Unit 5

Programming Language Grammars-

Context Free Grammar-

Context-Free Grammar (CFG) is a set of rules that define how you can construct sentences or strings in a language. It's often used in formal language theory and programming languages to describe the structure of sentences.

Formal Definition-

A Grammar G has a 4 components such that-

G = (V , T , P , S)

where-

V = Variables/non-terminal symbols
T = terminal symbols
P = production rules
S = Start symbol

Terminals:

- That terminates the string and cannot be replaced further. Terminal symbols are those which are the constituents of the sentence generated using a grammar.
- Terminal symbols are denoted by using small case letters such as a, b, c etc.

Digits: 0, 1, 2, ..., 9
Operators: + (plus), * (multiply)

Non-terminals:

- Non-Terminal symbols are those which take part in the generation of the sentence but are not part of it.
- Non-Terminal are replaced by terminals.

- Non-Terminal symbols are denoted by using capital letters such as A, B, C etc on LHS.

S: Start symbol: production rules start from start symbole.

Now, let's define the Production rules (P): we can have combination of terminals and non terminals.

$E \rightarrow E + E \mid E * E \mid id$
or
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow id$

Terminals- +, *, id
Non Terminals- V = {E}

Q. 1- check the string id+id*id belongs to the grammar $E \rightarrow E + E \mid E * E \mid id$ or not (form a parse tree).

Or derive the string id+id*id from the given grammar (start with the start symbol on LHS. Repeatedly replace the non terminals by production rules until we are only left with the terminals in a string.)

Q2- derive an expression with single digits having either * or – between them.
$E \rightarrow D * D \mid D - D \mid D$
$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
Based on the above production rules, check whether 5*6 or 8-7 belongs to the grammar or not (draw a parse tree).

Left Most Derivation and Right Most Derivation(Derivation is a sequence of production rules)

Leftmost and rightmost derivations are two ways to derive a given string of symbols from a grammar in the context of compiler design. These derivations help in understanding how a parser processes a source code and generates a parse tree.
While deriving the strings we need to decide-

- Which non terminals to replace
- Select a production rule by which non terminals will be replaced

Left Most Derivation

Start from start symbol. I/P is scanned and is replaced with the production rules from left to right.

Example- S→ S+S |S-S |a |b |c

I/P - a-b+c (derive this string on the basis of given production rules).

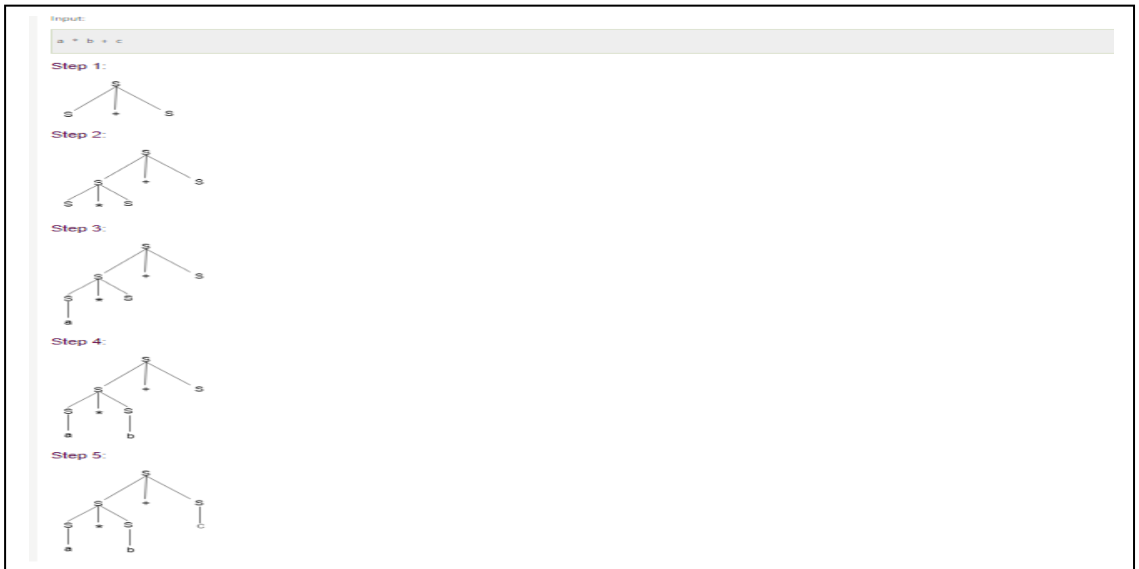
S→ S + S
S→ S – S + S
S→ a – S + S
S→a – b + S
S→a – b + c

