GRAMMARS

It is basically a set of rules (or productions) that defines the valid structure of particular Language.

A grammer consists of 4 components.

G(V, T, P, S)

G → Grammers
V → Variables / Non terminals
T → Terminals
P → Productions
S → Start Symbols

(i) Set of terminals :- i.e that terminate they are not replaced by any other thing further

(ii) Set of Non-terminals :- values / variables are replaced by terminals.

(iii) Set of productions :- on LHS → Non terminal followed by arrow & on RHS.
We can have T and / or V or combo of both.

(iv) Start symbol :- One of the non-terminals is designated as the start symbol from where the production begins.

Eg :-
E → E + E
E → E * E
E → E ↑ E
E → id

V → {E}
T → {+, *, ↑, id}
P → {4}
S → {E}

Left-Most Derivation :- [LMD]

Derivation is a sequence of production rules. It is used to get the i/p string through these production rules.

We have to decide

→ which Non-terminal to replace
→ production rule by which the non-terminal will be replaced.

two options for this

LMD → i/p scanned & replaced production rule from Left → Right

RMD → i/p scanned & replaced production rule from Right → Left

Eg:

$S \to S+S$
$S \to S-S$
$S \to a|b|c$

input $w = a-b+c$

LMD =

$S \to \boxed{S}+S$
$S \to \boxed{S-S}+S$
$S \to \boxed{a}-\boxed{S}+S$
$S \to a-b+\boxed{S}$
$S \to a-b+\boxed{c}$

Right-Most Derivation :- [RMD] :-

In RMD i/p is scanned & replaced with the production rule from Right → Left.

Eg: $S \to S+S/S-S/a/b/c$

input $w = a-b+c$

RMD will be

$S \to S-\boxed{S}$
$S \to S-\boxed{S+S}$
$S \to S-\boxed{S+c}$
$S \to \boxed{S}-b+c$
$S \to a-b+c$

PARSE TREE:- It is a typical depiction of how the start symbol of a grammer derives a string in the language.

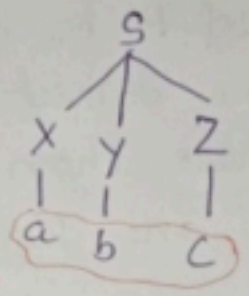
OR

It is a graphical representation of symbol that can be terminals or Non-terminals.

Properties:-

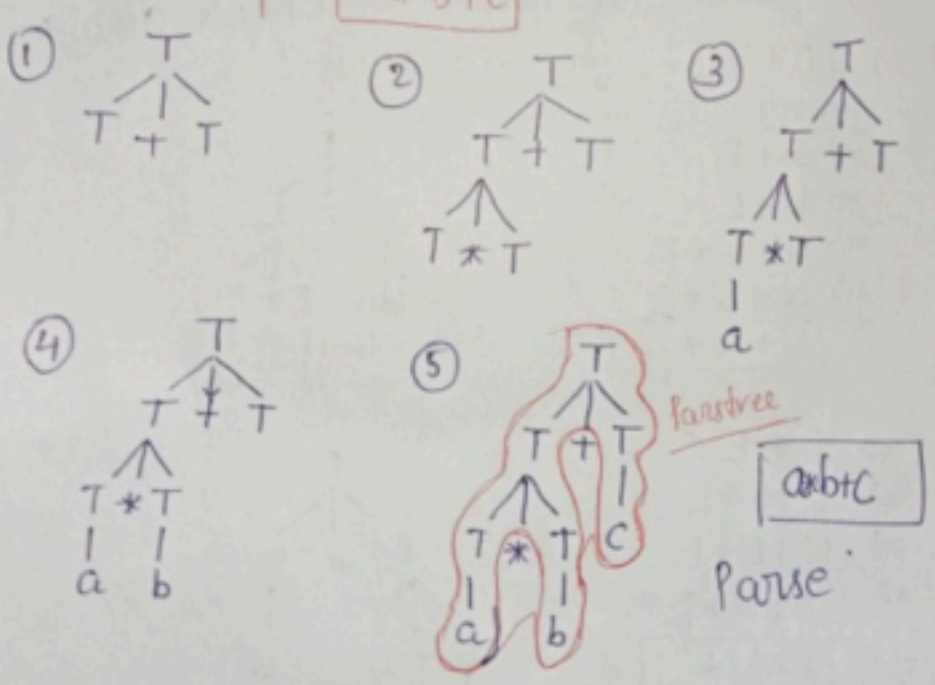
1) Root is always the start symbol
2) All leaf nodes are terminals
3) All interior nodes → Nonterminals

