeatedly reduce handles until start symbol is obtained.
- If we reach start symbol and input is consumed, string is accepted.
- Works for Rightmost derivation in reverse.

3. Algorithm (Shift-Reduce Parsing) :

① Initialize : Put $ at bottom of stack. Input string with $ at end.

② If next input symbol is a terminal → Shift it onto stack.

③ If the top of stack matches RHS of any production $A \to \beta$, then Reduce β to A.

④ If stack contains only $ and start symbol S and input is $ → Accept.

4. Example Grammar :

Consider the grammar G :

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

Input String :

id + id * id $

5. Rightmost Derivation (Reverse for Bottom Up) :

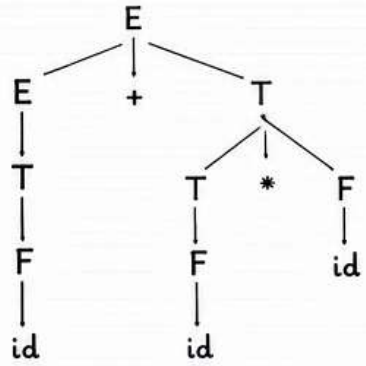
$E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * id \Rightarrow E + F * id$
$\Rightarrow E + id * id \Rightarrow T + id * id \Rightarrow id + id * id$

(Reverse order of reduction)

6. Shift-Reduce Parsing (Stack Simulation) :

Step	Stack	Input	Action
1	$	id + id * id $	Start
2	$ id	+ id * id $	Shift id
3	$ F	+ id * id $	Reduce id → F
4	$ T	+ id * id $	Reduce F → T
5	$ E	+ id * id $	Reduce T → E
6	$ E +	id * id $	Shift +
7	$ E + id	* id $	Shift id
8	$ E + F	* id $	Reduce id → F
9	$ E + T	* id $	Reduce F → T
10	$ E + T *	id $	Shift *
11	$ E + T * id	$	Shift id
12	$ E + T * F	$	Reduce id → F
13	$ E + T	$	Reduce F → T
14	$ E	$	Reduce T → E
15	$ E	$	Accept ✓

7. Parse Tree (for id + id * id) :



Note :

- Bottom Up Parsing builds tree from leaves to root.
- Reductions are performed using handles.

8. Handle in Bottom Up Parsing :

- Handle is a substring that matches RHS of a production and can be reduced to LHS.
- Example in above string : Handles in reverse order are → id, F, T, E.

9. Comparison (Top Down vs Bottom Up) :

Feature	Top Down Parsing	Bottom Up Parsing
Start	Root → Leaves	Leaves → Root
Approach	Expand	Reduce
Derivation	Leftmost derivation	Rightmost derivation (Reverse)
Method	Recursive / Predictive	Shift-Reduce
Backtracking	May require	Not required
Efficiency	Less efficient	More efficient (LR parsers)
Example Parsers	Recursive Descent, Predictive Parser	SLR, LALR, LR Parser

10. Advantages of Bottom Up Parsing :

- ✓ More powerful (can handle larger class of grammars).
- ✓ No left recursion problem.
- ✓ No backtracking required.
- ✓ Used in LR Parsers (SLR, LALR, CLR).
- ✓ More suitable for programming languages.

11. Limitations :

- ✗ Construction of parsing table is complex.
- ✗ Harder to implement compared to top down.
- ✗ Needs more states and memory.

12. Applications :

- ◆ Used in compiler design (most real-world compilers).
- ◆ Used in syntax analysis of programming languages like C, C++, Java.
- ◆ Basis of LR Parser, SLR, LALR, CLR parsing methods.

Summary :

Bottom Up Parsing (Shift-Reduce Parsing) is a method where parsing starts from the input string and reduces it step by step to the start symbol using handles. It constructs parse tree from leaves to root and is more powerful and efficient than top down parsing.

Important Points for Exams :

- ★ Bottom Up Parsing → Shift Reduce Parsing.
- ★ Start = Input string, End = Start symbol.
- ★ Uses Handle & Rightmost derivation in reverse.
- ★ Builds tree from leaves to root.
- ★ No backtracking.

7. Operator Precedence Parsing

RGPV Notes Hub

1. Introduction

Operator Precedence Parsing is a Bottom Up Parsing technique. It is used for a subset of context free grammars called Operator Precedence Grammars. It handles operators and operands based on precedence relations between terminals.

2. Operator Precedence Grammar :

A grammar is said to be Operator Precedence Grammar if it satisfies following conditions :

- It does not have ε-productions.
- It does not have productions of the form $A \to a$.
- For any production $A \to \alpha$, where $\alpha = a_1 a_2 \dots a_n$, a_i and a_{i+1} are not adjacent non-terminals.
- The grammar is free from ambiguity.

3. Precedence Relations :

For any two terminals a and b, one of the following relations may hold :

a	$<\cdot$ b	(a yields precedence to b)
a	= b	(a has equal precedence with b)
a	$>\cdot$ b	(a takes precedence over b)

$: End marker

4. Construction of Operator Precedence Table :

1. List all terminals of the grammar and $.
2. For every production $A \to \alpha$, put precedence relations as per following rules :
   - If a and b are adjacent in α and a is a terminal and b is a terminal then a = b.
   - If a is a terminal, b is a terminal and β is a string (possibly empty) of grammar symbols, and $A \to \alpha a \beta b \gamma$ is a production, then a $<\cdot$ b.
   - If a is a terminal, b is a terminal and β is a string (possibly empty) of grammar symbols, and $A \to \alpha a \beta b \gamma$ is a production, then a $>\cdot$ b.
3. If more than one relation is obtained for a pair (a,b), then the grammar is not Operator Precedence Grammar.

5. Example Grammar :

Consider grammar G :

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

Input String :

id + id * id $

6. Precedence Relations for Grammar :

a	Relation	b	Reason
id	=	id	(both terminals adjacent in F → id)
id	$>\cdot$	+	(F → id is followed by in E → E+T)
id	$>\cdot$	*	(F → id is followed by T → T * F)
id	$>\cdot$	)	(id is followed by) in F → (E))
+	$<\cdot$	id	(+ is followed by id in E → E + T)
+	$<\cdot$	(	(+ is followed by (in F → (E))
+	$>\cdot$	+	(E → E + T gives + $>\cdot$ +)
+	$<\cdot$	*	(+ is followed by * in T → T * F)
*	$<\cdot$	id	(* is followed by id in T → T * F)
*	$<\cdot$	(	(* is followed by (in F → (E))
*	$>\cdot$	*	(T → T * F gives * $>\cdot$ *)
*	$>\cdot$	+	(in T → T * F, * is before + in E → E+T)
(	$<\cdot$	id	((is followed by id in F → (E))
(	$<\cdot$	(	((is followed by (in F → (E))
(	$>\cdot$	)	(F → (E) gives ($>\cdot$))
)	$>\cdot$	+	( ) is before + in E → E + T)
)	$>\cdot$	*	( ) is before * in T → T * F)
)	$>\cdot$	)	( ) is before))