Right Most Derivation

Start from start symbol. I/P is scanned and is replaced with the production rules from right to left (refer to the previous example).

S→ S - S
S→ S - S + S
S→ S – S + c
S→ S – b + c
S→ a – b + c

Q.3 T → T + T | T * T
T → a |b |c
Derive an input string a*b+c using parse tree.



Ambiguous Grammar-

For a given string, a grammar is said to ambiguous if it produces more than one-

- Parse tree
- Or derivation tree
- Or syntax tree (leaf nodes are operands and intermediate nodes are operators)
- Or leftmost derivation
- Or rightmost derivation

Example-

Consider the following grammar-
$E \longrightarrow E + E \mid E \times E \mid id$

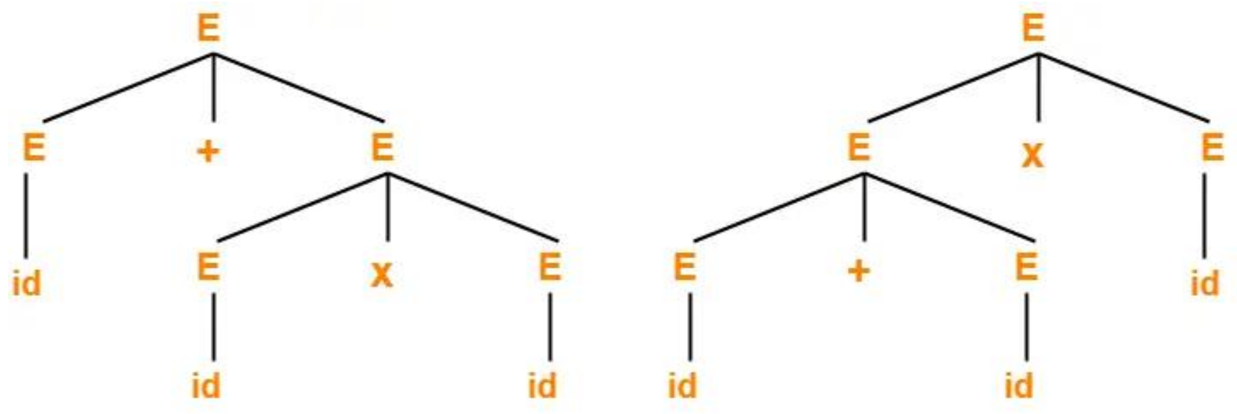
This grammar is an example of ambiguous grammar.

Reason 1-

Let us consider a string w generated by the grammar-

w = id + id x id

Now, let us draw the parse trees for this string w.



Parse Tree-01

Parse Tree-02

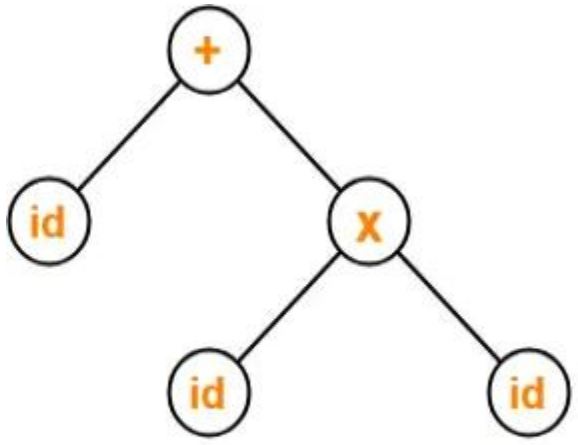
Since two parse trees exist for string w, therefore the grammar is ambiguous.