Eg:- $S \to XYZ$
$X \to a$
$Y \to b$
$Z \to c|d$

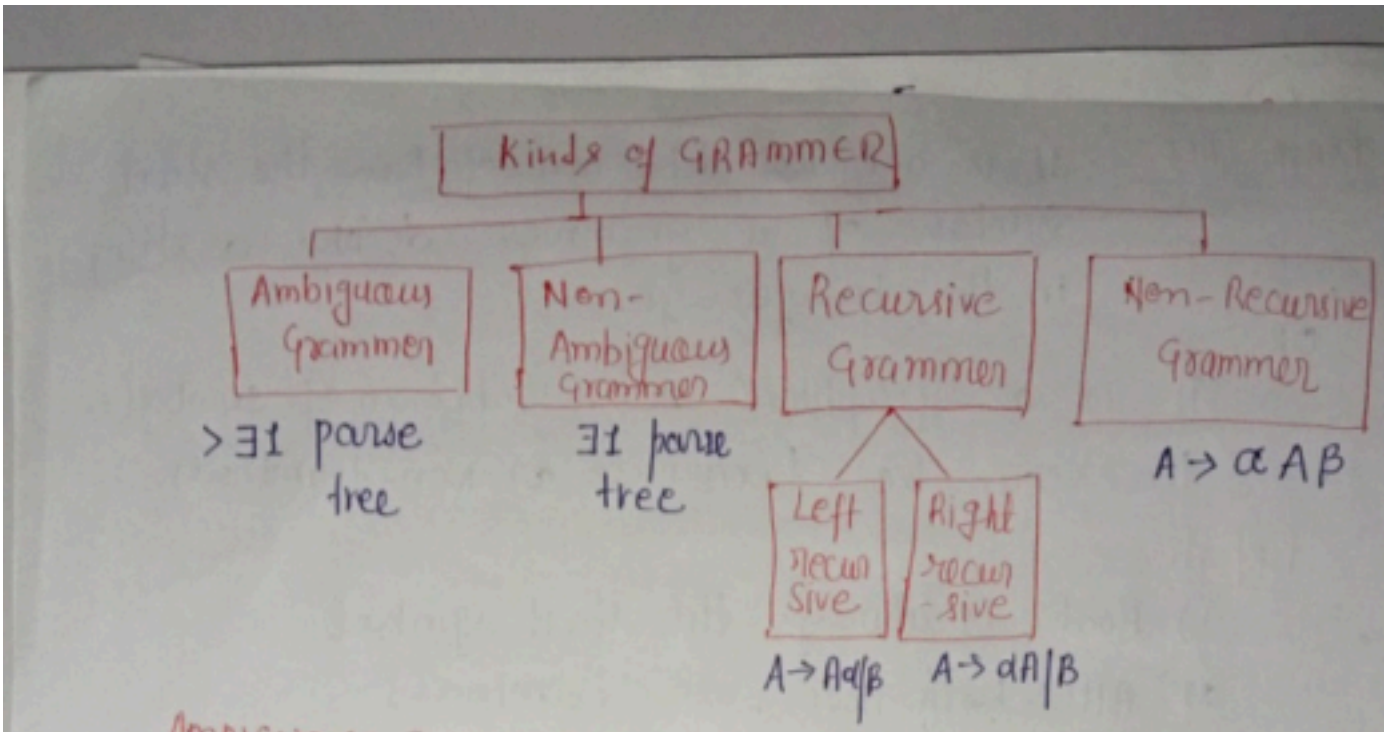


Eg:- $T \to T+T \mid T*T \mid a \mid b \mid c$

Input a*b+c



Parse



AMBIGUOUS GRAMMER :-

A CFG is said to be ambiguous if there exist more than one derivation trees for the given i/p string or more than one Left most Derivation & more than one Right most Derivation.

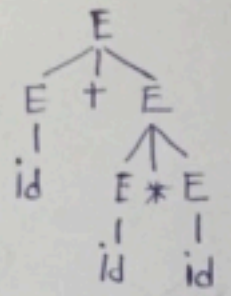
There is no standard Method to check ambiguity.