Note

$ $<\cdot$ id , $ $<\cdot$ (, id $>\cdot$ $,) $>\cdot$ $

7. Operator Precedence Table :

	id	+	*	(	)	$
id	=	$>\cdot$	$>\cdot$	–	$>\cdot$	$>\cdot$
+	$<\cdot$	$>\cdot$	$<\cdot$	$<\cdot$	$>\cdot$	$>\cdot$
*	$<\cdot$	$>\cdot$	$>\cdot$	$<\cdot$	$>\cdot$	$>\cdot$
(	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	=	–
)	$>\cdot$	$>\cdot$	$>\cdot$	–	$>\cdot$	$>\cdot$
$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	–	=

$<\cdot$ = yields precedence = = equal precedence $>\cdot$ = takes precedence

8. Parsing (Stack Simulation) for id + id * id $:

Step	Stack	Input	Relation	Action
1	$	id + id * id $	$ $<\cdot$ id	Shift id
2	$ id	+ id * id $	id $>\cdot$ +	Reduce F → id
3	$ F	+ id * id $	F $>\cdot$ +	Reduce T → F
4	$ T	+ id * id $	T < +	Shift +
5	$ T +	id * id $	+ $<\cdot$ id	Shift id
6	$ T + id	* id $	id $>\cdot$ *	Reduce F → id
7	$ T + F	* id $	F < *	Shift *
8	$ T + F *	id $	* $<\cdot$ id	Shift id
9	$ T + F * id	$	id $>\cdot$ $	Reduce F → id
10	$ T + F * F	$	F $>\cdot$ $	Reduce T → T * F
11	$ T + T	$	T $>\cdot$ $	Reduce E → T
12	$ E + T	$	T $>\cdot$ $	Reduce E → E + T
13	$ E	$	E = $	Accept

9. Advantages :

- More powerful than simple Top Down Parsing.
- Handles a class of grammars larger than LL(1) grammars.
- No left recursion problem.
- No backtracking required.
- Suitable for operator rich languages.

10. Limitations :

- Grammar must be Operator Precedence Grammar.
- Not suitable for all context free grammars.
- Table construction can be difficult.
- Not as powerful as LR parsers.

11. Applications :

- Used in compilers for expression parsing.
- Database query parsers.
- Mathematical expression evaluators.
- Languages with well defined operator precedence.

Summary :

Operator Precedence Parsing is a Bottom Up Parsing method that uses precedence relations ($<\cdot$, =, $>\cdot$) between terminals to construct the parse tree. It is efficient and easy for operator dominated languages.

Important Points for Exams :

- ★ Bottom Up Parsing method.
- ★ Uses precedence relations ($<\cdot$, =, $>\cdot$).
- ★ Requires Operator Precedence Grammar.
- ★ Use stack and precedence table.
- ★ No left recursion and no backtracking.
- ★ More efficient than Top Down Parsing.

All the Best!

8. LR Parsers

1. Introduction :

LR Parsers are Bottom Up Parsers that scan the input string from Left to Right constructing a Rightmost derivation in reverse.

L → input is read from Left

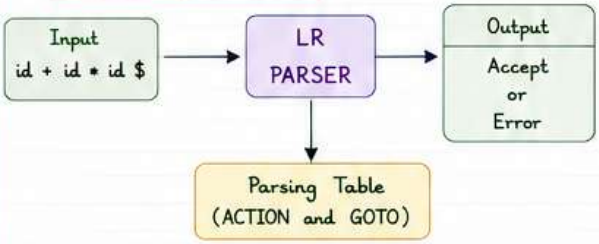
R → Rightmost derivation is produced in reverse.

2. Definition :

An LR Parser is a Shift - Reduce parser that makes parsing decisions based on a small number of input symbols (lookahead).

It uses a stack to store grammar symbols and states.

3. Structure of LR Parser :



4. Components :

- Stack : Stores states and grammar symbols.
- Input : String of tokens with $ (end marker).
- Parsing Table : Consists of ACTION and GOTO tables.
- ACTION Table : Tells whether to Shift, Reduce, Accept or Error.
- GOTO Table : Tells next state after a Reduce.

5. Working of LR Parser :

① Initially push state 0 on stack. Input has $ at end.

② Look at the top state on stack and current input symbol.

③ If ACTION [top, a] = shift s → push a and state s.

④ If ACTION [top, a] = reduce A → β, Pop 2|β| symbols, then push A and state from GOTO [top, A].

⑤ If ACTION [top, a] = accept → Parsing successful.

⑥ If ACTION [top, a] = error → Reject the string.

6. Example Grammar :

Consider the grammar

E → E + T | T
T → T * F | F
F → (E) | id

Input String :

id + id * id $

7. Items (LR(0) Items) :

An item is a production with a dot (.) at some position.

For example for production E → E + T

The possible items are :

- E → . E + T
- E → E . + T
- E → E + . T
- E → E + T .

Note :
Dot shows how much of the RHS has been seen so far.

8. Canonical Collection of LR(0) Items (CLOSURE and GOTO) :

- CLOSURE(I) : If A → α . B β is in I and B → γ is a production, then add B → . γ to I (and its closure) until no more can be added.
- GOTO(I, X) : From state I, on symbol X, move dot over X in all items and take closure.

Example :

I_0 = CLOSURE({E' → . E})

E' → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

id →

GOTO(I_0, id) = I_5

F → id .
T → F .
T → T . * F
E → T .
E → E . + T

Similarly, we construct all states I_0, I_1, I_2, ..., I_n.

9. LR Parsing Table :

- ACTION[i, a] : action for terminal a in state i (shift / reduce / accept / error)
- GOTO[i, A] : state to go when non-terminal A is on stack top in state i

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5	-	-	s4	-	-	1	2	3
1	-	s6	-	-	-	acc	-	-	-
2	-	r2	s7	-	r2	r2	-	-	-
3	-	r4	r4	-	r4	r4	-	-	-
4	s5	-	-	s4	-	-	8	2	3
5	-	r6	r6	-	r6	r6	-	-	-
6	s5	-	-	s4	-	-	-	9	3
7	s5	-	-	s4	-	-	-	-	10

s i → shift and go to state i ri → reduce by production i acc → accept

10. Example Parsing (id + id * id $) :

Step	Stack (States)	Input (Remaining)	Action
1	0	id + id * id $	Shift id (s5)
2	0 5	+ id * id $	Reduce F → id (r6)
3	0 3	+ id * id $	Reduce T → F (r4)
4	0 2	+ id * id $	Reduce E → T (r2)
5	0 1	+ id * id $	Shift + (s6)
6	0 1 6	id * id $	Shift id (s5)
7	0 1 6 5	* id $	Reduce F → id (r6)
8	0 1 6 3	* id $	Reduce T → F (r4)
9	0 1 6 9	* id $	Shift * (s7)
10	0 1 6 9 7	id $	Shift id (s5)
11	0 1 6 6 9 7 5	$	Reduce F → id (r6)
			Reduce T → T * F (r3)
	0 1		Reduce E → E + T (r1)
12	0 1	$	Accept

11. Advantages :

- More powerful than operator precedence parsing.
- Handles a larger class of context free grammars.
- No left recursion problem.
- Efficient and practical (especially LALR(1)).