Reason-02:

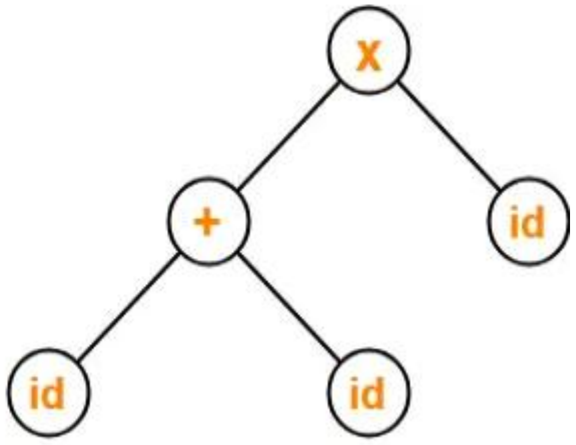
Let us consider a string w generated by the grammar-

w = id + id x id

Now, let us draw the syntax trees for this string w.



Syntax Tree-01



Syntax Tree-02

Since two syntax trees exist for string w, therefore the grammar is ambiguous.

Reason-03:

Let us consider a string w generated by the grammar-

w = id + id x id

Now, let us write the leftmost derivations for this string w.

$$\begin{aligned} E &\rightarrow \mathbf{E} + E \\ &\rightarrow id + \mathbf{E} \\ &\rightarrow id + \mathbf{E} \times E \\ &\rightarrow id + id \times \mathbf{E} \\ &\rightarrow id + id \times id \end{aligned}$$

Leftmost Derivation-01

$$\begin{aligned} E &\rightarrow \mathbf{E} \times E \\ &\rightarrow \mathbf{E} + E \times E \\ &\rightarrow id + \mathbf{E} \times E \\ &\rightarrow id + id \times \mathbf{E} \\ &\rightarrow id + id \times id \end{aligned}$$

Leftmost Derivation-02

Since two leftmost derivations exist for string w, therefore the grammar is ambiguous.

Reason-04:

Let us consider a string w generated by the grammar-

w = id + id x id

Now, let us write the rightmost derivations for this string w.

$$
\begin{aligned}
E &\rightarrow E + \mathbf{E} \\
&\rightarrow E + E \times \mathbf{E} \\
&\rightarrow E + \mathbf{E} \times id \\
&\rightarrow \mathbf{E} + id \times id \\
&\rightarrow id + id \times id
\end{aligned}
$$

Rightmost Derivation-01

$$
\begin{aligned}
E &\rightarrow E \times \mathbf{E} \\
&\rightarrow \mathbf{E} \times id \\
&\rightarrow E + \mathbf{E} \times id \\
&\rightarrow \mathbf{E} + id \times id \\
&\rightarrow id + id \times id
\end{aligned}
$$

Rightmost Derivation-02

Since two rightmost derivations exist for string w, therefore the grammar is ambiguous.

2. Unambiguous Grammar

A grammar is said to unambiguous if for every string generated by it, it produces exactly one-

- Parse tree
- Or derivation tree
- Or syntax tree
- Or leftmost derivation
- Or rightmost derivation

Example-

Consider the following grammar-

E → E + T | T

T → T x F | F

F → id

Unambiguous Grammar

This grammar is an example of unambiguous grammar.

Q.1 The grammar G, which is defined as

S → aS | Sa | a

String- aa, check whether the grammar is ambiguous or not.