Eg:-1 $E \to E+E \mid E*E \mid id$ $\omega = id+id*id$

MM SMP

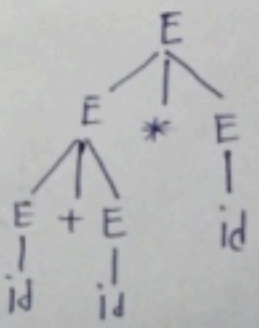
LMD1

$E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow \boxed{id+id*id}$



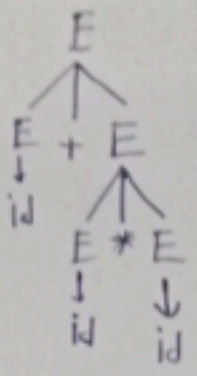
LMD2

$E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow \boxed{id+id*id}$



RMD1

$E \Rightarrow E+E$
$\Rightarrow E+E*E$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow \boxed{id+id*id}$

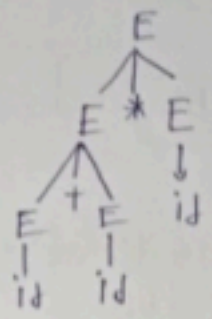


Imp

If a Gramm
Both Left &
Right recu
it will be a
Ambigious
Grammer

RMD2

$E \Rightarrow E*E$
$\Rightarrow E*id$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow \boxed{id+id*id}$

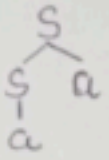


Eg: 2:-

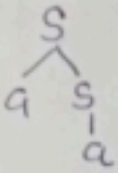
$S \rightarrow aS \mid Sa \mid a$

$\boxed{w = aa}$

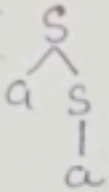
LMD1 $\Rightarrow$ $S \rightarrow aS$
$\rightarrow aa$