12. Limitations :

- Parsing table construction is complex.
- More memory requirement.
- Difficult to implement manually compared to LL parsers.

12. Types of LR Parsers :

1. LR(0) Parser - No lookahead.
2. SLR(1) Parser - Uses FOLLOW sets.
3. CLR(1) Parser - Uses full 1-symbol lookahead.
4. LALR(1) Parser - Merged states of CLR for efficiency.

Summary :

LR Parsers are bottom up parsers that perform rightmost derivation in reverse using shift and reduce operations with the help of ACTION and GOTO tables. They are powerful and widely used in real compilers.

9. SLR PARSER (Simple LR Parser)

RGPV Notes Hub

1. Introduction :

SLR Parser (Simple LR Parser) is a Bottom Up Parser that uses LR(0) items for constructing items and FOLLOW sets for deciding REDUCE operations.

2. Features :

- Uses LR(0) items.
- Uses FOLLOW sets to fill reduce actions.
- No lookahead required for shift actions.
- Simpler than CLR and LALR parsers.

3. Construction of SLR Parsing Table :

① Augment the grammar : Add $S' \rightarrow S$

② Construct LR(0) items and Canonical Collection of LR(0) items.

③ Compute FOLLOW set for each non-terminal.

④ Construct ACTION and GOTO tables :

- If item $[A \rightarrow \alpha . a \beta] \in I_i$ and a is terminal
  → ACTION[i, a] = shift j (where goto(I_i, a) = I_j)
- If item $[A \rightarrow \alpha .] \in I_i$ and $A \neq S'$
  → For each $a \in$ FOLLOW(A), ACTION[i, a] = reduce $A \rightarrow \alpha$
- If $[S' \rightarrow S .] \in I_i$ → ACTION[i, $] = accept
- For GOTO table, if goto(I_i, A) = I_j, then GOTO[i, A] = j

4. Grammar Example) :

Consider the grammar G :

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

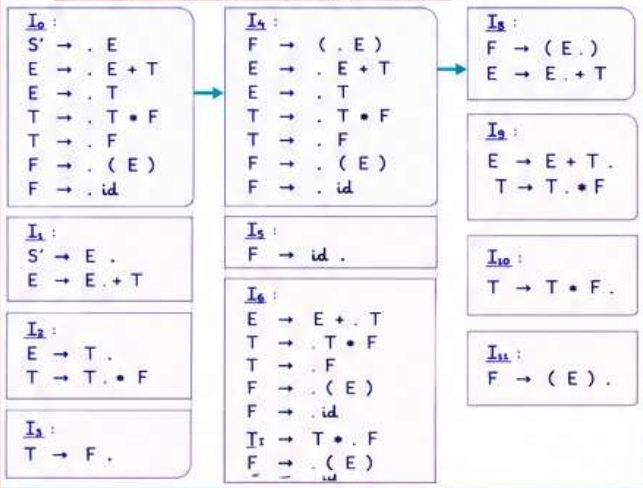
Augmented Grammar :

(0) $S' \rightarrow E$
(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow id$

5. FOLLOW Sets :

- FOLLOW(E) = { +,), $ }
- FOLLOW(T) = { *, +,), $ }
- FOLLOW(F) = { *, +,), $ }

6. Canonical Collection of LR(0) Items :



7. Transitions (GOTO Function) :

From State	On Symbol						
	id	+	*	(	)	E	T
I_0	5	-	-	4	-	1	3
I_1	-	6	-	-	-	-	-
I_2	-	-	7	-	-	-	-
I_3	-	-	-	-	-	-	-
I_4	5	-	-	4	-	8	3
I_5	-	-	-	-	-	-	-
I_6	5	-	-	4	-	9	3
I_7	5	-	-	4	-	-	10
I_8	-	6	-	-	11	-	-
I_9	-	-	7	-	-	-	-
I_{10}	-	-	-	-	-	-	-
I_{11}	-	-	-	-	-	-	-

8. SLR Parsing Table :

State	ACTION (Terminals)						GOTO (Non-terminals)		
	id	+	*	(	)	$	E	T	F
0	s5	-	-	s4	-	-	1	2	3
1	-	s6	-	-	-	acc	-	-	-
2	r2	s7	-	-	r2	r2	-	-	-
3	r4	r4	r4	r4	r4	r4	-	-	-
4	s5	-	-	s4	-	-	8	2	3
5	r6	r6	r6	r6	r6	r6	-	-	-
6	s5	-	-	s4	-	-	-	9	3
7	s5	-	-	s4	-	-	-	-	10
8	-	s6	-	-	s11	-	-	-	-
9	r1	s7	-	-	r1	r1	-	-	-
10	r3	r3	r3	r3	r3	r3	-	-	-
11	r5	r5	r5	r5	r5	r5	-	-	-

r1 : $E \rightarrow E + T$ r2 : $E \rightarrow T$ r3 : $T \rightarrow T * F$
r4 : $T \rightarrow F$ r5 : $F \rightarrow (E)$ r6 : $F \rightarrow id$

9. Example Parsing : Input String : id + id * id $

Step	Stack	Input	Action
1	0	id + id * id $	s5
2	0 5	+ id * id $	r6 (F → id)
3	0 3	+ id * id $	r4 (T → F)
4	0 2	+ id * id $	r2 (E → T)
5	0 1	+ id * id $	s6
6	0 1 6	id * id $	s5
7	0 1 6 5	* id $	r6 (F → id)
8	0 1 6 10	* id $	r3 (T → T * F)
9	0 1 6	* id $	s7
10	0 1 6 7	* id $	s5
11	0 1 6 7 5	$	r6 (F → id)
12	0 1 6 7 10	$	r3 (T → T * F)
13	0 1 6 9	$	r1 (E → E + T)
			acc

10. Advantages :

- ✓ Simple and easy to construct.
- ✓ More efficient than Top Down Parsers.
- ✓ Uses small lookahead (1 symbol).

11. Limitations :

- ✗ Not as powerful as CLR or LALR parsers.
- ✗ Cannot handle all LR grammars, (only SLR grammars).
- ✗ More conflicts compared to LALR.

12. Applications :

- Used in many compiler front-ends.
- Suitable for deterministic context free languages (SLR grammars).
- Practical for medium sized grammars.

Important Points for Exams :

- ★ SLR uses LR(0) items + FOLLOW sets.
- ★ Reduce entries are filled using FOLLOW sets.
- ★ Simpler than CLR and LALR parsers.
- ★ Conflicts may occur for some grammars.

Summary :

SLR Parser is a Simple LR Parser that uses LR(0) items and FOLLOW sets. It performs shift, reduce and accept actions using ACTION and GOTO tables. It is simple but less powerful than CLR and LALR parsers.