Sol.-

- Using the production S → aS :

S => aS => aa

- Using the production S → Sa twice:

S => Sa => aS => aa

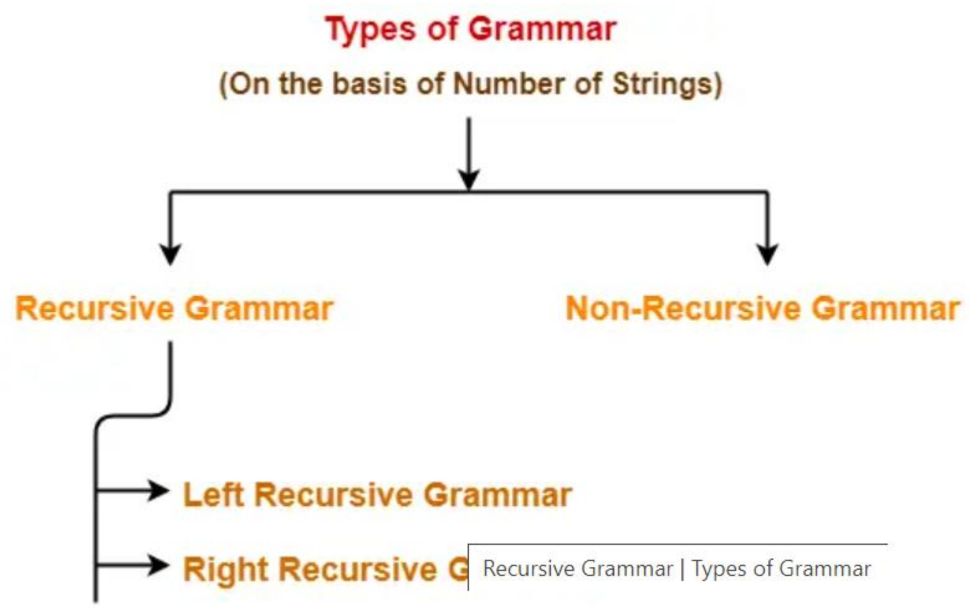
There are nore than one LMD.So it is an ambiguous grammar.

Q.2- E → E +E | E-E | id

I/P String- id+id-id. check whether the grammar is ambiguous or not.

Types of Grammar

On the basis of number of strings, grammars are classified as-



1. Recursive Grammar
2. Non-Recursive Grammar

1. Recursive Grammar-

A grammar is said to be recursive if it contains at least one production that has the same variable at both its LHS and RHS.

OR

A grammar is said to be recursive if and only if it generates infinite number of strings.

Ex.- S→Sa

S→b

{b,ba,…..infinite}

A recursive grammar may be either-

Left recursive grammar

Right recursive grammar

Left Recursive Grammar-

A recursive grammar is said to be left recursive if the leftmost variable of RHS is same as variable of LHS.

OR

A recursive grammar is said to be left recursive if it has Left Recursion.

Example-

$S \rightarrow Sa / b$

Right Recursive Grammar-

A recursive grammar is said to be right recursive if the rightmost variable of RHS is same as variable of LHS.

OR

A recursive grammar is said to be right recursive if it has right recursion.

Ex- $S \rightarrow aS \mid b$

Non-Recursive Grammar-

A grammar is said to be non-recursive if it contains no production that has the same variable at both its LHS and RHS.

OR

A grammar is said to be non-recursive if and only if it generates finite number of strings.

Note- A non-recursive grammar has neither left recursion nor right recursion.

Ex- S → aA / bB

A → a / b

B → c / d

The language generated from this grammar is L = { aa , ab , bc , bd }

Since the grammar generates finite number of strings, therefore it is a non-recursive grammar.