LMD2 $\Rightarrow$ $S \rightarrow Sa$
$\rightarrow aa$

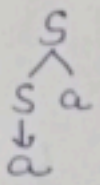


RMD1 $\Rightarrow$ $S \rightarrow aS$
$\rightarrow aa$



RMD2

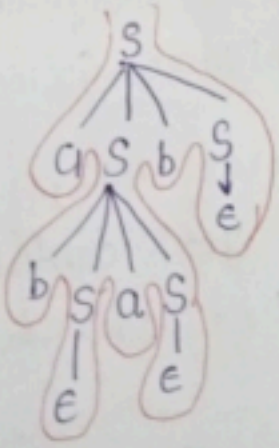
$S \rightarrow Sa$
$\rightarrow aa$

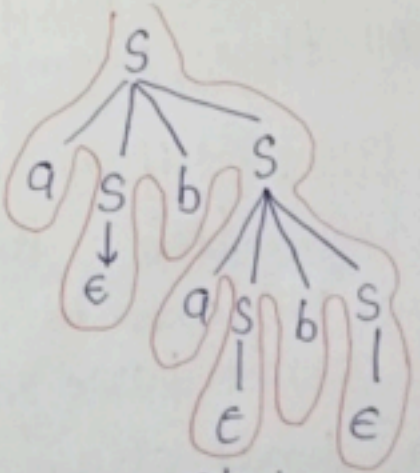


Eg: 3:-

$S \rightarrow aSbS \mid bSaS \mid \epsilon$

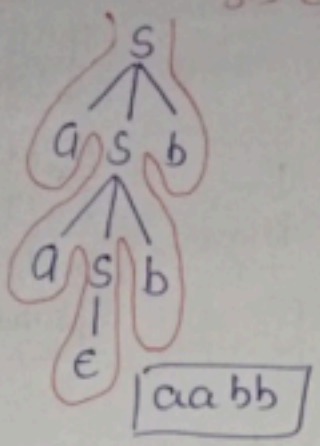
$\boxed{w = abab}$

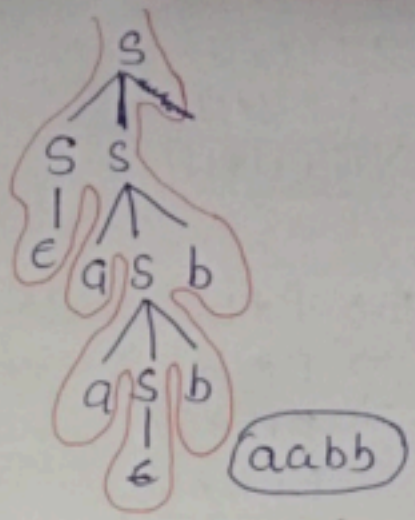




Eg: 4 $S \to aSb \mid SS$
$S \to \epsilon$

$w = aabb$

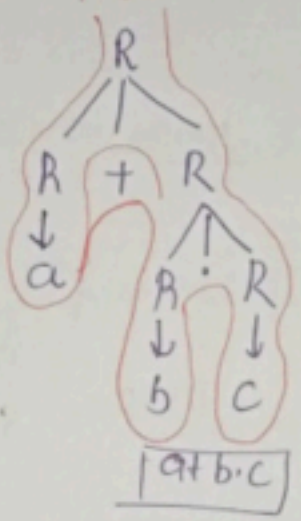


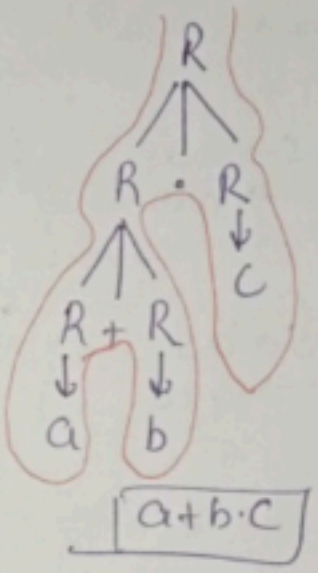


Eg: 5

$R \to R+R \mid R \cdot R$
$R \to a \mid b \mid c$

$w = id \cdot a+b \cdot c$





So, since we have more than 1 parse Tree possible
∴ All these Example of Ambiguous Gramme

Ambiguous means:- if we have 2 parse trees, then Parser will get confus about which one to generate / or which one is correct parse tree.

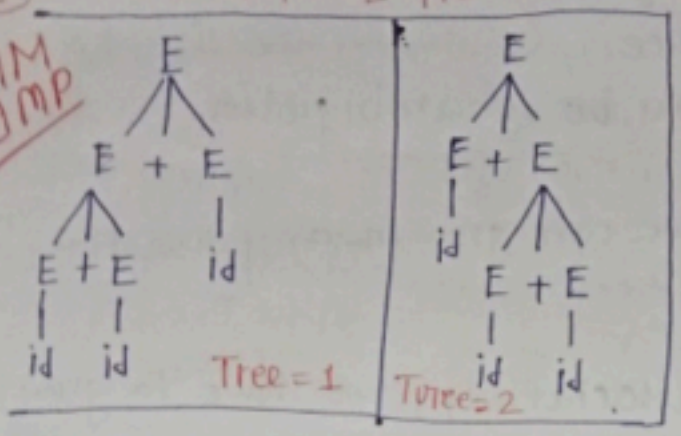
So Parser donot allow ambiguous grammer so we convert ambiguous grammer into Unambiguous Grammer.

* Converting Ambiguous Grammer to Unambiguous Gramm

Eg 1 $E \to E+E \mid E*E \mid id$

$w = id+id+id$

MM IMP



id = 1
id = 2
id = 3

$((1+2)+3) = 3+3 = 6$

$[1+(2+3)] = 1+5 = 6$

It has 2 operators on either side of it. which operator should associate.