10. LALR Parser (Look Ahead LR Parser)

RGPV Notes Hub

1. Introduction :

LALR Parser (Look Ahead LR Parser) is an improvement over CLR Parser. It merges states of CLR(1) items that have the same LR(0) cores. This reduces the size of the parsing table while retaining most of the power of CLR. Hence LALR is widely used in practical compilers.

2. Features :

- Uses LR(1) lookahead information.
- Merges states with same LR(0) cores.
- Table size is much smaller than CLR.
- More powerful than SLR.
- Handles more context free grammars.

3. Construction of LALR Parsing Table :

① Construct the Canonical Collection of LR(1) items.
② Group states having the same LR(0) core.
③ Merge their lookahead sets.
④ Build ACTION and GOTO tables using the merged states.

4. Grammar Example :

Consider the grammar G :

E → E + T | T
T → T * F | F
F → (E) | id

Augmented Grammar :

(0) S' → E
(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

5. LR(0) Cores that will be Merged :

States to be Merged	Reason
I_1 and I_4	Same LR(0) core [E → E • + T]
I_2 and I_7	Same LR(0) core [T → T • • F]
I_3 and I_6	Same LR(0) core [T → F •]

In merging, we take union of lookahead sets of items with same core.

6. Merged LALR(1) Item Sets :

I_0 :
S' → • E , {$}
E → • E + T , {$,+}
E → • T , {$,+}
T → • T * F , {$,+}
T → • F , {$,+,*,)}
F → • (E) , {$,+,*,)}
F → • id , {$,+,*,)}

$I_1 = I_4$:
E → E • + T , {$,+}
E → E • + T , {+,)}

$I_2 = I_7$:
T → T • * F , {$,+}
T → T • * F , {+,)}

$I_3 = I_6$:
T → F • , {$,+,*,)}
T → F • , {+,)}
F → id • , {$,+,*,)}
F → id • , {+,)}

I_5 :
E → E + • T , {$,+}
T → • T * F , {$,+}
T → • F , {$,+,*,)}
F → • (E) , {$,+,*,)}
F → • id , {$,+,*,)}

I_6 :
T → T * • F , {$,+}
F → • (E) , {$,+,*,)}
F → • id , {$,+,*,)}

I_7 :
F → (• E) , {$,+,*,)}
E → • E + T , {)}
E → • T , {)}
T → • T * F , {)}
T → • F , {),*,}
F → • (E) , {),*,}
F → • id , {),*,}

Important Points for Exams :

★ LALR is obtained by merging CLR(1) states with same LR(0) core.
★ Lookahead sets are merged using UNION operation.
★ LALR table is more compact than CLR but more powerful than SLR.
★ Most practical parser generators (YACC, BISON, CUP) use LALR.

7. LALR(1) Parsing Table :

State	ACTION (Terminals)						GOTO (Non-terminals)		
	id	+	*	(	)	$	E	T	F
0	s3	-	-	s4	-	-	1	2	3
1	-	s5	-	-	acc	-	-	-	-
2	-	-	s6	-	-	-	-	-	-
3	-	r4	r4	r4	r4	r4	-	-	-
4	s3	-	-	s4	-	-	7	2	3
5	s3	-	-	s4	-	-	-	-	-
6	s3	-	s6	s4	-	-	-	-	-
7	-	s5	-	-	s8	-	-	-	-
8	-	r2	r2	r2	r2	r2	-	-	-

s i → shift to state i r j → reduce by production j acc → accept

8. Example Parsing : Input String : id + id * id $

Step	Stack	Input	Action
1	0	id + id * id $	s3
2	0 3	+ id * id $	r4 (F → id)
3	0 2	+ id * id $	r2 (T → F)
4	0 1	+ id * id $	s5
5	0 1 5	+ id * id $	s3
6	0 1 5 3	* id $	r4 (F → id)
7	0 1 5 2	* id $	r2 (T → F)
8	0 1 5 2	* id $	s6
9	0 1 5 2 6	+ id $	s3
10	0 1 5 2 6 3	$	r4 (F → id)
11	0 1 5 2 6 3	$	r3 (T → T * F)
12	0 1 5 2	$	r1 (E → E + T)
13	0 1	$	acc

9. Comparison :

Feature	SLR	CLR(1)	LALR(1)
Uses Lookahead	No	Yes (exact)	Yes (merged)
Number of States	Least	Very Large	Medium
Table Size	Smallest	Largest	Smaller than CLR
Power	Least	Most Powerful	Very Powerful
Practical Use	Rare	Rare	Widely Used

10. Advantages of LALR Parser :

✓ More powerful than SLR Parser.
✓ Smaller parsing table than CLR Parser.
✓ Can parse most programming language grammars.
✓ Used in tools like YACC, BISON, CUP.

11. Limitations :

✖ More complex to construct than SLR.
✖ Conflicts (if any) need grammar modification.

12. Applications :

◆ Used in compiler construction for syntax analysis.
◆ Used in YACC, Bison, CUP parser generators.
◆ Used in programming languages (C, C++, Java, etc.).
◆ Used in real time language processing systems.

Summary :

LALR Parser is a practical parser that uses LR(1) lookahead but merges states with same LR(0) core. It provides a good trade-off between power and efficiency, which is why it is the most widely used bottom up parsing technique.

11. Parser Generation

RGPV Notes Hub

Parser Generation is the process of constructing a parser automatically from a given grammar using a parser generator tool. It avoids manual construction of parsing tables and coding.

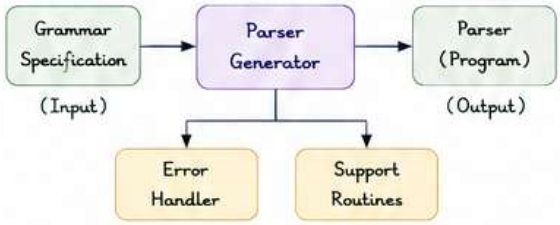
1. Introduction :

A parser generator takes as input a context free grammar and produces as output a program (parser) which can analyze strings of that grammar and report whether they belong to the language or not.

2. Need of Parser Generation :

- Manual construction of parsers is time consuming and error prone.
- Parser generators produce efficient and reliable parsers.
- Easy to modify grammar and regenerate parser.
- Used in modern compiler construction tools.

3. Components of a Parser Generator System :



4. Working of Parser Generator :

① The user writes grammar in specification language of parser generator.

② Grammar is analyzed and necessary tables / code structures are constructed.

③ Parser generator produces source code of parser in a target programming language (e.g., C, C++).

④ The generated code is compiled and linked with support routines to form the final parser.

⑤ The parser reads input strings and performs parsing to check syntactic correctness.

5. Steps in Parser Generation :

Step		Description
1	Grammar Input	Specify grammar using tool's language.
2	Grammar Analysis	Check for syntax errors in grammar.
3	Automaton / Table Construction	Construct LR automaton or tables.
4	Code Generation	Generate source code for parser.
5	Compilation	Compile generated code with support routines to create executable parser.
6	Execution	Run parser on input strings.