Conversion of Ambiguous to Unambiguous Grammar

Example- G:E→E + E | id

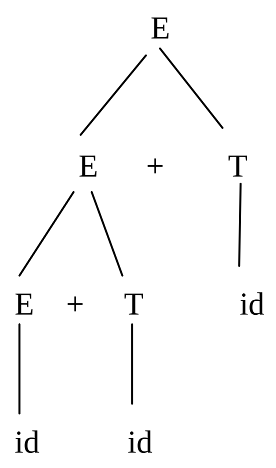
Where w=id + id +id

Soln-

E→E + T | T

T → id

Parse Tree-



Left Factoring (Grammar with Common Prefixes-)

Note- If RHS of more than one production starts with the same symbol, and then such a grammar is called as Grammar with Common Prefixes.

Left factoring is a technique used in compiler design to resolve ambiguities in a grammar, making it easier for a parser to understand and process the programming language's syntax. Here's a simple explanation of left factoring:

Imagine you have a grammar for a programming language, and there's a rule like this:

$A \rightarrow \alpha\beta \mid \alpha\gamma$

In this rule, 'A' is a non-terminal symbol, and 'α', 'β', and 'γ' are sequences of symbols (terminals or non-terminals). When you encounter a string that matches this rule, it could be ambiguous because both alternatives ('αβ' and 'αγ') start with the same 'α'. Left factoring helps remove this ambiguity.

To left factor this rule, you create a new non-terminal symbol, let's call it 'A1', and rewrite the rule like this:

$A \rightarrow \alpha A1$

$A1 \rightarrow \beta \mid \gamma$

Now, the 'A' rule always starts with 'α', and the alternatives ('β' and 'γ') are moved to a new non-terminal 'A1'. This way, the parser can easily decide which alternative to choose based on the input string.

Example-

$A \rightarrow \alpha\beta1 \mid \alpha\beta2 \mid \alpha\beta3$

After left factoring- $A \rightarrow \alpha A'$

$A' \rightarrow \beta1 \mid \beta2 \mid \beta3$

- This kind of grammar creates a problematic situation for Top down parsers.
- Top down parsers cannot decide which production must be chosen to parse the string in hand.

To remove this confusion, we use left factoring-

Note- Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

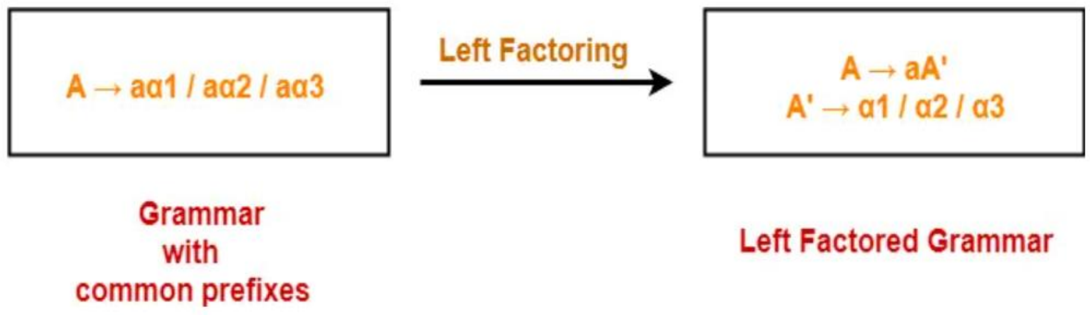
How?

In left factoring-

- We make one production for each common prefixes.
- The common prefix may be a terminal or a non-terminal or a combination of both.
- Rest of the derivation is added by new productions.

The grammar obtained after the process of left factoring is called as Left Factored Grammar.

Example-



PRACTICE PROBLEMS BASED ON LEFT FACTORING-

Problem-01: do left factoring in the following grammar-

S → iEtS | iEtSeS | a
E → b

Solution-

The left factored grammar is-

S → iEtSS’ | a

S’ → eS | ∈

E → b
Here, the symbol "∈" represents the empty string or epsilon, which is a string containing no symbols at all. It's used to indicate that a production rule can generate an empty string as part of its derivation.