If we associate with Left operator

Left associativity $\Rightarrow [(id+id)+id]$

Right associativity $\Rightarrow [id+(id+id)]$

So + → Left associative

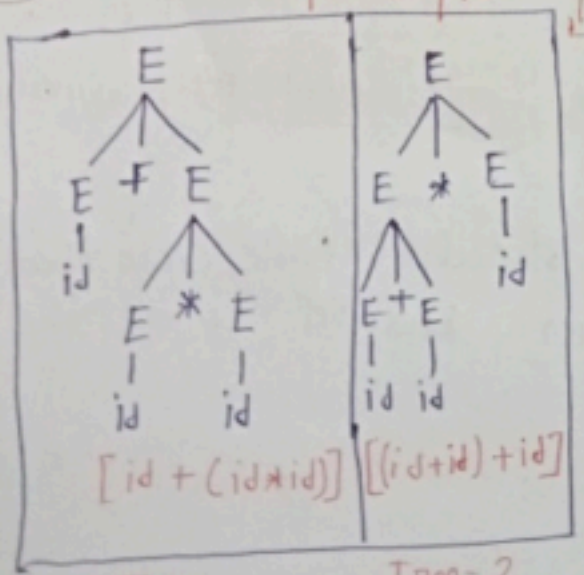
1st Parse Tree correct

Rules of Associ failed

MMIMP

Eg 2 $E \to E+E \mid E*E \mid id$

$w = id+id*id$



Tree-1 Tree-2

$[id+(id*id)]$

$[1+(2*3)]$

$1+6=7$

Valid Tree 1

$[(id+id)*i$

$(1+2)*3$

$3*3=9$

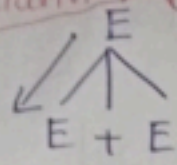
Not valid

Rules of Precedance is not taken care of

IMP

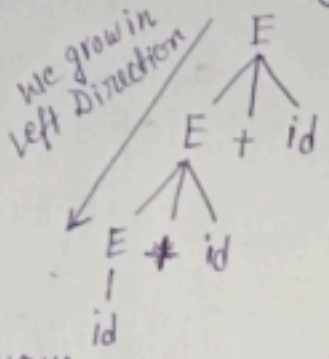
So, if we can take care of
(i) Associativity &
(ii) Precedence
Then Grammer can be unambiguous.

Associativity problem



So, we can grow in any direction

To achive Left associativity, we have to grow in Left Direction only.



$E \rightarrow E+id \mid id$

We are growing only in left direction because grammer is defiend as Left Recursive.

Nonambiguous → $E \rightarrow E+T \mid T$, $T \rightarrow id$

So, there is no possibility of getting different parse Tree.

If we want operator to be Left Associative we have to see that grammer is Left Recursive always.

Precedence Problem

In this, we should take care that highest precedence operator should be at the least level.

E → E+E Grammer is ambiguous
E → E*E
E → id

w = id+id*id

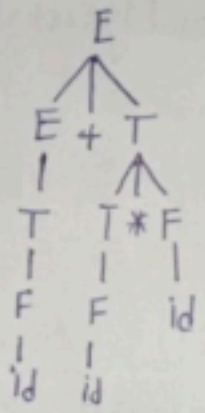
MMSMP

Convert into unambiguous
[+, *], both operators are Left associative so tree will be grow in Left direction or Left recursive.

E → E+T / T
T → T*F / F
F → id

} Once we have reached T, we cannot generate any '+' operator

Now id+id*id become a tree



Now Grammer is Unambiguous

IMP
We can convert some ambigious grammer into unambigious grammer
But we can't convert all ambigious grammer into unambigious.
So it is UNDECIDABLE Problem

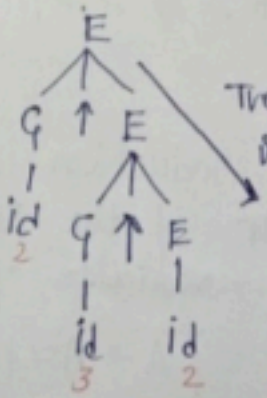
Eg:- w = [2 ↑ (3 ↑ 2)] ⇒ $2^{3^2} = 2^9$ {↑ - is Right associative}
so tree should grow in right direction..

E → E ↑ E
E → (id

[id ↑ (id ↑ id)]

E → G ↑ E / G
G → id

this Grammer will be Right recursive



Tree grow in right direction

Imp.

+L > +R
*L > *R
↑L < ↑R

Eg:- 2

bExp → bExp OR bExp
bExp → bExp AND bExp
bExp → NOT bExp
bExp → True / False

AND, OR, NOT are operators
True & False are terminal

convert into unambiguous Grammer.

Precedence

NOT > AND > OR

Associativity

OR } Left associative
AND }

E → E OR F | F
F → F AND G | G
G → NOT G | True | False

} Unambiguous Grammer

Problem:-

A → A $B | B
B → B #C | C
C → C @D | D
D → d

What is the associativity of precedance.

Soln :- { $, #, @ } all are left associative

for precedance operater which is far from starting symbol.