6. Example : (Using YACC / BISON)

- Input : Grammar written in .y file (YACC specification).
- Output : C source code (y.tab.c) + header file (y.tab.h).
- Steps : Lexical analyzer (.l) + YACC file (.y) → yacc / bison → C code → gcc → Parser executable.

Example Commands (Unix)

```
yacc -d calculator.y   // generates y.tab.c and y.tab.h
lex calculator.l       // generates lex.yy.c
gcc y.tab.c lex.yy.c -ll -o parser.exe    // executable
```

Important Points for Exams :

★ Parser Generation automates parser construction.
★ Uses tools like LEX, YACC, BISON, ANTLR, JavaCC.
★ YACC / BISON → Bottom Up (LALR(1)).
★ ANTLR / JavaCC → Top Down (LL, Recursive Descent).
★ Reduces manual effort and improves reliability.

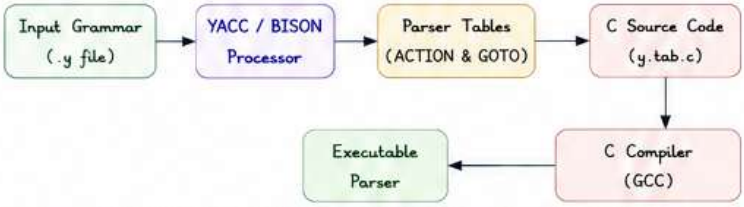
7. Parser Generator Tools :

Tool	Type	Input	Output	Language
LEX	Lexical Analyzer Generator	Regular Expression Specification (.l)	C code for Lexer	C
YACC	Parser Generator (LALR(1))	Grammar Specification (.y)	C code for Parser	C
BISON	Enhanced YACC (FREE)	Grammar Specification (.y)	C code for Parser	C
ANTLR	Parser Generator (Top Down)	Grammar Specification (.g)	Parser in Java, C#, Python, etc.	Multi
JavaCC	Recursive Descent Parser Generator	Grammar Specification (.jj)	Parser in Java	Java

8. Classification of Parser Generators :

- Based on Parsing Technique :
  - Top Down Parser Generators (e.g., ANTLR, JavaCC)
  - Bottom Up Parser Generators (e.g., YACC, BISON)
- Based on Algorithm :
  - LR-based (SLR, LALR, CLR)
  - LL-based (Recursive Descent, Predictive)

9. YACC / BISON Processing Phases :



Note : It also generates header file (y.tab.h) for token definitions.

10. Example Grammar and Generated Parser :

Grammar G :

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

Tokens :

id = identifier
\+ = plus
* = multiply
(= open bracket
) = close bracket
$ = end marker

Parser Generator (YACC) will :

- Construct LR(1) items and LALR(1) states.
- Build ACTION and GOTO tables.
- Generate C code for parsing using these tables.

11. Advantages :

✓ Automatic parser construction.
✓ Reliable and efficient.
✓ Easy maintenance (change grammar and regenerate).
✓ Well tested tools with error handling.

12. Limitations :

✗ Limited to grammar class supported by the tool.
✗ Learning the tool's specification language.
✗ Debugging grammar can be difficult.

Summary :

Parser Generation uses a formal grammar as input and produces an efficient parser program automatically using tools like LEX, YACC, BISON, ANTLR, etc. It is the practical approach used in real world compilers.

12. Syntax Directed Definitions (SDD)

Syntax Directed Definitions (SDD) are formal notations to associate semantic rules or actions with the productions of a context free grammar.

1. Introduction :

An SDD is a context free grammar along with a set of attributes for its grammar symbols and a set of semantic rules (semantic functions) which compute the values of these attributes.

2. Components of SDD :

① Grammar : Set of productions.

② Attributes : Each grammar symbol is associated with a set of attributes.

③ Semantic Rules : Set of rules (functions or actions) that specify the value of each attribute.

④ Evaluation Mechanism : Order in which attributes are evaluated.

3. Types of Attributes :

- Synthesized Attribute : Value computed from the attributes of children (in the parse tree) and passed upward.
- Inherited Attribute : Value computed from the attributes of parent or siblings and passed downward or across.

Type	Flow
Synthesized	From children → Parent (Bottom - Up)
Inherited	From Parent / Siblings → Child (Top - Down / Across)

4. Example Grammar :

Consider grammar G :

E → E + T | T

T → T * F | F

F → (E) | id

Assume Attribute :
val → stores the numeric value of the expression.

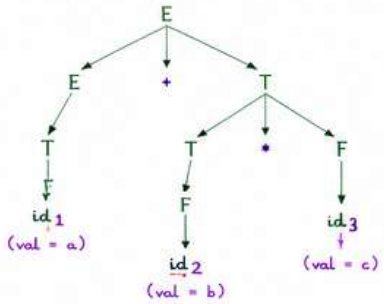
5. SDD for the Grammar :

We associate synthesized attribute val with each non-terminal (E, T, F).

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow id$	$F.val = id.lexval$

6. Parse Tree and Attribute Evaluation :

Example String : id + id * id



Evaluation (Bottom - Up)

F → id : F.val = a

F → id : F.val = b

F → id : F.val = c

T → F : T.val = b

T → T * F : T.val = b * c

E → T : E.val = a

E → E + T : E.val = a + b * c

Final Result : E.val = a + b * c

Important Points for Exams :

★ SDD = Grammar + Attributes + Semantic Rules.

★ Synthesized → from children to parent.

★ Inherited → from parent/siblings to child.

★ L-Attributed SDDs can be evaluated in one left to right scan (Top Down).

★ Dependency graph must be acyclic.

7. Inherited Attributes Example :

Consider grammar for declarations :

D → T L

T → int | float

L → L , id | id

Attributes :
T.type → Inherited by L and id
id.name → synthesized (lexeme)

Semantic Rules :

Production	Semantic Rule
D → T L	L.inh = T.type
T → int	T.type = integer
T → float	T.type = real
$L \rightarrow L_1$, id	$L_1.inh = L.inh$ id.inh = L.inh
L → id	id.inh = L.inh
id	id.name = lexeme(id) (copy the lexeme)

Meaning :
The type specified in T is passed to all identifiers in L.

Example :
int a , b , c ;
All a, b, c will get type = integer

8. SDD Representation :

An SDD is written as :

SDD = (G, A, R)

Where,

G → Context Free Grammar

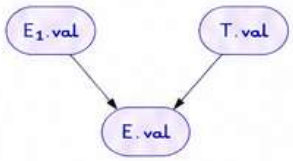
A → Set of Attributes

R → Set of Semantic Rules

9. Dependency Graph :

A dependency graph shows the relative order of evaluation of attributes.

Example for $E \rightarrow E_1 + T$: $E.val = E_1.val + T.val$



Rules for Dependency Graph :

- If A depends on B, draw edge B → A.
- The graph must not have cycles for evaluation.