If you do not include "∈" in the production rule, and you only have:

$S' \rightarrow eS$

Then it implies that S' can only be replaced by "eS," and it cannot be replaced by an empty string.
In this case, the grammar would not allow the generation of an empty string as part of its language.

Including "∈" in the production rule gives you the flexibility to generate strings that can potentially be empty, which may be necessary in some contexts to describe the language accurately.

Problem-02:

Do left factoring in the following grammar-

$S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$

Solution-

Step-01:

$S \rightarrow bSS' \mid a$

$S' \rightarrow SaaS \mid SaSb \mid b$

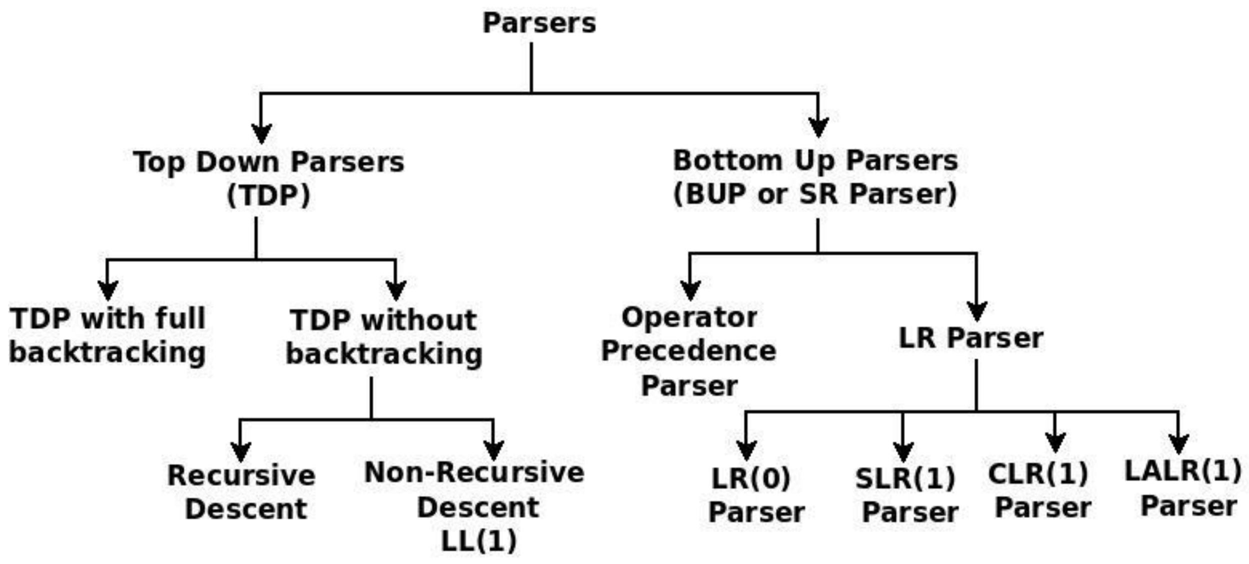
Again, this is a grammar with common prefixes.

Step-02:

$S \rightarrow bSS' \mid a$
$S' \rightarrow SaA \mid b$
$A \rightarrow aS \mid Sb$

This is a left factored grammar.

Different Types of Parsers



- In Top Down Parser, we start with a Starting Non-Terminal , we use the grammar productions and try to reach to the string.
- Bottom Up Parsing (Shift Reduce Parser) is opposite to the Top Down Parsing. Here we start with String. We reduce it by using grammar productions and try to reach to the Start symbol.
- Top Down Parser doest not work for left recursive grammar.
- Top Down Parser doesn’t allow non deterministic grammar.