{ @ > # > $ }

@ => having more precedance.

LEFT RECURSION:-

A production of grammer is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS.

i.e A → Aα | β

Left recursion

Direct LR: $B \to Ba$

Indirect LR: $S \to Aa$, $A \to Sm$

How to remove Left Recursion:-

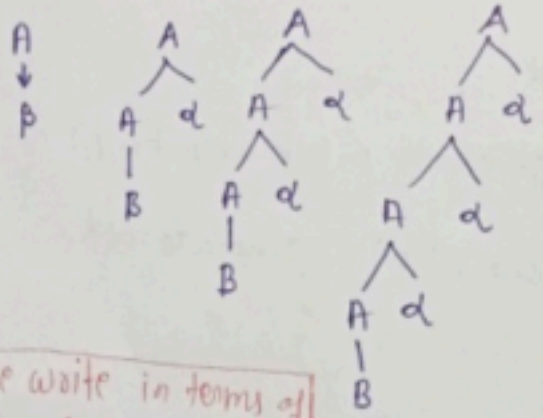
Why To remove ?

Because of it can create an infinite loop, leading errors & a significant decrease in performance. Top Down Parser cannot accept the grammer having Left recursion.

So, we have to remove Left recursion but preserve the Language generated by grammer.

Left recursion Problem

$A \to A\alpha \mid B$



$[\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha \ldots$

$[\beta\alpha^0, \beta\alpha^1, \beta\alpha^2, \beta\alpha^3 \ldots \beta\alpha^*$

Language $= \beta\alpha^*$

If we write in terms of function

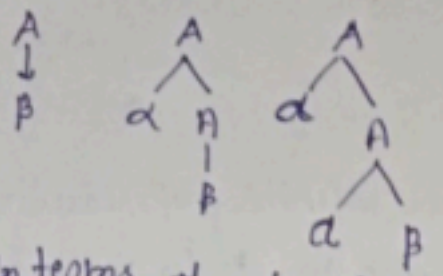
```
A()
{
   A()
     α;
}
```

Create Infinite loop.

* Optimization of REP-Based Pattern Matcher

Right Recursion :-

$A \rightarrow \alpha A | \beta$



$[\beta, \alpha\beta, \alpha^2\beta \ldots]$

$\alpha^*\beta$

In terms of function :-

```
A()
{
  α;
  A();
}
```

Remove Left Recursion :-

$A \rightarrow A\alpha | \beta$

Language :

$A \rightarrow \beta\alpha^*$

$A \rightarrow \beta A'$ } $\Rightarrow$ A will generate β followed by A'

$A' \rightarrow \alpha A' | \epsilon$ } $\Rightarrow$ now A' will generate α^*

Formula to remove LR

$$\boxed{\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' | \epsilon \end{array}} \equiv \boxed{A \rightarrow A\alpha | \beta}$$

This right recursive grammer is equivalent to this left recursive grammer.

So, to eliminate left recursion, we will follow these rules.

Eg: Convert LR grammer into right recursive or Eliminate Left Recursion.

$E \to E+T \mid T$

Now we will compare it with

$A \to A\alpha \mid \beta$

$\underset{A}{E} \to \underset{A}{E} \underset{\alpha}{+T} \mid \underset{\beta}{T}$

Formulae

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

Ans

$E \to TE'$

$E' \to \epsilon \mid +TE'$

Eg: Eliminate Left Recursion

$S \to S0S1S \mid 01$

Compare with $A \to A\alpha \mid \beta$

$\underset{A}{S} \to \underset{A}{S} \underset{\alpha}{0S1S} \mid \underset{\beta}{01}$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

$S \to 01S'$

$S' \to \epsilon \mid 0S1SS'$ Ans

Eg: Eliminate Left Recursion

MMGP

$S \to (L) \mid x$ — here no Left recursion.