10. Type of SDD based on Dependency :

- S-SDD (Syntax Directed SDD) : Can be evaluated using any parsing order (no dependency on parse tree structure).
- L-Attributed SDD : All inherited attributes of a symbol depend only on attributes of its parent and left siblings.
- L-Attributed SDDs can be evaluated during Top Down Parsing.

11. Advantages :

✓ Provides a formal way to specify semantics.

✓ Supports code generation and translation.

✓ Easy to implement with parser.

✓ Separates syntax from semantics.

12. Applications :

◆ Used in syntax directed translation.

◆ Used in type checking, intermediate code generation.

◆ Used in compiler construction for semantic analysis.

◆ Used in attribute grammar based tools.

Summary :

Syntax Directed Definitions associate attributes with grammar symbols and semantic rules with productions. Attributes carry semantic information and rules compute their values. SDDs are essential for semantic analysis and code generation in compilers.

13. Syntax Directed Translation (SDT)

Syntax Directed Translation (SDT) is a method of translating a source program from one language (source language) to another language (target language) using the syntax of the source program and semantic actions.

1. Introduction :

SDT uses a context free grammar along with semantic rules (actions) to produce output in a target language. It performs translation during parsing by attaching semantic actions to grammar productions.

2. SDT vs SDD :

Feature	SDD	SDT
Output	Computes values of attributes	Produces translation (target code)
When actions are executed	After parsing (may be)	During parsing
Purpose	Semantic analysis	Translation
Example	Type checking, value evaluation	Intermediate code, machine code, assembly code

3. Components of SDT :

① Context Free Grammar : The syntactic structure.
② Semantic Actions : Code fragments associated with grammar productions.
③ Attributes (optional) : To hold intermediate information.
④ Translation Mechanism : Specifies order of execution of actions.

4. Forms of SDT :

- Can be implemented using
  - > Syntax Directed Definitions (SDD)
  - > or directly using actions.
- Two types based on implementation strategy
  - > Syntax Directed Translation Scheme
  - > Translation Scheme with Control.

5. Types of Actions in SDT :

- Computation Actions : Perform some computation.
- Output Actions : Generate output (e.g., print, emit).
- Control Actions : Alter the flow or control of translation.

6. Example Grammar :

Consider the grammar G :

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Goal : Generate postfix expression (Reverse Polish Notation).

7. SDT for Postfix Translation (Using Actions) :

Grammar with Semantic Actions :

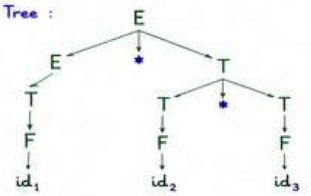
Production	Semantic Action (postfix)
$E \rightarrow E_1 + T$	{ print ("+") }
$E \rightarrow T$	{ }
$T \rightarrow T_1 * F$	{ print ("*") }
$T \rightarrow F$	{ }
$F \rightarrow (E)$	{ }
$F \rightarrow id$	{ print (id.lexeme) }

Note : The action is executed after the RHS is completely parsed.

8. Example Translation :

Input Infix Expression : id + id * id

Parse Tree :



Applying semantic actions (postfix) :
1. id_1
2. id_2
3. id_3
4. *
5. +

Postfix Output : $id_1\ id_2\ id_3\ *\ +$

Important Points for Exams :
- ★ SDT = CFG + Semantic Actions for translation.
- ★ Actions are executed during parsing.
- ★ Two forms : Translation Scheme and Scheme with Control.
- ★ SDT is used to generate output (target code).
- ★ Example outputs : Postfix, Three Address Code, etc.

9. Translation Schemes in SDT :

Scheme	Description
Syntax Directed Translation Scheme	Uses SDD like mechanism. Actions are attached to productions and executed during parsing in a defined order (e.g., Leftmost or Rightmost derivation).
Translation Scheme with Control	Uses explicit control statements (if-else, loops, switch, etc.) to control the flow of actions. More flexible and suitable for complex translations.

10. Implementation of SDT :

- Attach actions to grammar productions.
- During parsing, when a production is applied, execute its action.
- Actions may use attributes and other information collected during parsing.
- Output is generated in target language (intermediate code, machine code, etc.).

11. Example : Generate Three Address Code (TAC)

For expression : a = b + c * d

Three Address Code (TAC) using SDT :

Step	Production Applied	Semantic Action (TAC)
1	$F \rightarrow id$ (b)	$t_1 = b$
2	$F \rightarrow id$ (c)	$t_2 = c$
3	$F \rightarrow id$ (d)	$t_3 = d$
4	$T \rightarrow T_1 * F$	$t_4 = t_2 * t_3$
5	$E \rightarrow E_1 + T$	$t_5 = t_1 + t_4$
6	$S \rightarrow id = E$	$a = t_5$

Generated TAC :
$t_1 = b$
$t_2 = c$
$t_3 = d$
$t_4 = t_2 * t_3$
$t_5 = t_1 + t_4$
$a = t_5$

Note :
- Temporary variables $t_1, t_2, \ldots$ are used.
- TAC is simple and easy for code generation.

12. Advantages of SDT :

- ✓ Produces output during parsing (one pass).
- ✓ Efficient and requires less storage.
- ✓ Easily implemented using parser generator tools.
- ✓ Suitable for generating intermediate code.
- ✓ More practical for real compilers.

13. Limitations of SDT :

- ✗ Difficult to debug because actions are embedded in grammar.
- ✗ Not suitable if translation requires global information.
- ✗ Complex translations may need many actions and attributes.

14. Applications of SDT :

- ◆ Used in compilers to generate intermediate code.
- ◆ Code generation for target machines.
- ◆ Used in interpreters and translators.
- ◆ Used in SQL to relational algebra translation.
- ◆ Used in syntax directed editors.

Summary :

Syntax Directed Translation (SDT) uses a context free grammar and semantic actions to translate a source program to a target program during parsing. It is efficient, practical and widely used in compiler design to generate intermediate code and other outputs.

14. S-Attributed Definition

An S-Attributed Definition is a special type of SDD in which all attributes are Synthesized attributes only (i.e., values are computed from children to parent). It can be evaluated in a bottom-up manner (postorder of parse tree).

1. Introduction :

In an S-Attributed Definition, a non-terminal has only synthesized attributes. The value of a non-terminal is computed from the attributes of its children. Therefore, it can be evaluated in any Bottom-Up traversal of the parse tree.

2. Key Points :

- Only synthesized attributes are used.
- The value of a node depends only on its children.
- It is a restricted and simple form of SDD.
- Easy to implement in Bottom-Up parsers (LR parsers).
- It cannot handle inherited attributes.

3. Definition :

An S-Attributed Definition is an SDD in which every attribute in the SDD is synthesized.