Difference between Top Down Parser and Bottom up Parser

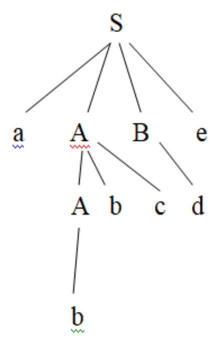
		Top Down Parser	Bottom up Parser
Parameters	Ambiguous	No	No(except operator precedence parser)
	Unambiguous	Yes	Yes
	Left Recursion	No	Yes
	Right Recursion	Yes	Yes
	Non Deterministic	No	Yes
	Deterministic	Yes	Yes

Top Down Parser uses Left Most Derivation (LMD).

Example-
S→ aABe
A→ Abc | b
B → d
I/p string- abbcde

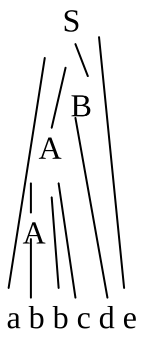
Solution-
S→ aABe
→ aAbcBe
→ abbcBe
→ abbcde

Parse Tree-



At every point, we have to decide that what is the next production we should use (i.e A is to be relaced by Abc or b).

Bottom up Parser- we decide that when to reduce the terminal (reverse of RMD).
Let’s consider the previous example-
Example-
S→ aABe
A→ Abc | b
B → d
I/p string- abbcde



LL(1) Parser-

In compiler design and formal grammar theory, LL(1) (Left-to-Right, Leftmost derivation, 1 lookahead symbol) is a class of grammars and parsers. LL(1) grammars are suitable for top-down parsing, which means parsing starts from the top of the parse tree and moves down to the leaves.

To understand LL(1) grammars, it's essential to know about two important functions associated with LL(1) parsing: the "First" function and the "Follow" function.

First(α) is a set of terminal symbols that begin in strings derived from α.

Example-

Consider the production rule-

A → abc | def | ghi

Then, we have-

First(A) = { a , d , g }

Rules For Calculating First Function-

Rule-01:
For a production rule X → ∈,
First(X) = { ∈ }

Example- Let G:A → aB | bC | ε
The FIRST function for A would be:
FIRST(A) = {a, b, ε}

Rule-02:

For any terminal symbol ‘a’,
First(a) = { a }

Rule-03:

For a production rule $X \rightarrow Y1Y2Y3$

Calculating First(X):

- If $\in \notin$ First(Y1), then First(X) = First(Y1)

- FIRST(Y1) is a set of possible starting terminals for a symbol Y1.
- If this set (FIRST(Y1)) doesn't contain the empty string ε, it means Y1 cannot directly produce an empty string as its first symbol.
- In such a case, if we are trying to find the possible starting terminals for another symbol X, we can simply say that the possible starting terminals for X are the same as those for Y1.

In essence, if Y1 cannot start with an empty string, then X also cannot start with an empty string, so their FIRST sets are the same.

- If $\in \in$ First(Y1), then First(X) = { First(Y1) – $\in$ } $\cup$ First(Y2Y3)

The statement "If ε (empty string) is in the FIRST set of Y1, then the FIRST set of X is a combination of two things:

- It includes the FIRST set of Y1 without the empty string ε.
- It also includes the FIRST set of the string formed by concatenating Y2 and Y3."

In simpler terms:

If Y1 can produce an empty string as its first symbol (i.e., ε is in FIRST(Y1)), then when we're trying to find the possible starting terminals for another symbol X, we start with the possible starting terminals of Y1 but remove the empty string ε from that set.
Additionally, we consider the possible starting terminals for the string formed by putting together Y2 and Y3.
This rule helps in LL(1) grammar analysis and parsing by guiding us on which production rule to choose based on the FIRST sets of symbols while handling ε-productions (productions that can derive the empty string).

Follow Function-

Rules For Calculating Follow Function-

Rule-01:

- For the start symbol S, place \$ in Follow(S)

Rule-02:

For any production rule A → αB

- Follow(B) = Follow(A)

Rule-03:

For any production rule A → αBβ,

- If ∈ ∉ First(β), then Follow(B) = First(β)
- If ∈ ∈ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

Important Notes-
Note 1:

- ∈ may appear in the first function of a non-terminal.
- ∈ will never appear in the follow function of a non-terminal.