$L \to L,S \mid S$

$\underset{A}{L} \to \underset{A}{L} \underset{\alpha}{,S} \mid \underset{\beta}{S}$

$L \to SL'$

$L' \to \epsilon \mid ,L'$ Ans

$S \to (L) \mid x$

$L \to SL'$

$L' \to \epsilon \mid ,L'$ Ans

Eg: Eliminate Left Recursion

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

$\Rightarrow$

$E \to TE'$
$E' \to \epsilon \mid +TE'$
$T \to FT'$
$T' \to \epsilon \mid *FT'$
$F \to id$

Eg:- $A \to Aab \mid c$

$\Downarrow$

$A \to cA'$
$A' \to \epsilon \mid abA'$

Eg: $S \to A$
$A \to Ab \mid d$

$S \to A$
$A \to dA'$
$A' \to \epsilon \mid bA'$

Eg: $S \to Sss \mid a$

$S \to aS'$
$S' \to \epsilon \mid sSS'$

When multiple left production will be given then formula of eliminate left Recursion

$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \ldots$

$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \ldots$
$A' \to \epsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \ldots$

Formula

Eg:- $S \to A$
$A \to \underbrace{Ad}_{\alpha_1} \mid \underbrace{Ac}_{\alpha_2} \mid \underbrace{a\beta}_{\beta_1} \mid \underbrace{ac}_{\beta_2}$
$B \to bBC \mid f$

$S \to A$
$A \to aBA' \mid acA'$
$A' \to \epsilon \mid dA' \mid cA'$
$B \to bBC \mid f$

Eg:- $S \to Aa B$
$A \to aA \mid Ba$
$B \to AB \mid b$

This is Indirect recursion $\Rightarrow$

$S \to AaB$
$A \to aA$
$A \to baA'$
$A' \to \epsilon \mid BaA'$
$B \to AB \mid b$

$S \to AaB$
$A \to aA \mid \underbrace{ABa}_{\alpha} \mid \underbrace{ba}_{\beta}$
$B \to AB \mid b$

Now Remove Left Recursion

Eg: $R \to \underbrace{RR}_{A\,\alpha_1} \mid \underbrace{Ra}_{A\,\alpha_2} \mid \underbrace{aR}_{\beta_1} \mid \underbrace{b}_{\beta_2}$

$R \to aRR' \mid bR'$

$R' \to \epsilon \mid RR' \mid aR'$

Eg:-

$S \to Aa \mid b$

$A \to Ab \mid Sc \mid \epsilon$

$S \to Sca \mid Aba \mid a$

$A \to Ab \mid Sc \mid \epsilon$

* Eliminate LR

$S \to AaB$

$A \to Ab \mid Aac \mid bc \mid \epsilon$

$S \to Aa \mid b$

$A \to bCA' \mid A'$

$A' \to \epsilon \mid bA' \mid acA'$

Eg: $S \to aBDh$, $B \to Bb \mid R$, $D \to EF$, $E \to g \mid \epsilon$, $F \to f \mid \epsilon$

$\Rightarrow$

$S \to aBDh$

$B \to hB'$

$B' \to \epsilon \mid bB'$

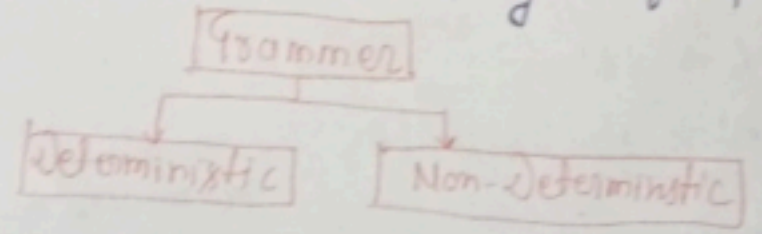
$D \to EF$

$E \to g \mid \epsilon$

$F \to f \mid \epsilon$

LEFT FACTORING :-

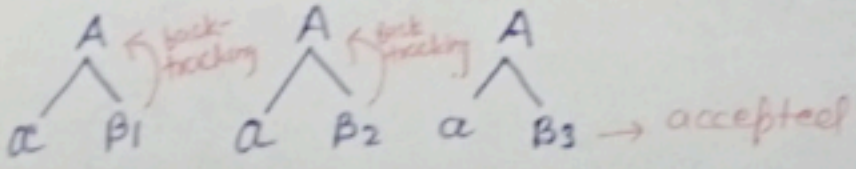
Sometimes, it is not clear th which production to choose to expand a non-terminal becaus Multiple productions begin with the same [terminal / non terminal / Lookahead.] This type of Grammer known as Non-Deterministic gramma an Grammar containing Left factoring.



$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

accept $\omega = \alpha\beta_3$

Suppose we have to



Here common prefix is 'α'