4. Example Grammar :

Consider the grammar for arithmetic expressions :

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

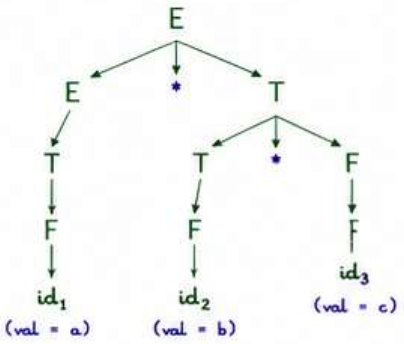
$F \rightarrow (E) \mid id$

Goal : Compute the value of an arithmetic expression.

5. S-Attributed Definition for the Grammar :

Production	Semantic Rule (Synthesized Attribute)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow id$	$F.val = id.lexval$

6. Parse Tree Example : Input : id + id * id



Evaluation (Postorder) :

1. $id_1.val = a$
2. $F.val = a$
3. $T.val = a$
4. $id_2.val = b$
5. $F.val = b$
6. $T.val = b$
7. $id_3.val = c$
8. $F.val = c$
9. $T.val = b * c$
10. $E.val = a + (b * c)$

Final Result : $E.val = a + b * c$

Important Points for Exams :

- S-Attributed Definition uses only synthesized attributes.
- Evaluation can be done in Bottom-Up (postorder).
- Suitable for implementation with LR parsers.
- Cannot handle inherited attributes.

7. Why Bottom-Up Evaluation is Possible ?

- Each attribute is computed from the attributes of its children.
- All required values are available before computing the parent.
- Hence, evaluation can occur during reduction in a Bottom-Up parser.

8. Comparison with General SDD :

Feature	General SDD	S-Attributed Definition
Attribute Types	Synthesized + Inherited	Only Synthesized
Dependency	May depend on parent, siblings, etc.	Depends only on children
Evaluation Order	May need Top-Down and/or Bottom-Up	Only Bottom-Up (Postorder)
Suitable For	Complex definitions	Simple definitions
Parser Support	Not suitable for LR	Suitable for LR parsers

9. Example: Three Address Code for Expression a = b + c * d

Step	Production Applied	Action (Semantic Rule)	Code Generated
1	$F \rightarrow id$ (b)	$F.val = b$	$t_1 = b$
2	$F \rightarrow id$ (c)	$F.val = c$	$t_2 = c$
3	$F \rightarrow id$ (d)	$F.val = d$	$t_3 = d$
4	$T \rightarrow T_1 * F$	$T.val = T_1 * F$	$t_4 = t_2 * t_3$
5	$E \rightarrow E_1 + T$	$E.val = E_1 + T$	$t_5 = t_1 + t_4$
6	$S \rightarrow id = E$	$S.code = E.val$	$a = t_5$

Generated Three Address Code :

$t_1 = b$

$t_2 = c$

$t_3 = d$

$t_4 = t_2 * t_3$

$t_5 = t_1 + t_4$

$a = t_5$

Note : All attributes are synthesized. Evaluation is done in Bottom-Up (Postorder).

10. Advantages :

- ✓ Simple and easy to implement.
- ✓ Can be evaluated during reductions in Bottom-Up parsers.
- ✓ No need for global data or symbol tables (in simple cases).
- ✓ Efficient for expression evaluation, type checking, etc.

11. Limitations :

- ✗ Cannot use inherited attributes.
- ✗ Not sufficient for definitions needing sibling/parent information.
- ✗ Less expressive compared to general SDD.

12. Applications :

- ◆ Arithmetic expression evaluation.
- ◆ Type checking of expressions.
- ◆ Intermediate code generation (e.g., Three Address Code).
- ◆ Simple declarations and assignments.
- ◆ Used in LR parser based compiler construction.

Summary :

S-Attributed Definition is an SDD with only synthesized attributes. It allows Bottom-Up evaluation (postorder) and is suitable for implementation in LR parsers. It is simple, efficient and widely used for expression evaluation and intermediate code generation.

15. L-Attributed Definition

An L-Attributed Definition is an SDD in which each inherited attribute of a symbol A_i in a production depends only on:
- Attributes of the head non-terminal A, and
- Attributes of the symbols to the left of A_i in the body of the production.

It can be evaluated in a single Left-to-Right (Top-Down) traversal of parse tree.

1. Introduction :

L-Attributed Definitions are more general than S-Attributed Definitions. They allow limited use of inherited attributes but still permit evaluation in one pass (Left to Right).

2. Key Points :

- Uses both synthesized and inherited attributes.
- Inherited attributes of a symbol A_i depend only on:
  - Attributes of the head (LHS) non-terminal, and
  - Attributes of the symbols to the left of A_i.
- Can be evaluated in a Left-to-Right, Depth-First (Top-Down) manner.
- Suitable for predictive parsers.

3. Definition :

An SDD is L-Attributed if for every production

$A \rightarrow X_1\ X_2\ \ldots\ X_n$

and for every $1 \le i \le n$,

any inherited attribute of X_i depends only on :

(i) Attributes of A, and

(ii) Attributes of $X_1, X_2, \ldots, X_{i-1}$.

4. Comparison with S-Attributed Definition :

Feature	S-Attributed	L-Attributed
Attributes Used	Only synthesized	Synthesized + Inherited
Dependency	No dependency on siblings or parent	Inherited depends on left siblings and parent
Evaluation Order	Postorder (Bottom-Up)	Left-to-Right (Top-Down)
Suitable For	Simple definitions	More general, still simple
Parser Support	Bottom-Up (LR)	Top-Down (Predictive)

5. Example Grammar :

Consider the grammar for declarations :

$D \rightarrow T\ L$

$T \rightarrow int \mid float$

$L \rightarrow L_1\ ,\ id \mid id$

Goal :
Pass the type from T to all identifiers in L.

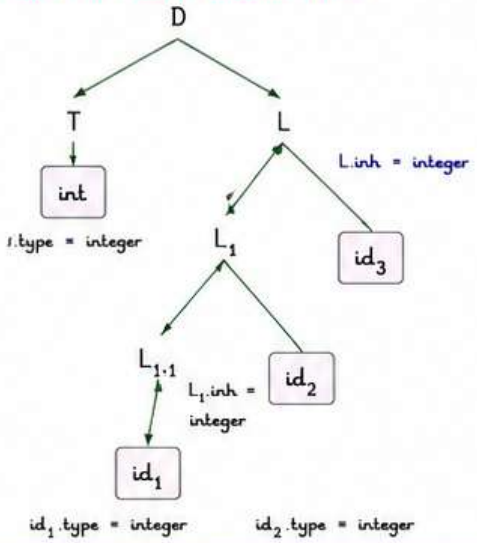
6. L-Attributed Definition (Semantic Rules) :

Production	Semantic Rules
$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow int$	T.type = integer
$T \rightarrow float$	T.type = real
$L \rightarrow L_1\ ,\ id$	L_1.inh = L.inh id.type = L.inh
$L \rightarrow id$	id.type = L.inh

Meaning :
The type computed in T is passed to L (inherited attribute L.inh). That value is then used for every id in L.