Note-02:

- Before calculating the first and follow functions, eliminate Left Recursion from the grammar, if present.

Note-03:

- We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

Example-	First()	Follow()
G:S→ABCD	{b}	{$}
A→b	{b}	{c} i.e first(BCD)
B→c	{c}	{d}
C→d	{d}	{e}
D→e	{e}	{$} i.e follow(D)=follow(A)

Example 2-

	First	Follow
E→ T E'	{id, (}	{ $,) }
E' → + T E' | ε	{+, ε }	{$,) }
T → FT'	{id, (}	{ +, $,) }
T' → * F T' | ε	{*, ε }	{ +, $,) }
F → id | (E)	{id, (}	{ *, +, $)}

// Note- E→ T E' first of E means first of T E' which is T, T → FT'. First of T is first of FT' which is F and first of F is { id, (}.
// in follow, for starting symbol E, follow will be started by $. Then we will check in RHS that whether E exists i.e. F → id | (E). so follow of E will be closed parenthesis.
// in 2nd production, E' → + T E' | ε, for follow of E', we check RHS. In 1st production, E' exists but follow of E' is nothing. So it will be follow of E' = follow of E i.e. now, check any other production if E' exists or not.
//now check RHS for T for follow of T, in 1st production follow of T is first of E'. first if E' is + and ε but we don't not consider ε in follow.
first of E' is + in 2nd production. But its also ε. So if we replace E' by ε in production 1 then follow of T is first of ε which is nothing but first of E i.e. id, (. Add these two symbols.
// T' → * F T' | ε. Find Follow of T'. check T' RHS in all productions. In 3rd production T' exists. Follow of T' will be follow of T i.e. +, $,). Now in 4th production, T' exists too. Here, follow of T' is follow of T which is already written.
// F → id | (E). Check productions for F. In 3rd production, F exists. Follow of F is first of T'. in 4th production first of T' is * so write it. But here is also ε. So if we replace ε in production 3 then follow of F will be first of ε which follow of T i.e.

LL(1) Parsing Table

Consider the previous example-

Example 2-

	First	Follow
E→ T E'	{id, (}	{ $,) }
E' → + T E' | ε	{+, ε }	{$,) }
T → FT'	{id, (}	{ +, $,) }
T' → * F T' | ε	{*, ε }	{ +, $,) }
F → id | (E)	{id, (}	{ *, +, $)

	id	+	*	(	)	$
E	E→ T E'			E→ T E'		
E'		E' →+T E'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → * F T'		T' → ε	T' → ε
F	F → id			F→(E)		

Row- non terminals
Columns- terminals
$ represents the end of string. Here in table we will not take ε. Now, we will see that how to fill this table.

// E→ T E'. first of E is first of T E' i.e. T. now go to T i.e. production no. 3. Now, see fist of T is id and open parenthesis. So in LL1 parse table, put this production for the these terminals i.e. id, (.
// E' → + T E' | ε can be written as-
E' → + T E' // we have already entered in the table.
E' → ε . we will replace it by follow of E' i.e. {$,) }
// now 3rd production, T → FT'. first of F T' is F i.e. {id, (}. So put 3rd production in the table accordingly.
// production 4, T' → * F T' | ε. First of T' is *. So put it in the table accordingly. Now T' → ε. We will put his in follow of T' i.e. { +, $,) }. So put this production in the table accordingly.
// 5th production F → id | (E). first of F is id. Put accordingly in the table. Now, F→(E). first of F is (. Put in the table accordingly.

Conclusion-

for each production, we check first of RHS. If we have ε in RHS then replace it by follow of LHS.
LR grammar and ND grammar can not be used for LL1 parsing.
First we need to convert LR and ND grammar to RR and D. grammar. Though, there may be certain cases where we cannot use LL1 parsing for RR and deterministic grammar.