Important Points for Exams :
- ★ L-Attributed SDT allows inherited attributes.
- ★ Inherited attribute of A_i depends only on A and left siblings.
- ★ Evaluation can be done in Left-to-Right (Top-Down).
- ★ Suitable for Predictive (LL) parsers.
- ★ More general than S-Attributed but less general than unrestricted SDD.

7. Parse Tree and Attribute Flow :



Evaluation Order (Left to Right) :
1. T.type = integer
2. L.inh = integer
3. L_1.inh = integer
4. id_1.type = integer
5. id_2.type = integer
6. id_3.type = integer

8. Example : Type Checking

For declarations : int a, b, c;

All identifiers a, b, c get type = integer.

9. Implementation of L-Attributed Definition :

- Implemented using Syntax Directed Translation Scheme.
- Translate during parsing in a single Left-to-Right pass.
- Each semantic rule is embedded in parser actions.
- Do not need explicit attribute tables.

10. L-Attributed SDT Example (Postfix Translation)

Production	Semantic Rule (Action)	Postfix Output
$E \rightarrow E_1 + T$	E.val = E_1.val + T.val print("+")	E_1 T +
$E \rightarrow T$	E.val = T.val	T
$T \rightarrow T_1 * F$	T.val = T_1.val * F.val print("*")	T_1 F *
$T \rightarrow F$	T.val = F.val	F
$F \rightarrow (\ E\ )$	F.val = E.val	E
$F \rightarrow id$	F.val = id.lexval print(id.lexval)	id

Note : Actions are executed Left to Right during parsing.

11. Advantages :

- ✓ Can handle inherited attributes.
- ✓ Evaluation in one pass (Top-Down).
- ✓ Suitable for LL(1) parsers.
- ✓ More expressive than S-Attributed definitions.
- ✓ Easy to implement in syntax directed compilers.

12. Limitations :

- ✗ Inherited attribute cannot depend on right siblings.
- ✗ Not suitable for all SDDs (e.g., Left recursion).
- ✗ Some definitions require Bottom-Up evaluation.

Summary :
L-Attributed Definition is a restricted form of SDD where inherited attribute of a symbol depends only on the head non-terminal and its left siblings. It can be evaluated in a Left-to-Right (Top-Down) traversal and is suitable for predictive parsers.

16. Syntax Trees

RGPV Notes Hub

A Syntax Tree (or Parse Tree) is a hierarchical tree representation that shows the syntactic structure of a string according to the grammar.

1. Introduction :

A syntax tree represents the derivation of a string in a context free grammar. It shows how the start symbol is expanded to produce the string.

2. Key Points :

- The root of the tree is the start symbol.
- Interior (non-leaf) nodes are non-terminals.
- Leaf nodes are terminals (or ε).
- The children of a node are arranged from left to right as in the production.
- Reading the leaves from left to right gives the input string.

3. Example Grammar :

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

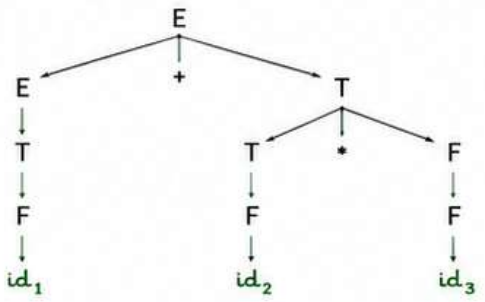
$F \rightarrow (E) \mid id$

Goal :
Generate the syntax tree for the string
id + id * id

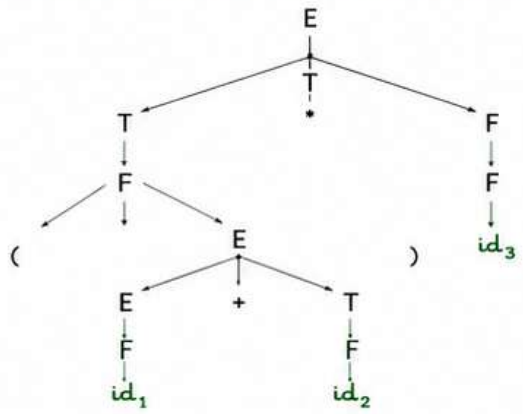
4. Construction of Syntax Tree :

① Start with the start symbol as the root.
② Apply productions to expand the nodes.
③ Expand non-terminals until only terminals remain.
④ Maintain the left to right order of symbols.

5. Syntax Tree for id + id * id :



6. Another Example ((id + id) * id) :



7. Rules / Properties :

- Each interior node has as many children as there are symbols in the right-hand side of the production.
- The yield (left to right leaves) is the input string.
- The tree shows the hierarchical structure of derivation.
- For ambiguous grammars, more than one syntax tree may exist for the same string.

8. Leftmost and Rightmost Derivation in Tree :

Derivation Type	Construction Order	Tree Behavior
Leftmost	Always expand the leftmost non-terminal first	Build tree by expanding from left to right
Rightmost	Always expand the rightmost non-terminal first	Build tree by expanding from right to left

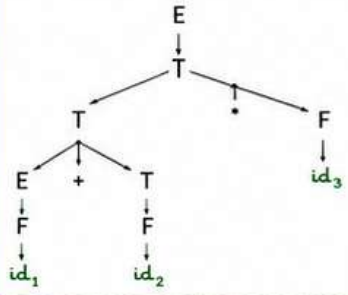
9. Uses of Syntax Trees :

- Used in syntax analysis (parsing) to check grammatical correctness.
- Helps to understand the hierarchical structure of expressions.
- Useful in code generation and translation.
- Basis for constructing abstract syntax trees and intermediate code.

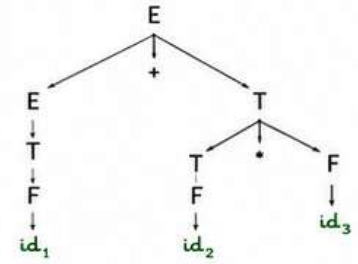
10. Ambiguity in Syntax Trees :

Example String : id + id * id

Tree 1 : (id + id) * id



Tree 2 : id + (id * id)



Since two different trees are possible, the grammar is ambiguous.

11. Summary Table :

Aspect	Description
Definition	A tree that represents the syntactic structure of a string.
Root	Start symbol of the grammar.
Interior Nodes	Non-terminals.
Leaf Nodes	Terminals or ε.
Order	Left to right order of symbols is maintained.
Purpose	Used in parsing, translation and code generation.

12. Important Points for Exams :

★ Every parse tree corresponds to a derivation (leftmost / rightmost).
★ The yield of the leaves is always the input string.
★ Different trees for same string ⇒ Ambiguity.
★ Syntax tree ≠ Abstract Syntax Tree (AST).

Advantages :

✓ Clear representation of structure.
✓ Easy to verify correctness of parsing.
✓ Useful for translation and semantic analysis.

Limitations :

✗ Large trees for long inputs.
✗ Not suitable directly for optimization.
✗ Ambiguous grammars give multiple trees.